DDF_L4

September 24, 2020

1 Eksplorative metoder i Python I: Deskriptive metoder

En central del af analysearbejde er at blive bekendt med sit datasæt. Det gælder både indhold og umiddelbare mønstre, som kan udledes.

I løbet af denne session introduceres en række metoder til at udforske og beskrive umiddelbare mønstre i datasættet. Undervejs introduceres forskellige måder at håndtere forskellige dataudfordringer; herunder håndtering af missingværdier, kategoriske variable samt tilføjelse af data (observationer og variable).

Der arbejdes igen med data fra European Social Survey fra 2014 (https://www.europeansocialsurvey.org/data/download.html?r=7).

2 Split-apply: Beskriv grupper i data

En effektiv måde at udforske og udføre indledende deskriptive analyser af sine data, er ved at undersøge de grupperinger, som data indeholder.

Her skal vi se på nogen forskellige måder, hvor der kan udregnes deskriptive mål og dannes visualiseringer for grupper i data

Vi starter med at indlæse de nødvendige pakker og data. Aldersvariablen (som også blev tilføjet i lektion 3) tilføjes igen:

[ ]:
```
import pandas as pd
import numpy as np

df = pd.read_csv('https://github.com/CALDISS-AAU/workshop_python-data-analysis/
 ↪raw/master/datasets/ESS2014DK_sub1.csv')
df['age'] = 2014 - df['yrbrn']
```

2.1 Brug af split-apply til at opsummere data

Metoden `.groupby()` grupperer datasæt efter de givne variable:

[ ]:
```
grouped_df = df.groupby(['gndr'])
```

Selve objektet, som bliver dannet, indeholder ikke information, der bare kan kaldes frem direkte:

[ ]:
```
grouped_df
```

Dog kan vi danne de samme deskriptive mål, som vi kan for hele datasættet, men hvor de opsummeres på gruppeniveauet:

[ ]:
```
grouped_df.mean()
```

Det samme kan gøres for enkeltvariable:

[ ]:
```
grouped_df['height'].mean()
```

En lang række metoder kan bruges på grupperede data. Se dem alle her: https://pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html.

- `mean()`: Middelværdi for grupperne
- `size()`: Størrelse af grupperne
- `count()`: Tællinger inden for grupperne
- `describe()`: Deskriptive mål inden for grupperne
- `min()`: Minimum for grupperne
- `max()`: Maximum for grupperne

Metoden `value_counts()` kan bruges til at lave optællinger af kategoriske inden for grupperne.

Herunder laves optælling for variablen `health` fordelt på køn:

[ ]:
```
grouped_df['health'].value_counts()
```

2.2 Visualisering af grupperinger

Grupperede data og optællinger af grupperede data kan plottes direkte:

[ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns

sns.set(rc={'figure.figsize':(15,9)}) # Ændrer standardstørrelsen på plots

grouped_df['health'].value_counts().plot.bar()
```

Ovenstående plot er ikke så kønt. Fx kunne det give mening at farvelægge efter værdierne i `health`. Det kan gøres ved at bruge metoden `unstack()` og så danne plottet:

[ ]:
```
grouped_df['health'].value_counts().unstack('health').plot.bar()
```

2.2.1 Sidenote: Hvad gør `unstack`?

`unstack()` er en metode, der bruges til såkaldt "wide"-konvertering.

Hvis vi først ser på `value_counts()`, gives tællinger i long format; altså hvor hver række er en optælling, og optællingerne differentieres med en kolonne (`health`).

[ ]:
```
grouped_df['health'].value_counts()
```

`unstack()` tager værdierne fra `health` og laver dem til kolonner. På den måde dannes der kun én række per gruppe, hvor værdier adskilles i kolonnerne (wide-format):

[ ]:
```
grouped_df['health'].value_counts().unstack('health')
```

Når sådan en tabel plottes med `.plot.bar()` farvelægges der efter kolonneværdierne automatisk.

[ ]:
```
grouped_df['health'].value_counts().unstack('health').plot.bar()
```

2.2.2 Rekodning af kategoriske

Rækkefølgen på kategorierne i `health` i ovenstående er lidt tosset...

VIDENSCHECK: *Hvorfor er de det?*

For at værdierne vender rigtigt, skal værdierne kodes om sådan, at de er i korrekt rækkefølge, når sorteret alfabetisk (der er andre løsninger, men dette er den umiddelbart nemmeste).

Der kodes her sådan, at hver kategori får en numerisk værdi foran sig, der angiver rækkefølgen. Dette gøres med `.replace()`:

[ ]:
```
df['health'] = df['health'].replace({'Very bad': '1: Very bad', 'Bad': '2:␣
↪Bad', 'Fair': '3: Fair', 'Good': '4: Good', 'Very good': '5: Very good'})

df['health'].unique()
```

Derefter dannes grupperingen på ny for at få ændringen med:

[ ]:
```
grouped_df = df.groupby('gndr')
```

Kategorierne står nu i rigtig rækkefølge i plottet.

[ ]:
```
grouped_df['health'].value_counts().unstack('health').plot.bar()
```

3 ØVELSE: Hvordan fordeler glæde sig på tværs af køn?

I skal i denne øvelse få et overblik over, hvordan fordelingen i glæde (variablen `happy`) er forskellige på tværs af køn.

1. Grupper data efter `gndr` med `.groupby()`

2. Brug `value_counts()` til at optælle fordelingen i variablen `happy`.
3. Brug `unstack()` og `plot.bar()` til at visualisere fordelingen.

BONUS ØVELSE Kategorierne i `happy` står i forkert rækkefølge. Løs dette ved at rekode værdierne sådan, at værdierne står i rækkefølge.

3.1 Sidenote: Tilføj til plottet

Vi har nu dannet et meget fint plot, men vi kan gøre mere!

Vi kan fx ændre aksetitlerne til at være mere sigende med `xlabel()`/`ylabel()`.

Til sidst kan vi tilføje en sigende titel med `plt.title()`.

[ ]:
```
grouped_df['happy'].value_counts(sort = False).unstack(['happy']).plot.bar()
plt.xlabel("Gender")
plt.ylabel("Count")
plt.title("Count distribution of 'happy' across gender")
```

4 Gruppering efter flere værdier

`groupby()` kan grupperer efter lige så mange variable, som man har lyst til, blot ved at specificere flere variable. Herunder grupperes efter `gndr` og en kategorisk variabel for alder (`agecat`), som dannes med `pd.cut()`:

[ ]:
```
df['agecat'] = pd.cut(df['age'], 5)
grouped_df = df.groupby(['gndr', 'agecat'])
```

Igen kan forskellige deskriptive mål dannes. Ved at bruge `size()`, får vi optællinger for hver gruppering af 'gndr' og 'agecat':

[ ]:
```
grouped_df.size()
```

Bemærk at ovenstående output svarer til at gruppere efter `gndr` og bruge `value_counts()` på `agecat`. Igen kan vi derfor plotte denne optælling med `plot.bar()` og bruge `unstack()` til at farvelægge værdierne:

[ ]:
```
grouped_df.size().unstack('agecat').plot.bar()
```

Da data er grupperet efter både `gndr` og `agecat`, kan vi også danne mål ud fra andre variable; fx middelvægten inden for hver gruppe:

[ ]:
```
grouped_df['weight'].mean()
```

Det er igen den samme type output, så også dette kan vi plotte:

[ ]:
```
grouped_df['weight'].mean().unstack('agecat').plot.bar()
plt.xlabel("Gender")
```

```
plt.ylabel("Mean weight")
```

4.1 Sidenote: Grupperinger med `seaborn` funktioner

Mange `seaborn` funktioner indeholder funktionalitet til at kunne foretage grupperinger i et plot.

`sns.catplot()` er en "overfunktion" til at danne forskellige typer af kategoriske plots. Typen specificeres med argumentet `kind =`. `kind = "count"` giver et barplot, der tæller antal inden for hver kategori:

[ ]:
```
ageplot = sns.catplot(data = df, kind = "count", x = "agecat")
ageplot.set_xticklabels(rotation=45)
```

Plottes kan "splittes ud" ved at bruge argumenter som `hue =` og `col =`.

- `hue`: splitter værdier op i søjler for hver variabel givet i hue
- `col`: danner graf for hver værdi i col

Herunder dannes bargrafer for fordelingen i health for hver alderskategori (bemærk at værdier sorteres efter `health` for at få søjlerne i rigtig rækkefølge):

[ ]:
```
healthplot = sns.catplot(data = df.sort_values('health'), kind = "count", x =␣
↪"health", hue = "gndr", col = "agecat")
healthplot.set_xticklabels(rotation=45)
```

5 ØVELSE: Flere grupperinger

1. Dan en kategorisk variabel, der inddeler respondeter i 3 lige store grupper efter vægt (brug `pd.cut()`)
2. Dan en visualisering, der viser fordeling af svar i `alcfreq` splittet ud på køn og vægtkategori (brug enten seaborn eller `.plot` på grupperet data (`.groupby()`)

6 Håndtering af missing værdier

Vi har i ovenstående udforsket data på forskellig vis, men vi har ikke al data med… nogen observationer er "missing" (`NaN`). Det kan ses, når vi sammenligner `count()` på en variabel med antallet af observartioner i data:

[ ]:
```
print(f"Data indeholder {df.shape[0]} rækker, men variablen 'weight' har kun␣
↪{df['weight'].count()} observationer. 'weight' har derfor {df.shape[0] -␣
↪df['weight'].count()} missingværdier")
```

Missingværdier håndteres overordnet på to måder:

- Rækker med missingværdier fjernes ("listwise deletion")
- Værdier "imputeres"; dvs. erstattes med en bestemt værdi.

Der findes et hav af avancerede teknikker til at imputere. I denne session gennemgås, hvordan missing fjernes eller erstattes med enkelte værdier.

6.1 Hvor er der missing?

Metoderne `isnull()` og `notnull()` bruges til at tjekke, om hhv. værdier er missing eller værdier ikke er missing.

Kombineret med `sum()` kan det optælles, hvor mange missingværdier der er i fx `weight`:

[ ]:
```
weight_miscount = df['weight'].isnull().sum()

print(f"Der er {weight_miscount} missingværdier i 'weight'")
```

Optælling af missing kan også laves for hele datasættet:

[ ]:
```
df.isnull().sum()
```

6.2 Fjern eller erstat missing

En pandas dataframe (og serie) har flere indbyggede metoder til at håndtere missing. De mest simple er:

- `.dropna()`: listwise deletion af observationer, som indeholder missing værdier
- `.fillna()`: erstatter missing med en angiven værdi

Derudover findes også disse metoder, som særligt er egnet til tidsserie data:

- `ffill()`: erstatter missing værdi med værdien i næste række eller kolonne
- `bfill()`: erstatter missing værdi med værdien i forrige række eller kolonne

6.2.1 Fjern missing

Lad os starte med at fjerne missing. Vi fjerner missing i en kopi af datasættet, for ikke at miste information:

[ ]:
```
df_nomis = df.dropna()

print("Det oprindelige datasæt har", df.shape[0], "rækker. Datasæt uden missing␣
↪har", df_nomis.shape[0], "rækker")
```

Ovenstående viser tydeligt, hvorfor det hurtigt kan blive uhensigtsmæssigt bare at fjerne rækker med missing: Der mistes alt for megen information.

Man kan evt. indskrænke sådan, at rækker kun fjernes, hvis de er missing i bestemte varialble (herunder for `weight`):

```
[ ]: df_nomis = df.dropna(subset = ['weight'])

print("Det oprindelige datasæt har", df.shape[0], "rækker. Datasæt uden missing␣
 ↪har", df_nomis.shape[0], "rækker")
```

Man kan også sætte en tærskel for, hvor mange ikke-missing, der skal være, før rækken fjernes:

```
[ ]: df_nomis = df.dropna(thresh = 10) # Der skal være 10 ikke-missing værdier per␣
 ↪række

print("Det oprindelige datasæt har", df.shape[0], "rækker. Datasæt uden missing␣
 ↪har", df_nomis.shape[0], "rækker")
```

6.2.2 Erstat missing

Som alternativ til at fjerne missing, kan missingværdier erstattes med en bestemt værdi.

BEMÆRK!: Der er ikke en gængs løsning for, hvordan missingværdier erstattes (hvis det overhovedet giver mening at erstatte). At erstatte missing skal afvejes ift. den analyse man laver, hvilke data, der kan imputeres ud fra og hvilken type information, som man imputerer.

Der vises her simple måder at imputere, men om de kan bruges afhænger altså af, hvilken metode man anvender.

`.fillna()` erstatter missingværdier med en bestemt værdi. Herunder erstattes missingværdier i `cgtsday` med 0:

```
[ ]: df['cgtsday'].fillna(0).head(10)
```

Ovenstående ændrer ikke data. Så skal vi skrive datasættet over; enten ved at tilskrive det igen eller bruge argumentet `inplace = True`:

```
[ ]: df['cgtsday'] = df['cgtsday'].fillna(0)

df.head()
```

Hvis værdier skal kodes om til missing, kan dette gøres med `replace()`.

I nedenstående rekodes 0 til missing i `cgtsday`. *BEMÆRK!*: Der var 0-værdier i variablen *inden* de blev kodet til missing, så dette ændrer ikke variablen tilbage, som den var:

```
[ ]: df['cgtsday'] = df['cgtsday'].replace(0, np.NaN)

df.head()
```

Erstat missing med middelværdi Ofte handler imputation (erstatning af missing) om at få observationerne til at "fylde" så lidt som muligt. Det kan fx gøres ved at erstatte missing med middelværdien for variablen.

Igen bruges `.fillna()`:

```
[ ]: print(df['height'].isnull().sum()) # Hvor mange missing er der i height?

mean_height = df['height'].mean() # Middelhøjde lagres for sig

df['height_imputed'] = df['height'].fillna(mean_height) # Missing erstattes med␣
 ↪middelhøjde

df.loc[df['height'].isnull(), ].head() # Kontrol for at værdier er imputeret
```

Erstat missing med brug af en funktion og betingelser Betingelser kan bruges til at differentiere ens imputation en smule. Det kan gøres i kombination med en funktion, som selv skrives, så middelværdi af højde kan tildeles inden for køn.

```
[ ]: mean_height_m = df.loc[df['gndr'] == "Male", 'height'].mean()
mean_height_f = df.loc[df['gndr'] == "Female", 'height'].mean()

def height_fill(gndr):
    if gndr == 'Male':
        return(mean_height_m)
    elif gndr == 'Female':
        return(mean_height_f)

df['height_imputed'] = df['height'].fillna(df['gndr'].map(height_fill))

df.loc[df['height'].isnull(), :].head()
```

7 VIDENSCHECK

Giver det mening at imputere/erstatte missing i `cgtsday` med middelværdien?

8 ØVELSE: Håndtering af missing

I skal lave jeres visualisering fra sidste øvelse igen (en visualisering, der viser fordeling af svar i `alcfreq` inden for hver gruppering af køn og vægtkategori), men denne gang vil vi gerne have nogen af missingværdierne i `weight` med.

1. Dan en `weight_imputed` variabel, hvori missingværdier for `weight` erstattes med middelvægten.
2. Dan vægtkategorien igen med `weight_imputed` variablen
3. Dan visualiseringen igen

BONUSØVELSE

Sammenlign fordelingerne i `alcfreq` indenfor hver køn og vægtkateogi før og efter imputering/erstatning af missingværdi - Er der forskel?

9 Kan det blive vildere?

Som sagt kan man gruppere efter flere variable. Ved at kombinere viden fra de tidligere eksempler, kan vi producere et plot, der viser fordelingen i selvvurderet helbred summeret til 100% for grupperinger i køn og alder.

Følgende gøres:

- For at udregne procenter, skal vi bruge gruppestørrelserne for `gndr` og `age_cat` uden missing. `grouped_sizes` dannes, som først fjerner missing, hvis der er missing i de tre variable, som skal bruges. Derefter grupperes efter `gndr` og `age_cat` og til sidst tælles gruppestørrelserne med `.size()`.
- Dernæst dannes gruppering med den trejde variabel `health`: Vi har nu tællinger for selvvurderet helbred per alderskategori per køn
- De to objekter divideres med hinanden: Antal i selvvurderet helbredskategori inden for gruppen divideret gruppens størrelse. `health` unstackes.
- Til sidst dannes grafen med `stacked = True` for at få et stablet barplot

[ ]:
```
grouped_sizes = df.dropna(subset = ['gndr', 'agecat', 'health']).
 ↪groupby(['gndr', 'agecat']).size()
grouped_df = df.groupby(['gndr', 'agecat', 'health'])

grouped_health_pct = (grouped_df.size() / grouped_sizes).unstack('health')

grouped_health_pct.plot.bar(stacked = True)
```

10 Crosstabs: Men kan det blive... VILDERE!?

Krydstabulering er en relativt ufarlig "leg" vi kan lege med kategorisk data når vi gerne vil have et overblik over, hvad der foregår mellem 2 variable. Heldigvis er standardformatet for krydstabulering simpelthen enormt let i Pandas. Man behøver ikke specificere andet end dataframe og de to variable man vil vide noget om:

[ ]:
```
pd.crosstab(df['health'], df['agecat'])
```

Vi kunne også få den ide, at vi måske skulle tilføje nogle totaler:

[ ]:
```
pd.crosstab(df['health'], df['agecat'], margins=True, margins_name="Total")
```

Eller måske skulle vi proppe nogle procenter på?

[ ]:
```
pd.crosstab(df['health'], df['agecat'], normalize='columns') # Til␣
 ↪kolonneprocenter
```

Vi kan også presse citronen lidt og gøre det endnu mere fedt (eller forvirrende, alt afhængig af hvem man er) ved at tilføje endnu mere data:

[ ]:
```
pd.crosstab(df['agecat'], [df['health'], df['gndr']])
```

Problemet med at "presse citronen" er at tabeller ret hurtigt kan blive alt for vilde og uoverskuelige. Vi ender med en situation hvor vi ikke helt forstår hvor "magien" sker. Heldigvis kan Seaborn hjælpe os med en visualisering! Bemærk dog, at lige præcis DEN her version laver pjatrøv i top og bund. Matplotlib version 3.1.1 har ligesom..... smadret heatmaps en lille smule, så den irriterende halvering af øverste og nederste række holder op hvis vi bruger matplotlib 3.1.0 eller 3.1.2.

[ ]:
```
sns.heatmap(pd.crosstab(df['agecat'], [df['health'], df['gndr']]),␣
↪cmap="YlGnBu", annot=True, cbar=False, fmt="d")
```

Måske er vi lidt for glade for ren visualisering og gider ikke de der irriterende tal? Look no further!

[ ]:
```
sns.heatmap(pd.crosstab(df['agecat'], [df['health'], df['gndr']]),␣
↪cmap="YlGnBu", annot=False, cbar=True)
```

11 ØVELSE: Krydstabel

Dan en krydstabel mellem `alcfreq` og vægtkategori-variablen fra sidste øvelse.

Kombiner evt. med seaborns heatmap-funktion.

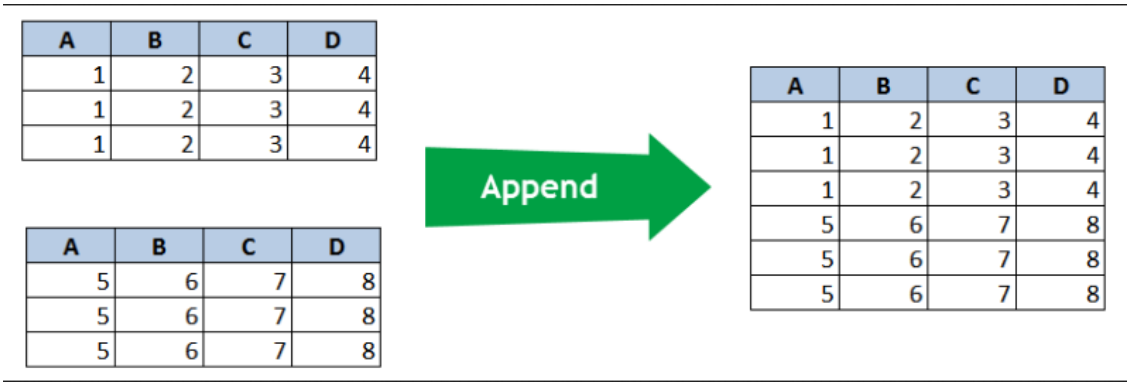
12 Kombinering af datasæt i Python

I nogle tilfælde bliver vi nødt til at tilføje data til vores eksisterende datasæt. Der findes mange forskellige metoder til at tilføje data men de to primære er concatenate (også kaldet append) or merge. Først skal vi kigge lidt nærmere på concatenate.

13 Tilføj rækker (concatenate)

Concatenate bruger vi når vi har de samme variable men ønsker at tilføje flere observationer til dem.

Concatenate ser sådan her ud:



I praksis betyder det, at vi er afhængig af, at vi har de samme variable men flere observationer af disse variable.

Hvis vi har "for mange" variable i det ene eller andet datasæt, bliver de missing (eller NAN) i det datasæt hvor variablene ikke er til stede.

13.1 Eksempel på concatenate

Concatenate i Python er rimelig ligetil, men før vi gør noget som helst, skal vi importere de pakker vi har brug for.

[ ]:
```
import pandas as pd # Pandas pakken til datamanipulation og analyse
import numpy as np # "Fancy" videnskabelig computation

#Og så gør vi lige noget fnidder-fnadder der er ligemeget:
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Når vi har vores pakker er det bare at losse data ind. I stedet for, som vi normalt gør, at have én dataframe kan vi nu udnytte, at Python er helt ok med, at vi har lige så mange dataframes som computerens (eller serverens i dette tilfælde) kan holde til:

[ ]:
```
df1 = pd.read_csv("https://github.com/CALDISS-AAU/workshop_python-data-analysis/
↪raw/master/datasets/ESS2014DK_sub1.csv")
df2 = pd.read_csv("https://github.com/CALDISS-AAU/workshop_python-data-analysis/
↪raw/master/datasets/ESS2014DK_sub2.csv")
```

Lad os først tjekke første dataframe (df1):

[ ]:
```
df1.head()
```

Og den anden:

[ ]:
```
df2.head()
```

Ved at lave en hurtig sammenligning kan vi se, at vi har samme datasæt men at respondenterne varierer (idno). Vi kunne pludselig tænke os, at det stoppede med at være to forskellige frames men i stedet én samlet. Med pandas er det heldigvis rimelig ligetil:

[ ]:
```
conc_data = pd.concat([df1, df2])
```

Mere er der ikke til det. Vi kan lige hurtigt sammenligne hvordan data så ud før:

[ ]:
```
print("Info on first dataset:")
df1.info(verbose=False)
print("")
print("Info on second dataset:")
df2.info(verbose=False)
```

For så derefter lige at sikre os, at vi har dobbelt op ved at plusse de to værdier i RangeIndex for df1 og df2 og sammenligne med RangeIndex for conc_data

[ ]:
```
conc_data.info(verbose=False)
```

Vi kan også, på en ret kluntet måde, hvis vi virkelig ikke kan overskue at plusse de to værdier for antal entries (rækker) fra df1 og df2 med conc_data, tjekke det automatisk

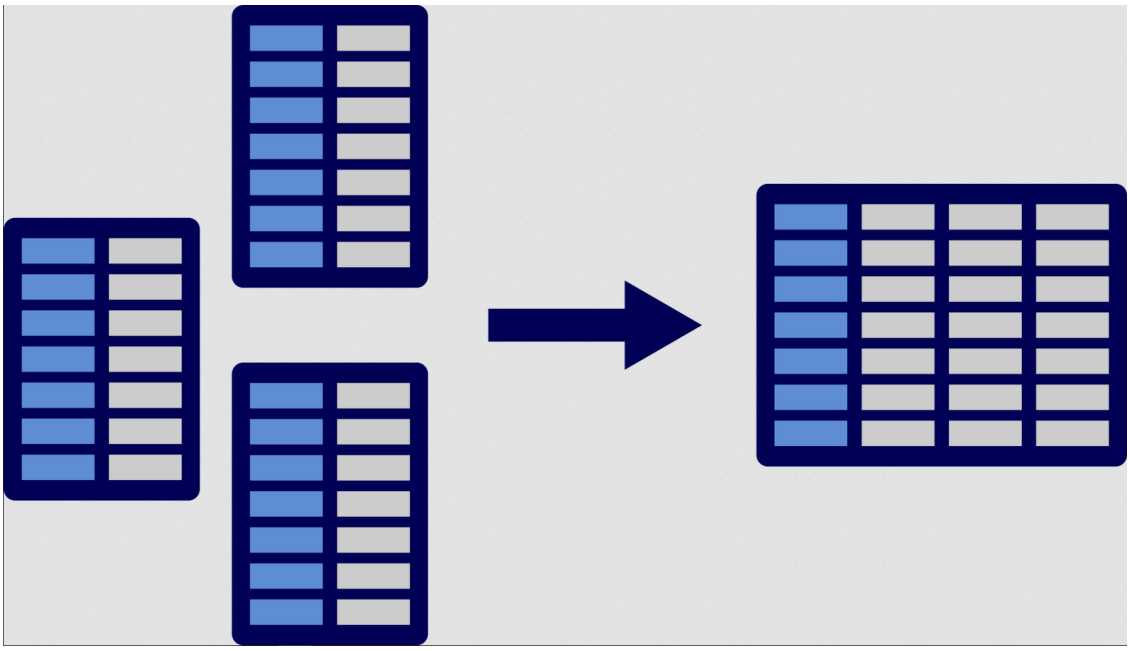
[ ]:
```
count_df1=len(df1)
count_df2=len(df2)
total_dfs=count_df1+count_df2
count_conc_data=len(conc_data)
print("The number of observations in df1 is", count_df1, "and", count_df2, "in␣
 ↪df2.")
print("The total count in conc_data is", count_conc_data, "and the total of df1␣
 ↪and df2 is", total_dfs )
print("It is", total_dfs==count_conc_data, "that the concatenation worked as we␣
 ↪hoped")
```

Vi er glade for vores concatenation - job well done

14 Joins / Merge

Hvad er et join (som jeg herefter kalder merge fordi…. det gør jeg bare)?

Et merge minder om concatenate i den forstand, at vi tilføjer data til et eksisterende datasæt. I dette tilfælde vil vi dog ikke putte flere observationer på eksisterende variable, men i stedet tilføje flere variable til eksisterende observationer. Det ser sådan her ud:



Når vi ved det, lad os så inspicere vores nye data. Vi laver først vores dataframes:

[ ]:
```
df1m = conc_data.copy()
df2m=pd.read_csv("https://github.com/CALDISS-AAU/workshop_python-data-analysis/
↪raw/master/datasets/ESS2014DK_trstsub.csv")
```

[ ]:
```
df1m.head()
df1m.info(verbose=False)
```

[ ]:
```
df2m.head()
df2m.info(verbose=False)
```

Nu har vi data. Det vi smadder gerne vil nu er, at undersøge hvad det er for noget data vi har til rådighed og om der er en **nøgle**

Nøgler er det der indikerer, at to ting hører sammen. Som tidligere, så bruger vi ESS, men nu er der en eller anden der har tævet data over i to på langs. Nogle variable er i df1m mens andre pludselig ligger i df2m. Vi vil smadder gerne have dem til at bo som ét datasæt og ikke som to.

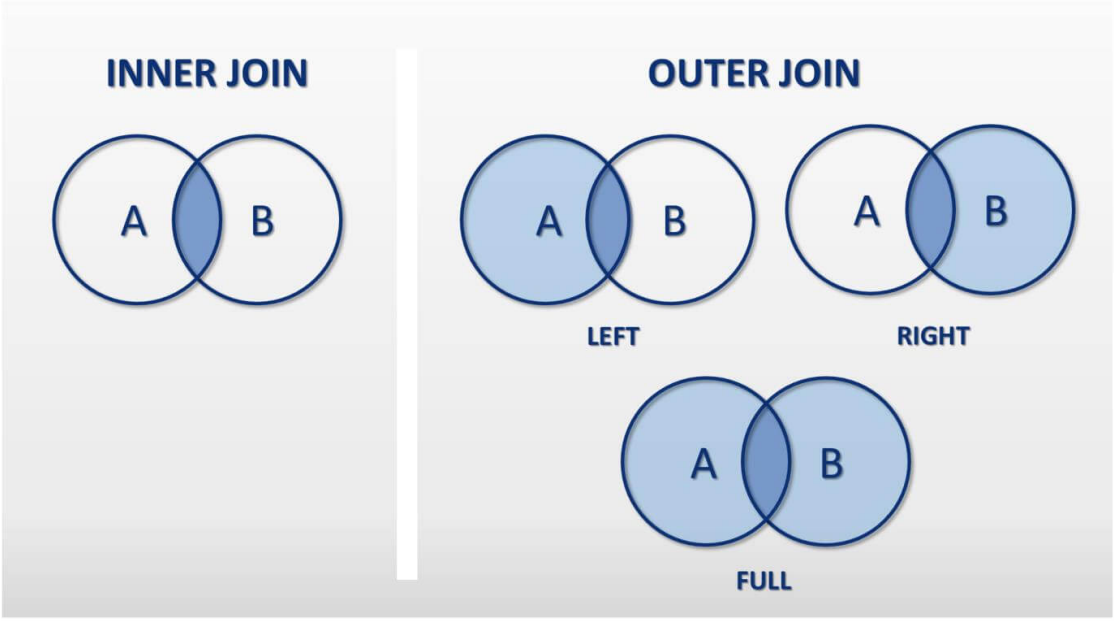
Python/Pandas kan ikke "gætte" hvad der hører sammen og vi kan ikke bare håbe, at den nok selv regner det ud. I stedet kan vi håbe på, at der er en **nøgle**.

I vores tilfælde er vi heldige; vi har et id-nummer - `idno` - der fortæller os hvem der hører sammen med hvem.

Før vi begynder at lege med reelt set at smide tingene sammen i en pærevælding skal vi nødt til at kende forskel på de forskellige merges man kan lave:

- Left outer
- Right outer
- Inner
- Outer

For at forstå hvad præcis de 4 "metoder" gør når vi merger er det nemmest at se, hvad resultatet bliver for dem visuelt… i hvert fald de sidste to.



Et left join (den dataframe der er til venstre - altså først) er den data bliver proppet på. Hvis der forekommer 100 værdier i den venstre dataframe og 200 i den anden kigger den udelukkende efter nøglerne i den første dataframe og ignorerer dem i den anden dataframe der er "for mange" og smider væk men kun dem der er for mange i det højre. Er der nogen i det venstre der ikke eksisterer i det højre bliver de bibeholdt. Den prioriterer ligeledes udelukkende nye variable i det højre datasæt og alt der der allerede eksisterer i den venstre dataframe bliver smidt væk fra den højre.

Sjovt nok er right merge det samme som left bare den anden vej rundt….

Til gengæld er der voldsom stor forskel på inner og outer joins.

Et inner join tager udelukkende ting med, hvor nøglerne eksisterer i begge datasæt. Alt andet bliver smidt væk og man har et "rent" datasæt.

Et outer join tager alt med. Dem der eksisterer i det venstre kun, i højre kun og fælles. I de tilfælde hvor de mangler i enten venstre eller højre bliver der hvor de mangler fyldt med missing.

Hvad er kommandoen? Easy-peasy; `pd.merge()`

Hvad er `options`? Mange…

De primære muligheder vi har er:

- on
- left_on / right_on
- how
- validate

Hvis **Nøglen** har samme navn i både df1 og df2 kan vi nøges med at specificere `on`, men hvis nøglen hedder noget forskelligt (f.eks. idno og idnum) kan vi bruge `left_on` til nøglen i første df og `right_on`til det andet.

`How` siger noget om, hvilken af de muligheder vi har (left, right, inner, outer). Vi kan derfor specificere metoden der.

`validate` er lidt anderledes. Validate kan enten tage funktionen `one_to_one`, `many_to_one` eller `one_to_many`.

Forskellen på metoderne er hvad vi vil acceptere vores merge ender med at spytte ud. One to one accepterer kun, hvis der er én observation for hver nøgle i hvert datasæt. Det betyder, at hvis respondenten går igen i enten det ene eller det andet. Der må, med andre ord, kun være én nøgle for hver respondent i hver df. Alt andet resulterer i en `error`.

De to andre er giver lidt mere sig selv; many to one indikerer, at der gerne må være mange nøgler til stede i df1 og at værdierne er unikke i df2. Hver respondent får derfor samme værdi fra df2 proppet på df1. Modsat, hvis der kun er én observation per id i df1 men mange i df2 kan vi specificere one to many i stedet.

Kommandoen ser sådan her ud:

[ ]:
```
df_merge = pd.merge(df1m, df2m, on='idno', how='outer', validate="one_to_one")
df1m.head()
df2m.head()
df_merge.head()
```

15 ØVELSE: Merge/Join med Pandas i Python

Denne øvelse samler op på det meste, som er blevet gennemgået i dag.

Vi vil gerne undersøge fordelingen i tillid til politiet (`trstplc`) på tværs af partitilhør.

1. Vores data indeholder endnu ikke partitilhør, men det kan vi måske gøre noget ved…

   Indlæs datasættet `ESS2014DK_polpartsub.csv` fra 'https://github.com/CALDISS-AAU/workshop_python-data-analysis/raw/master/datasets/ESS2014DK_polpartsub.csv'

   Undersøg datasættet: Er der en nøgle?

   Brug `pd.merge()` til at tilføje `ESS2014DK_polpartsub` til det datasæt, I allerede har.

2. Dan en krydstabel med `pd.crosstab()` over partitilhør (`polpartvt`) og tillid til politiet (`trstplc`)

3. Der er værdier i partitilhør-variablen og variablen for tillid til politiet, som ikke kan bruges til noget. Konverter disse til missing med `replace()` (husk at `np.nan` værdien bruges til at ændre til missing).
4. Dan krydstabellen igen

BONUSØVELSE

1. Undersøg, om fordelingen er forskellig på tværs af køn med groupby
2. Find en passende måde at vise forskellen; enten ved brug af grafer eller ved at udregne middelværdi for tillid til politiet (dette kræver, at variablen konverteres til numerisk)