Intro til Dataanalyse med Python

Notater

Geir Arne Hjelle / Tekna

14. april 2021

Dette dokumentet inneholder notater fra kurset som ble holdt onsdag 14. april 2021 på Teams. Koden er det som ble gjennomgått underveis, med noen ekstra notater og kommentarer. Det gis også noen lenker til artikler som går litt mer i dybden.

Ta gjerne kontakt med Geir Arne på `geirarne@gmail.com` om du har spørsmål, forslag til forbedringer eller andre tilbakemeldinger.

Kurset var essensielt delt i tre deler:

1. Lese inn data fra Excel, enkle analyser med Pandas
2. Grupperinger og koblinger av data
3. Visualisere data på kart

Dette dokumentet følger den samme organiseringen.

1 Kom i gang med dataanalyse i Python

På forhånd hadde alle installert Anaconda Python og gjort noen små oppgaver for å teste at installasjonen fungerte.

> **Merk:** Demonstrasjonen ble gjort på Windows 10 Home på en ny installasjon av Anaconda med følgende versjoner:
>
> - Python 3.8.5
> - Spyder 4.1.5
> - Pandas 1.1.3 (Skriv `print(pd.__version__)`)
> - Folium 0.11.0 (Skriv `print(folium.__version__)`)
>
> Det har kommet nye versjoner av flere av disse programmene og pakkene, inkludert Spyder 5 som ble sluppet tidligere i april. Det som ble gjennomgått i kurset vil fungere på de fleste nyere versjoner.

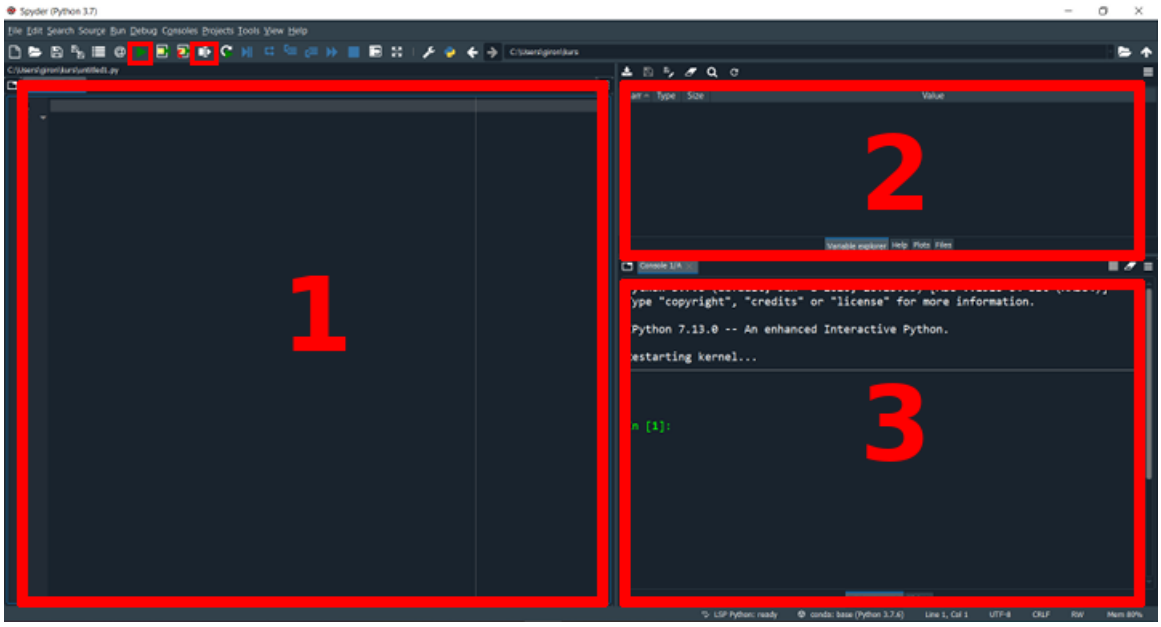


Figure 1: Spyder

Hele kurset foregikk i Spyder. Spyder er en editor som passer godt for dataanalyse, men det meste som ble gjennomgått i kurset kan gjøres i alle editorer.

1.1 Oversikt over Spyder

Spyder består av flere områder som du kan organisere selv. I standardinstillingen ser du tre hovedområder:

1. Editorområdet: Til venstre. Her kan du skrive programmer og andre tekstfiler. Programmene kan du senere kjøre gang på gang.
2. Informasjonsområdet: Øverst til høyre. Her finner du flere faner, inkludert *Variable Explorer* for å se verdien av variable og *Plots* som viser figurer og plott du lager.
3. Konsollet: Nederst til høyre. Her kan du kjøre enkeltstående Pythonkode. Det er også her du vil se resultatet når du kjører programmer gjennom Spyder.

Spyder har også en menylinje og en knappelinje som kan brukes til kjøring av programmer og lignende.

Utforskende arbeidsflyt:

Du kan bruke Spyder ganske effektivt i en utforskende arbeidsflyt hvor du utforsker dataene dine, og undersøker effekten av forskjellig kode.

- Skriv kode i editorvinduet (1). Klikk *Run file*-knappen (F5) for å kjøre all koden i konsollet (3).
- Bruk *Variable explorer* (2) og *Plots* (2) for å undersøke resultatet av koden.
- Skriv enkle kodelinjer i konsollet (3).
- Kjør deler av koden i editorvinduet (1) ved å merke den og klikke *Run selection or current line*-knappen (F9).

Vær oppmerksom på at konsollet "husker" tidligere kode som har blitt kjørt. Dette er nyttig i forhold til at du kan spare mye tid på for eksempel å slippe å lese inn data gang på gang.

Det kan likevel skape noen problemer. For å være sikker på at du ikke er avhengig av tidligere kode som har blitt slettet, bør du en gang i blant:

1. Restarte konsollet (*Consoles* > *Restart kernel*)
2. Kjøre programmet ditt på nytt (*Run file*-knappen)

1.2 Les data fra Excel

Pandas er en kraftig pakke for dataanalyse i Python. Pandas baserer seg på en datastruktur som heter `DataFrame`. En `DataFrame` ligner en del på et regneark, i det at det består av data organisert i rader og kolonner. En `DataFrame` har litt mer struktur som er vanlig, men ikke nødvendig å bruke i regneark:

- Hver "celle" består av en verdi, man kan ikke lagre formler på samme måte som i Excel. I stedet kan bruke funksjoner for å oppdatere verdien i en celle.
- Alle verdier i en kolonne må ha samme datatype, for eksempel dato, tekst, tall, osv.
- Hver kolonne har en tittel
- Hver rad identifiseres av en `index`. Slike indekser må ikke være unike, selv om det ofte er enklere å jobbe med unike indekser.

Mer informasjon:

- Artikkel: *The Pandas DataFrame: Make Working With Data Delightful*
  - realpython.com/pandas-dataframe

For å jobbe med Pandas må du først laste inn pakken:

Python

```
1 import pandas as pd
```

Det er vanlig å forkorte `pandas` som `pd` på denne måten. For å bruke funksjoner definert i Pandaspakken skriver du da `pd` etterfulgt av punktum etterfulgt av navnet på funksjonen.

> **Merk:** I denne første delen brukte vi det samme regnearket som i installasjonsoppgavene. Du kan laste ned regnearket `kap1.xlsx` fra nettsiden statsbudsjettet.no/Revidert-budsjett-2020 ved å klikke **Tallene bak figurene** og deretter Excelarket som er lenket til **Kapittel 1**.

Pandas støtter å lese data fra mange forskjellige filformater. Du kan lese Excelfiler ved hjelp av `pd.read_excel()`:

Python

```
1 data = pd.read_excel("kap1.xlsx")
```

Skriv både `import`-linjen og `.read_excel()` linjen i editorområdet. Klikk deretter *Run file* (F5) for å kjøre programmet ditt. Første gang du gjør dette må du sannsynligvis velge et filnavn (bruk `budsjett.py`) og trykke OK på innstillinger for hvordan filen skal kjøres.

Du vil deretter se `runfile(...)` i konsollet, og etterhvert skal `data` dukke opp i *Variable explorer*. Hvis du i stedet får en feilmelding i konsollet så gjør en dobbelsjekk på at du har skrevet koden riktig. Pass også på at `kap1.xlsx` og `budsjett.py` er lagret i samme katalog.

> **Merk:** Noen har opplevd problemer med å lese inn Excelfilen—spesielt om den var lagret i en skyløsning som Sharepoint. Om du har problemer med å lese inn `kap1.xlsx` kan du prøve å gi en fullstending filsti til filen ved å skrive omtrent `data = pd.read_excel(r"C:\User\GeirArne\python\kap1.xlsx")`. Legg merke til `r` foran filstien. Denne er nødvendig for Windows filstier fordi `\` vanligvis har en spesiell betydning. `r` betyr i praksis at `\` ikke tolkes på en spesiell måte.

Dersom du dobbelklikker på `data` i *Variable explorer* vil du se at vi har lest *noen* data fra regnearket, selv om det kanskje ikke er de mest spennende dataene:

	Dette regnearket inneholder tallene bak figurene i	Unnamed: 1
0	Kapittel 1 Hovedlinjer i den økonomiske politi…	NaN
1	NaN	NaN
2	Figur 1.1	Samlet overskudd i statsbud…
3	Figur 1.2	Finanspolitisk repons på v…

Om du åpner filen `kap1.xlsx` i Excel vil du se at den består av 3 regneark. Dataene du leste inn kommer bare fra det første arket. Videre ønsker vi å bruke arket **1.2**.

For å bruke `.read_excel()` må du oppgi filnavnet. I tillegg finnes det mange andre innstillinger du kan oppgi. Ofte må du bruke flere av disse for å lese dataene riktig, for eksempel for å lese det regnearket du ønsker. Det er flere forskjellige måter du kan få hjelp om Python- og Pandas-funksjoner:

- Hjelp direkte i konsollet ved å skrive `kommando?` eller `help(kommando)`. For eksempel, `pd.read_excel?`
- Dokumentasjon på internett, for eksempel
  - pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html
- Søk i søkemotorer, f.eks. på *pandas read excel*

Fra hvilken som helst av disse metodene vil du etterhvert finne en liste over mulige innstillinger for `.read_excel()` og noe forklaring rundt hva hver av dem gjør.

Det første vi trenger er å lese inn riktig regneark fra filen. `sheet_name` virker som en nyttig kandidat for dette. Endre koden din i editorvinduet:

Python

```python
data = pd.read_excel("kap1.xlsx", sheet_name="1.2")
```

Du kan nå lese dataene på nytt ved å markere linjen du endret og klikke *Run selection or current line*-knappen (F9). Du vil da se at linjen blir kopiert til konsollet (i stedet for `runfile()` som du så tidligere. Om du nå klikker på `data` i *Variable explorer* vil du se litt mer interessante data.

1.3 Spesifiser hvilke data du vil lese

Selv om du nå leser riktig regneark kan du se at dataene fortsatt ser litt "skitne" ut. Det er en del støy rundt tallene som er interessante.

Vi skal legge på litt flere innstillinger. Vi tar dette litt raskere her i notatet, men legg gjerne til hver av disse innstillingene en etter en for å sikre deg at du skjønner effekten av dem og hvorfor det er nyttig å legge dem til. Prøv gjerne også å bytte ut verdiene med andre verdier for å se hvordan dette endrer dataene som blir lest inn:

Python

```python
data = pd.read_excel(
    "kap1.xlsx",
    sheet_name="1.2",
    header=4,
    usecols=[0, 1, 2],
    na_values="-",
    index_col=0,
)
```

Se på `data` i *Variable explorer*, og legg merke til at de nå er ryddige uten unødig støy. Dette betyr innstillingene du har lagt til:

- **`header`:** Fra hvilken rad skal kolonneoverskriftene hentes? Noe som er litt klønete er at Python begynner alltid å telle fra 0, mens rader i Excel er nummerert fra 1. Om du ser i Excelarket vil du se at kolonneoverskriftene ligger i rad 5. Siden Python teller fra 0 betegnes den femte raden som 4.
- **`usecols`:** Hvilke kolonner skal leses inn? Pandas leser vanligvis kun inn kolonnene som inneholder data. I dette arket er overskriften i rad 1 i en spleiset celle som dekker kolonne 2 til 6, mens dataene bare dekker kolonne 1 til 3. Du må derfor oppgi kolonnene eksplisitt. Som for `header` må du justere for at Python teller fra 0. Firkantklammer brukes som notasjon for lister (`list`) i Python. Du kan også bruke Excels notasjon for kolonner: `usecols="A:C"`.
- **`na_values`:** Legg merke til at det mangler en verdi for *Nederland* i det opprinnelige regnearket. Pandas kan tolke en del manglende verdier automatisk. Men her er det brukt `-` som Pandas ikke kjenner til. Du kan fortelle om verdier som skal tolkes som manglende ved å skrive dem inn i `na_values`. Du kan også oppgi en liste av verdier ved å bruke firkantklammer: ‘na_values=[“-”, ”*”],
- **`index_col`:** Pandas lager automatisk en telleindeks når du leser inn et Excelark. I dette tilfellet virker det fornuftig å bruke navn på land som indeks. (Tenk på indeksen som en identifikator for raden.) Du oppgir derfor at kolonne 0 (altså kolonne A i Excelarket) skal brukes som indeks.

Mer informasjon:

- Artikkel: *Pandas: How to Read and Write Files*
  - realpython.com/pandas-read-write-files
- Artikkel: *Lists and Tuples in Python*
  - realpython.com/python-lists-tuples

1.4 Hent data fra en rad

For å hente data fra en gitt rad bruker du kommandoen `.loc[]` sammen med indeksen du er interessert i. For eksempel:

Python

```
data.loc["Norge"]
```

Dette viser dataene som er i raden indeksert med *Norge*. `.loc[]` er en kraftig kommando som du vil bruke til mye.

1.5 Hent data fra en kolonne

I Pandas kan du gjøre operasjoner på alle dataene i en kolonne samtidig. For eksempel, for å gange en kolonne med 2 gjør du omtrent `kolonne * 2`. Du bruker kolonnenavn for å referere til kolonner. For eksempel:

Python

```
data.Budsjettiltak
```

For enkle navn kan du skrive `data.kolonnenavn` som her. Men det virker ikke om kolonnenavnet inneholder for eksempel bindestrek eller mellomrom, fordi Python da prøver å tolke disse som nye kommandoer. Da må du i stedet bruke en litt mer knotete notasjon, basert på `.loc[]` som du så tidligere:

Python

```
data.loc[:, "Lån og garantier"]
```

Husk at `.loc[]` brukes for å hente rader. Om du vil hente kolonner i stedet må du først oppgi at vil ha alle rader ved å skrive kolon. Deretter kan du oppgi kolonnenavnet etter komma.

Dette virker, men er litt tungvint. Utfordringen er at en overskrift som *Lån og garantier* er brukt for å være beskrivende i Excelarket. Når du jobber med kolonner i Pandas er det ofte greit å bruke kolonnenavn som fortsatt er beskrivende, men også enklere å jobbe med i Python. Du kan gjøre dette ved å bruke `.rename()` når du leser dataene:

Python

```python
data = pd.read_excel(
    "kap1.xlsx",
    sheet_name="1.2",
    header=4,
    usecols=[0, 1, 2],
    na_values="-",
    index_col=0,
).rename(
    columns={
        "Budsjettiltak": "tiltak",
        "Lån og garantier": "lån",
    }
)
```

Notasjonen med krøllparantes, `{}`, definerer en slags ordbok som sier at *Budsjettiltak* kan oversettes til *tiltak* og så videre. På engelsk kalles dette en *dictionary* og Python bruker ofte navnet `dict`.

Mer informasjon:

- Artikkel: *Dictionaries in Python*
  - realpython.com/python-dicts
- Artikkel: *How to Iterate Through a Dictionary in Python*
  - realpython.com/iterate-through-dictionary-python

Du kan nå bruke notasjonen `data.kolonnenavn` for begge kolonnene. For eksempel kan du regne ut summen av budsjettiltak og lån på denne måten:

Python

```python
data.tiltak + data.lån
```

> **Advarsel:** Det er en bakdel med notasjonen `data.kolonnenavn`. Hvis kolonnenavnet er likt med en eksisterende `DataFrame`-kommando virker ikke denne notasjonen, og du må bruke `data.loc[:, "kolonnenavn"]` i stedet. Dette kan skape litt frustrasjon når du utforsker dataene dine, men kan skape større problemer i programmer du skriver.
>
> Hvilke kommandoer som er definert kan endre seg over tid (om du for eksempel oppdaterer til en ny versjon av Pandas), slik at kode som tidligere fungerte kan slutte å virke fordi et kolonnenavn plutselig tolkes som en kommando. I lengre programmer bør du derfor bruke `.loc[]`-notasjonen.

1.6 Regn ut nye data

Du kan legge til nye kolonner med kommandoen `.assign()`. For eksempel kan du legge til en *total*-kolonne på denne måten:

Python

```python
budsjett = data.assign(total=data.tiltak + data.lån)
```

Dette legger til kolonnen `total` som summen av tiltak og lån og lagrer dette i en ny variabel som vi har kalt `budsjett`.

> **Merk:** De fleste kommandoene i Pandas endrer **ikke** på de opprinnelige dataene. For eksempel blir ikke `data` endret av `data.assign()`. I stedet returneres en ny `DataFrame` som du kan velge å tilordne til en variabel eller koble sammen med andre kommandoer.
>
> Om du vil endre den opprinnelige `data` kan du gjøre det ved å eksplisitt skrive tilbake til variabelen: `data = data.assign(...)`.

Du kan koble sammen flere kommandoer ved å skille dem med punktum. For eksempel kan du sortere etter den nye `total`-kolonnen:

Python

```python
budsjett = data.assign(total=data.tiltak + data.lån).sort_values("total")
```

Etterhvert som du kobler sammen flere kommandoer kan koden bli litt uoversiktlig. Du kan bruke linjeskift for å organisere koden din bedre. For at ikke Python skal feiltolke linjeskiftene dine kan du bare gjøre dem inne i en parantes. Du kan derfor legge en parantes på utsiden av hele kommandoen, og starte hver delkommando på en egen linje:

Python

```python
budsjett = (
    data.assign(total=data.tiltak + data.lån)
    .sort_values(by="total")
)
```

Dette tolkes på nøyaktig samme måte som tidligere, men er enklere å holde oversikt over etterhvert som kommandoen vokser.

1.7 Håndter manglende data

Du så tidligere at det manglet data for *Lån og garantier* i **Nederland**. I dataene er disse nå markert som `NaN`. Dette betyr *Not a Number* og brukes for å markere manglende data i Pandas. Om du ser i verdiene til `budsjett` vil du se at også totalen til Nederland er markert som `NaN`, siden det ikke går å regne ut en total siden verdien på lån mangler.

Som regel vil du eksplisitt behandle manglende data. Den enkleste måten å behandle dem på er å slette dem fra dataene dine. Til dette kan du bruke `.dropna()`:

Python

```python
budsjett = (
    data.assign(total=data.tiltak + data.lån)
    .sort_values(by="total")
    .dropna(how="any")
)
```

Dette vil slette **Nederland** fra `budsjett`. `how="any"` betyr at du vil slette alle rader som har `NaN` i minst en av kolonnene. Du kan også bruke `how="all"` om du bare vil slette rader hvor alle data er `NaN`. Til slutt kan du også spesifisere `subset=...` for å bare sjekke spesifiserte kolonner.

Du kan også fylle `NaN`-felter med gitte verdier ved å bruke `.fillna()`. For eksempel kan du legge inn 0 i stedet for `NaN`:

Python

```python
budsjett = (
    data.assign(total=data.tiltak + data.lån)
    .sort_values(by="total")
    .fillna(value=0)
)
```

Dette legger inn 0 i stedet for `NaN`. Om du ser nærmere på **Nederland**-raden vil du se at dette ikke hadde helt effekten vi hadde tenkt. Riktignok er `lån` lik 0, men `total` har også blitt satt til 0. Det var ikke helt meningen.

Problemet er at du regner ut `total` før du fyller inn de manglende verdiene. Dette må snus slik at du først fyller inn `NaN`er. Dette skaper en ny liten utfordring: husk at Pandas-kommandoer ikke endrer de opprinnelige dataene. Selv om du gjør `.fillna()` før `.assign()` har ikke `data` endret seg slik at utregningen av `total` fortsatt bruker de manglende verdiene.

`.assign()` løser dette ved at du kan bruke et såkalt lambda-uttrykk for å lage en midlertidig variabel med den gjeldende verdien av dataene dine:

Python

```python
budsjett = (
    data.fillna(value=0)
    .assign(total=lambda nå: nå.tiltak + nå.lån)
    .sort_values(by="total")
)
```

Under utregningen av `total` inneholder variabelen `nå` en referanse til dataene slik de ser ut "akkurat nå", etter at `NaN` har blitt erstattet av 0.

En annen måte å håndtere dette på er å dele opp kommandoene dine og skrive til en midlertidig DataFrame underveis. For eksempel:

Python

```python
data_uten_na = data.fillna(value=0)

budsjett = (
    data_uten_na.assign(total=data_uten_na.tiltak + data_uten_na.lån)
    .sort_values(by="total")
)
```

1.8 Koble til andre data

Det finnes utallige måter du kan koble sammen data på i Pandas. Her gjør vi en kobling hvor vi vil lage en ny kolonne som sier hvorvidt landet er en del av Norden eller ikke. Dette baserer vi på en liste over land i Norden:

Python

```python
norden = ["Norge", "Sverige", "Danmark", "Finland", "Island"]
```

Her skriver vi bare listen i koden, men denne kan gjerne være en mer avansert liste hentet fra for eksempel en database eller et annet Excelark.

Når listen er tilgjengelig som en variabel, kan den brukes i `.assign()` for å lage en ny kolonne:

Python

```python
budsjett = (
    data.fillna(value=0)
    .assign(
        total=lambda nå: nå.tiltak + nå.lån,
        i_norden=lambda nå: nå.index.isin(norden),
    )
    .sort_values(by="total")
)
```

`.assign()` lager nå to nye kolonner: `total` og `i_norden`. Den siste kolonnen vil inneholde verdiene `True` og `False` hvor `True` markerer at landet tilhører Norden.

1.9 Gjør enkle spørringer

Du kan plukke ut enkelte rader fra dataene dine ved hjelp av spørringer. Disse kan gjøres ved hjelp av `.query()`. Her er noen eksempler:

Python

```python
# Total er høyere enn 10
budsjett.query("total > 10")

# Lån er høyere enn tiltak
budsjett.query("lån > tiltak")

# Total er høyere enn 10 og lån er høyere enn tiltak
budsjett.query("total > 10 and lån > tiltak")

# Landet er i norden (i_norden er True)
budsjett.query("i_norden")

# Indeks er Norge
budsjett.query("index == 'Norge'")

# Indeks (landnavn) begynner på N
budsjett.query("index.str.startswith('N')")

# Landet er i norden, oppslag på index
# @ refererer til variabler som ikke er kolonnenavn
budsjett.query("index.isin(@norden)")
```

Du kan også gjøre spørringer ved hjelp av `.loc[]`-notasjonen, men den kan ofte være litt mer knotete. Du bruker da et uttrykk inne i `.loc[]` som beskriver hvilke rader du er interessert i:

Python

```python
# Total er høyere enn 10
budsjett.loc[budsjett.total > 10]

# Lån er høyere enn tiltak
budsjett.loc[budsjett.lån > budsjett.tiltak]

# Total er høyere enn 10 og lån er høyere enn tiltak
budsjett.loc[(budsjett.total > 10) & (budsjett.lån > budsjett.tiltak)]

# Landet er i norden (i_norden er True)
budsjett.loc[budsjett.i_norden]

# Indeks er Norge
budsjett.loc["Norge"]

# Indeks (landnavn) begynner på N
budsjett.loc[budsjett.index.str.startswith("N")]

# Landet er i norden, oppslag på index
budsjett.loc[budsjett.index.isin(norden)]
```

Begge metodene, `.query()` og `.loc[]`, fungerer stort sett likt. `.query()` fungerer bedre i en kjede av kommandoer fordi spørringen gjøres på feltene slik de ser ut "akkurat nå" i kjeden.

1.10 Skriv til Excel

På samme måte som Pandas kan lese fra mange forskjellige filformater, har du mange alternativer for å skrive ut dataene igjen. Du finner en oversikt over alle formater som er støttet på pandas.pydata.org/docs/user_guide/io.html.

For å skrive til Excel kan du bruke `.to_excel()`. På samme måte som for `.read_excel()` oppgir du filnavnet:

Python

```python
budsjett.to_excel("totalt.xlsx")
```

Dette lager en ny Excelfil som heter `totalt.xlsx`. Om du åpner denne i Excel vil du se dataene, inkludert de nye `total` og `i_norden`-kolonnene.

Excelarket har nå fått de "nye" kolonnenavnene **tiltak** og **lån**. For å bruke mer beskrivende navn i regnearket kan du gjøre en `.rename()` før du skriver dataene:

Python

```python
budsjett.rename(
    columns={
        "tiltak": "Budsjettiltak",
        "lån": "Lån og garantier",
        "i_norden": "Norden?",
    }
).to_excel("totalt.xlsx")
```

Du kan også kombinere dette med spørringer og andre kommandoer:

Python

```python
(
    budsjett.query("i_norden")
    .drop(columns=["i_norden"])
    .rename(
        columns={
            "tiltak": "Budsjettiltak",
            "lån": "Lån og garantier",
            "total": "Tiltak pluss lån",
        }
    )
    .to_excel("norden.xlsx", sheet_name="Budsjett")
)
```

Dette skriver ut filen `norden.xlsx` som kun inneholder data for de nordiske landene. Siden kolonnen `i_norden` alltid vil være `True` for disse landene kaster vi den ut før vi lagrer. Også `.to_excel()` has mange instillinger for å skrive dataene på forskjellige måter. Her passer vi på at vi gir arket et bedre navn.

pandas er et effektivt verktøy for å skrive data til Excel. Dersom du trenger detaljert kontroll over det resulterende Excelarket, inkludert flere muligheter for design og bruk av formler er pakken openpyxl et bedre valg. Du kan lese mer om openpyxl i denne artikkelen:

- *A Guide to Excel Spreadsheets in Python With openpyxl*
  - realpython.com/openpyxl-excel-spreadsheets-python/

1.11 Lag plott

For å avslutte eksempelet, la oss se på noen av mulighetene for å lage figurer i pandas. Pandas bruker en pakke som heter Matplotlib (matplotlib.org/gallery) i bakgrunnen for å tegne figurer. Disse er tilgjengelige gjennom kommandoen `.plot()`:

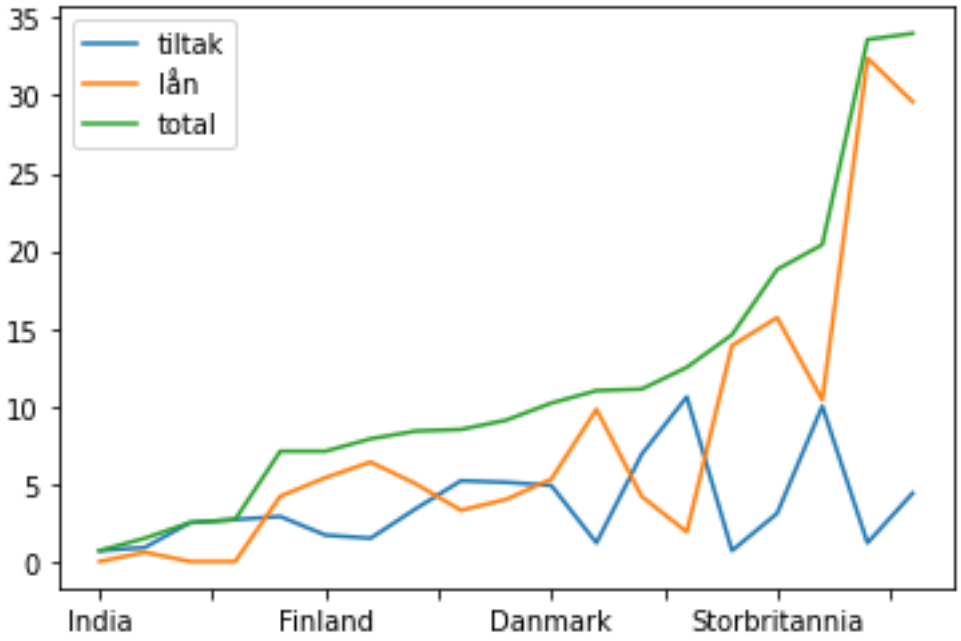


Figure 2: Figuren generert av `budsjett.plot()`

Python

```
budsjett.plot()
```

Dette viser dataene som et linjeplott som nok ikke er den beste måten å visualisere disse dataene på. Det finnes flere andre typer du kan bruke. Disse er tilgjengelige som kommandoer på `.plot`, og inkluderer `area`, `bar`, `barh`, `box`, `density`, `hexbin`, `hist`, `kde`, `line`, `pie` og `scatter`. Du kan for eksempel lage et stolpediagram med `.bar()`:

Python

```
budsjett.plot.bar()
```

Det virker mer fornuftig. Du kan også sette forskjellige innstillinger, delvis avhengig av hvilken plottetype du bruker. Du kan for eksempel bruke `stacked` for å stable stolpene oppå hverandre:

Python

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

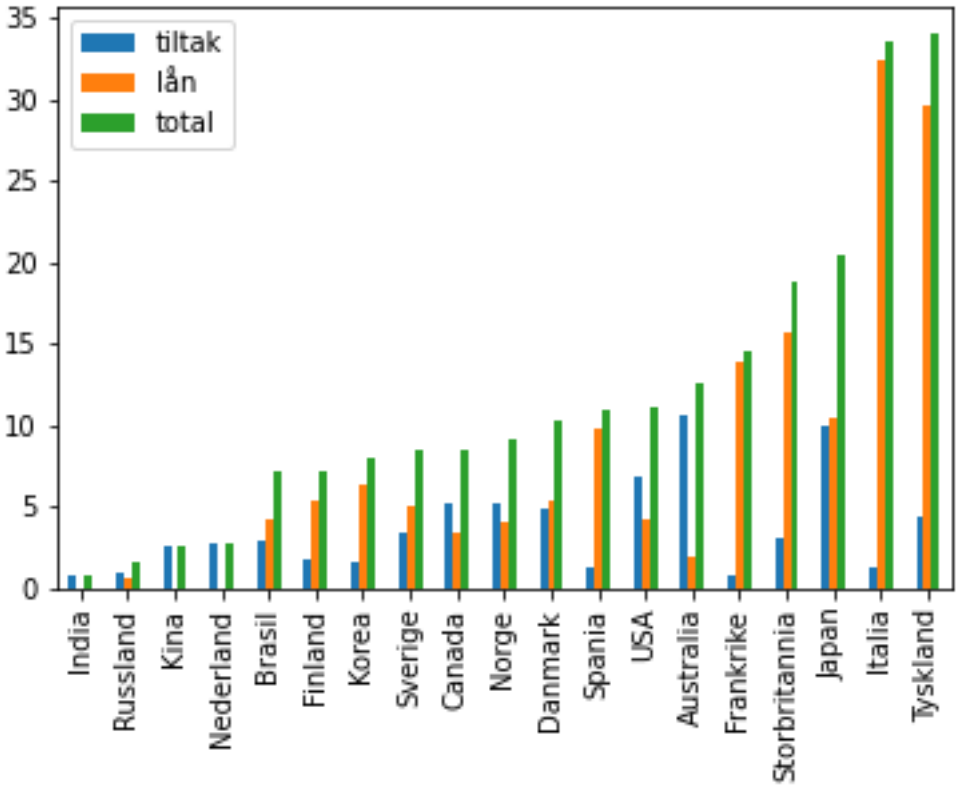


Figure 3: Figuren generert av `budsjett.plot.bar()`

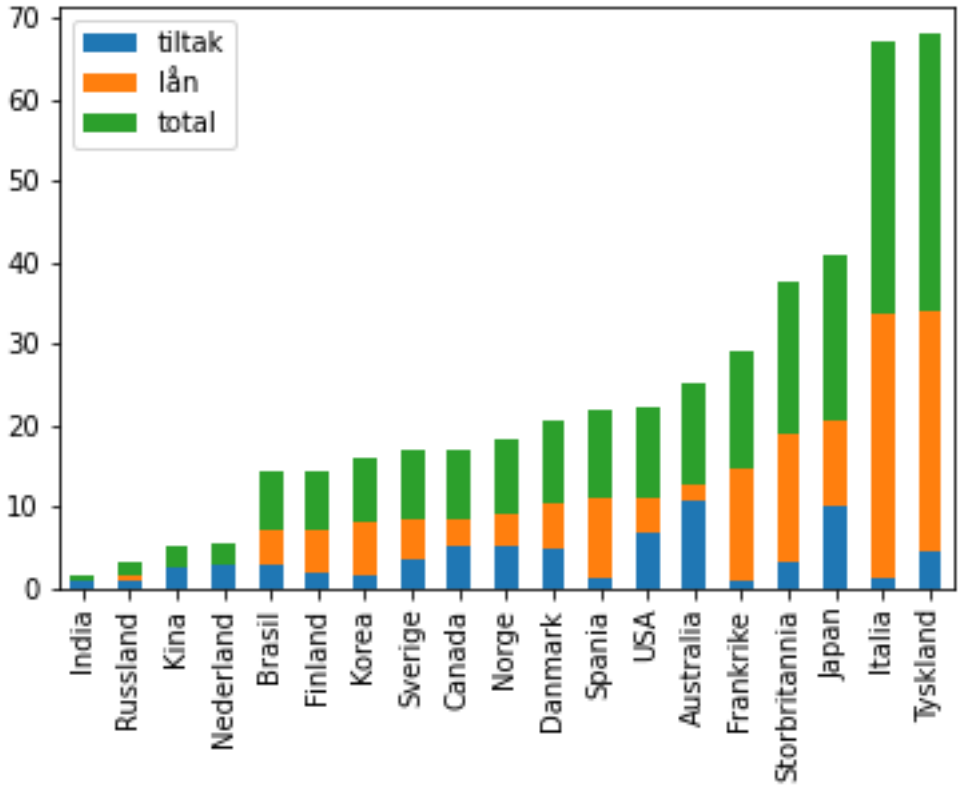


Figure 4: Figuren generert av `budsjett.plot.bar(stacked=True)`

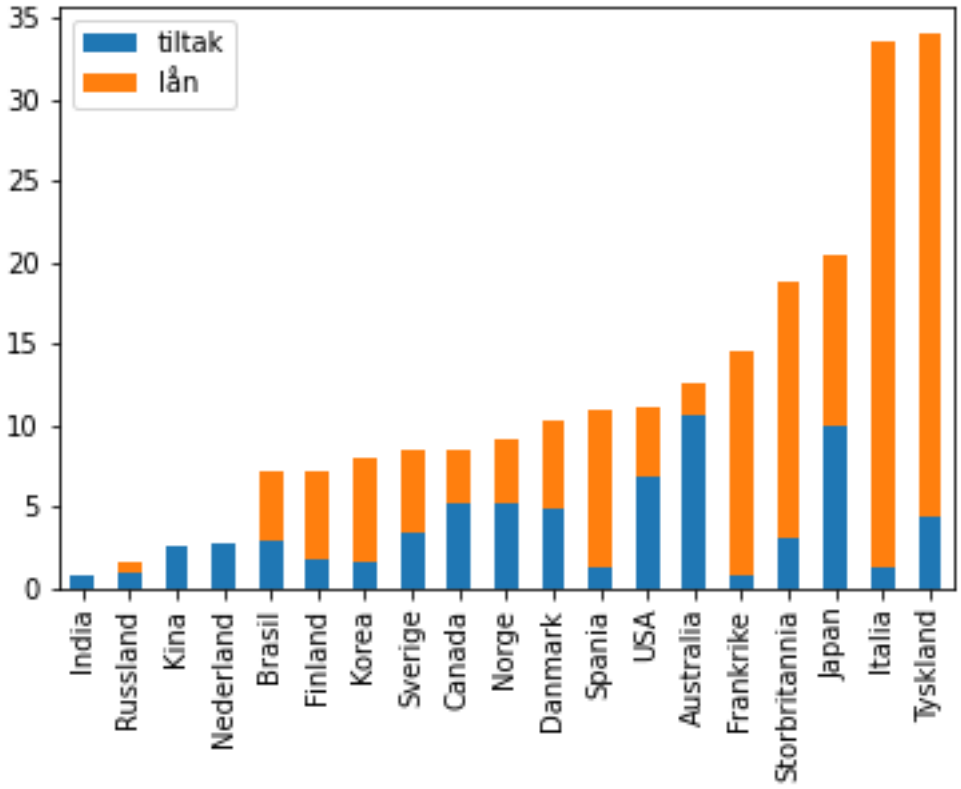


Figure 5: Figuren generert av `budsjett.loc[:, ["tiltak", "lån"]].plot.bar()`

Når vi stabler stolpene slik blir det litt feil å ta med totalkolonnen. Du kan inkludere `.plot()` som den siste i en kjede kommandoer for å plukke ut eller transformere data spesielt for en figur. For eksempel kan du bare ta med tiltak- og lån-kolonnene ved å plukke dem ut med `.loc[]`:

Python

```
1 budsjett.loc[:, ["tiltak", "lån"]].plot.bar(stacked=True)
```

Noen figurtyper krever spesielle parametre. For eksempel må du ta med `x` og `y` om du vil bruke `scatter()`. Disse forteller hvilke kolonner som skal vises langs x- og y-aksene. Du kan i tillegg ta med andre innstillinger:

Python

```
1 budsjett.plot.scatter(x="tiltak", y="lån", c="total", cmap="coolwarm")
```

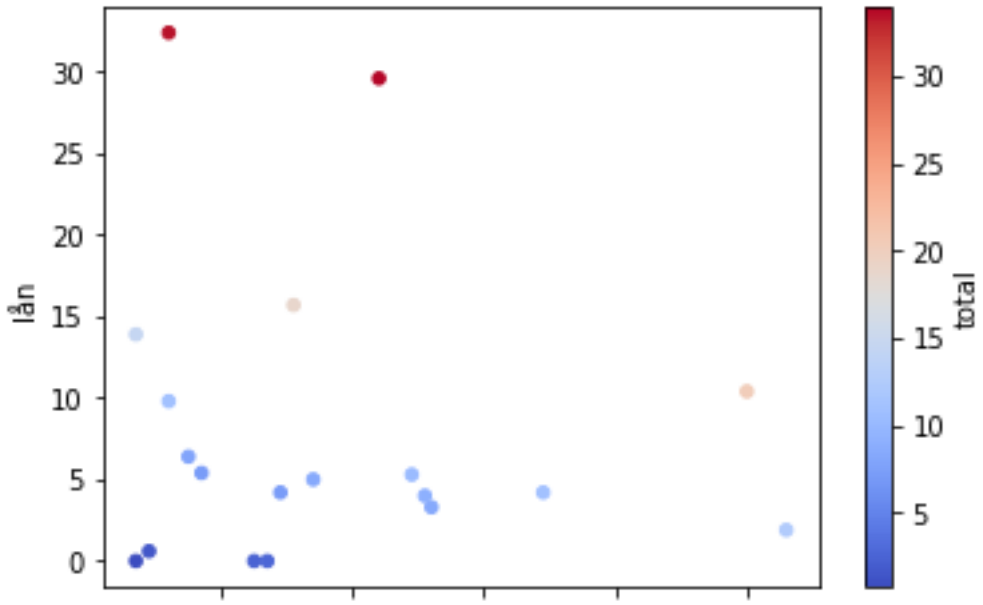


Figure 6: Figuren generert av `budsjett.plot.scatter(x="tiltak", y="lån", c="total", cmap="coolwarm")`

Her har du også brukt `c` som setter fargen og `cmap` som bestemmer hvilke farger som skal brukes.

Mer informasjon:

- Dokumentasjon for `plot`:
  - pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html
- Hvordan lage figurer i pandas:
  - pandas.pydata.org/docs/getting_started/intro_tutorials/04_plotting.html
- Matplotlib eksempelgalleri:
  - matplotlib.org/gallery/
- Oversikt over fargeløp:
  - matplotlib.org/tutorials/colors/colormaps.html
- Artikkel om plotting i pandas: *Plot With pandas: Visualization for Beginners*
  - realpython.com/pandas-plot-python/

2 Koble sammen data

> **Merk:** I kurset brukte vi data fra februar og mars 2021, men i denne seksjonen vises eksempler fra desember 2020 og januar 2021.

I del to av kurset jobbet vi med en litt større datafil, for å se eksempler på hvordan disse kan håndteres. Vi brukte åpne data på bruk av bysyklene i Oslo. Disse er tilgjengelige på oslobysykkel.no/apne-data. Spesielt lastet vi ned historiske data for desember 2020 og januar 2021 i CSV format. Filene `12.csv` og `01.csv` lagres i samme katalog som tidligere filer.

> **Merk:** Du kan bruke Excel for å se på innholdet av CSV-filer. Pass på at Excel av og til kan forandre den opprinnelige formatteringen av filen. Hvis det skjer må du muligens bruke noen andre innstillinger i `pd.read_csv()` eller eventuelt laste ned originalfilen på nytt.

Du kan lese inn CSV filer med `pd.read_csv()`. På samme måte som `pd.read_excel()` har denne flere innstillinger hvor du kan justere hvordan filen skal leses. Som et første forsøk, la oss bare lese filen og se på den:

Python

```python
data = pd.read_csv("01.csv")
data.head()
```

`.head()` er nyttig når du jobber med store datasett. Den vil bare skrive ut de første 5 radene i dataene dine. I dette tilfellet vil det se omtrent slik ut:

	started_at	ended_at	duration	start_station_id	...
0	2021-01-01 04:02:29.028	2021-01-01 04:19:33.805	1024	398	
1	2021-01-01 04:03:50.320	2021-01-01 04:11:34.677	464	742	
2	2021-01-01 04:06:39.033	2021-01-01 04:19:07.700	748	421	
3	2021-01-01 04:27:48.600	2021-01-01 04:34:55.220	426	383	
4	2021-01-01 04:37:11.115	2021-01-01 04:43:30.561	379	423	

Datasettet inneholder også flere kolonner. Du kan også bruke `.info()` for å få informasjon om hvilke kolonner det er i datasettet, hvilken type kolonnene har og hvor mye minne datasettet bruker:

Python

```python
data.info()
```

I dette tilfellet får du resultatet:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 35353 entries, 0 to 35352
Data columns (total 13 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   started_at                  35353 non-null  object
 1   ended_at                    35353 non-null  object
 2   duration                    35353 non-null  int64
 3   start_station_id            35353 non-null  int64
 4   start_station_name          35353 non-null  object
 5   start_station_description   35351 non-null  object
 6   start_station_latitude      35353 non-null  float64
 7   start_station_longitude     35353 non-null  float64
 8   end_station_id              35353 non-null  int64
 9   end_station_name            35353 non-null  object
 10  end_station_description     35351 non-null  object
 11  end_station_latitude        35353 non-null  float64
 12  end_station_longitude       35353 non-null  float64
dtypes: float64(4), int64(3), object(6)
memory usage: 3.5+ MB
```

Her kan du blant annet lese ut at datasettet:

- har 35353 rader
- har 13 kolonner med navn som listet opp
- har 3 kolonner som behandles som heltall (`int64`), 4 kolonner som behandles som flyttall (`float64`) og 6 kolonner som behandles som tekst (`object`)
- bruker rundt 3.5 MB minne

2.1 Jobb med datoer

De to første kolonnene er datoer med klokkeslett (gjerne kalt **datetime** i Python). Det vil være nyttig om pandas behandler disse som faktiske datoer og ikke som tekststrenger. Om slike datokolonner er på en rimelig standard format holder det å fortelle `.read_csv()` at kolonnene skal tolkes som datoer:

Python

```python
data = pd.read_csv("01.csv", parse_dates=["started_at", "ended_at"])
```

Om du ser på `data.info()` nå vil du blant annet se:

```
 0   started_at                35353 non-null  datetime64[ns, UTC]
 1   ended_at                  35353 non-null  datetime64[ns, UTC]
```

Kolonnene har nå fått typen `datetime64` som betyr at de tolkes som datoer med klokkeslett. Du kan da for eksempel gjøre oppslag som skjønner datoer:

Python

```python
turer = data.set_index("started_at").loc["2021-01-28"].reset_index()
```

Her lager vi et mindre datasett som indekseres på starttidspunkt, og bare inneholder turene som ble gjort 28. januar 2021.

> **Merk:** `.loc[]` gjør oppslag på indeksen. Vi setter derfor indeksen lik starttidspunkt for å gjøre oppslaget. Senere kan det være nyttig å behandle starttidspunktet som dataverdier. Vi setter derfor indeksen tilbake til en standard "telleindeks" til slutt.

Du kan nå for eksempel regne ut hvor lenge hver tur varer:

Python

```python
turer.loc[:, "ended_at"] - turer.loc[:, "started_at"]
```

```
0    00:01:24.543000
1    00:08:29.121000
2    00:20:46.556000
3    00:01:57.423000
dtype: timedelta64[ns]
```

Fordi pandas vet at `started_at` og `ended_at` er dato/klokkeslett klarer den å regne ut forskjellen som en *timedelta* som er en struktur som kan tolkes og konverteres til sekunder, minutter og timer. Du kan for eksempel legge til en kolonne med turlengde i hele minutter på denne måten:

Python

```python
from datetime import timedelta

turer = turer.assign(
    duration_minutes=(turer.ended_at - turer.started_at)
        // timedelta(minutes=1)
)
```

Vanligvis bruker du `/` i Python for å dele to tall på hverandre. Dersom du trenger å gjøre *heltallsdivisjon* kan du bruke `//` i stedet. For eksempel er `7 / 4` lik 1.75 mens `7 // 4` gir svaret 1.

Dette datasettet inneholder allerede informasjon om turlengde i kolonnen `duration`. Vi bruker den videre.

Python

```python
turer.plot(x="started_at", y="duration", style=".", alpha=0.2)
```

2.2 Grupper data

Gruppering av data er ofte nyttig i store datasett. For eksempel kan vi sammenligne turer basert på hvilke stasjoner de går mellom. Du kan for eksempel gruppere på start-stasjon og telle hvor mange turer det er fra hver:

Python

```python
turer.groupby("start_station_name").size().sort_values(ascending=False)
```

Dette vil vise at 28. januar var de mest populære startstasjonene Marcus Thranes gate, Ringnes Park og Tjuvholmen.

Du kan også gjøre operasjoner på hver gruppe. For eksempel kan du hente ut medianen i hver gruppe for alle tallkolonnene på denne måten:

Python

```python
turer.groupby("start_station_name").median().sort_values(by="duration")
```

Du kan også gruppere på flere variabler. For å sammenligne faktiske turer kan det være nyttig å grupper på både startstasjon og endestasjon:

Python

```python
antall_turer = (
    data.groupby(["start_station_name", "end_station_name"])
    .size()
    .sort_values(ascending=False)
)
```

Dette vil vise at den mest populære bysykkelturen i januar var fra *Tjuvholmen* til *Frognerstranda*.

Etter en slik gruppering er dataene på et litt uvanlig format, hvor både `start_station_name` og `end_station_name` er indekser. Dette kan være nyttig for å slå opp spesielle turer. Men det kan også være nyttig å ha dataene i et mer tradisjonelt format. Du kan bruke `.reset_index()` for å gå tilbake til en vanlig telleindeks og med stasjonsnavnene som vanlige kolonner.

Python

```python
antall_turer = (
    data.groupby(["start_station_name", "end_station_name"])
    .size()
    .sort_values(ascending=False)
    .reset_index()
    .rename(columns={0: "num_trips"})
)
```

Du kan også pivotere dataene. Pandas har ett par funksjoner som kan brukes til pivotering, men `.pivot_table()` er den mest fleksible.

Du kan for eksempel lage en matrise over start- og endestasjoner som viser antall turer mellom dem:

Python

```python
antall_turer.pivot_table(
    index="start_station_name",
    columns="end_station_name",
    values="num_trips",
    fill_value=0
)
```

Dette gir en tabell som denne (men for alle 250 stasjoner):

start_station_name	7 Juni Plassen	AHO	Adamstuen	Aker Brygge	Akersgata
7 Juni Plassen	5	0	0	0	0
AHO	1	23	0	2	3
Adamstuen	1	0	7	0	2
Aker Brygge	0	1	0	3	0
Akersgata	1	6	2	2	5

Du kan også kjøre tilsvarende spørringer som du har sett tidligere på disse aggregerte dataene:

Python

```python
antall_turer.query("num_trips > 50")
```

2.3 Slå sammen datasett

Pandas har flere metoder for å slå sammen flere datasett, inkludert `.merge()`, `.join()`, og `.concat()`.

Husk at du lastet ned både desember 2020 (`12.csv`) og januar 2021 (`01.csv`). La oss først se hvordan du kan slå sammen disse to filene og behandle dem som ett datasett. `pd.concat()` brukes for slå sammen (*konkatenere*) to eller flere DataFrames.

Python

```python
data_des = pd.read_csv("12.csv", parse_dates=["started_at", "ended_at"])
data_jan = pd.read_csv("01.csv", parse_dates=["started_at", "ended_at"])

data = pd.concat([data_des, data_jan]).reset_index(drop=True)
```

Her har du først lest inn de to filene på samme måte som tidligere. Kommandoen `pd.concat()` tar inn en liste av DataFrames og slår dem sammen. Både desember og januar-dataene har hver sin telleindeks. Disse beholdes vanligvis når du slår sammen datasett. Med `.reset_index()` sier du at du vil lage en ny telleindeks som løper fra den første til den siste raden i det sammensatte datasettet.

Du kan bruke for eksempel `data.info()` til å bekrefte at du har fått ett nytt datasett som er større enn de enkelte datasettene:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 72970 entries, 0 to 72969
Data columns (total 13 columns):
...
```

Om du skal slå sammen mange datasett kan denne fremgangsmåten raskt bli tungvinn. Det vil da være nyttig å lage en løkke som kan lese inn flere datasett:

Python

```
1 filer = ["12.csv", "01.csv"]
2
3 data_ark = []
4 for filnavn in filer:
5     ark = pd.read_csv(filnavn, parse_dates=["started_at", "ended_at"])
6     data_ark.append(ark)
7
8 data = pd.concat(data_ark).reset_index(drop=True)
```

Sluttresultatet av denne koden er det samme som den forrige. Umiddelbart kan dette virke mer komplisert. Fordelen med denne fremgangsmåten er at den skalerer fint selv om du ønsker å slå sammen hundrevis av filer. Så hva er det som skjer i koden?

- **Linje 1:** Du beskriver navnene på filene du ønsker å lese inn i en liste. I stedet for å skrive ut filnavn eksplisitt kan disse ofte genereres automatisk. For eksempel kan du bruke `filer = pathlib.Path.cwd().glob("*.csv")` for å lese inn alle CSV-filer i den aktive katalogen (`.cwd()` peker på *current working directory*).
- **Linje 3:** `data_ark` starter som en tom liste. Denne vil etterhvert fylles opp med en DataFrame for hver fil som leses.
- **Linje 4:** En `for`-løkke brukes for å gjenta kode for hvert element i en liste (eller lignende datastrukturer). Inne i løkka vil variabelen `filnavn` til enhver tid peke på det gjeldende filnavnet. Altså vil den være `"12.csv"` første runde og `"01.csv"` i andre runde.

  Hvilke kommandoer som er en del av en løkke bestemmes ved hjelp av innrykk. Legg merke til at de to neste kommandoene (linje 5 og 6) er rykket inn med 4 mellomrom. Den siste kommandoen (linje 8) er ikke rykket inn. Det betyr at linje 5 og 6 gjentas for hvert filnavn. Linje 8 kjøres bare en gang etter at løkka er ferdig.
- **Linje 5:** Les inn en datafil på samme måte som tidligere. Filnavnet er bestemt av navnene som er definert i listen på linje 1.
- **Linje 6:** Legg til DataFrame'n som nettopp ble lest inn i listen `data_ark`. Tilslutt vil denne listen inneholde alle datasettene vi ønsker å lese inn.

- **Linje 8:** Slå sammen begge (alle) datasettene i en DataFrame. På samme måte som tidligere lager vi en ny telleindeks over de sammenslåtte dataene.

`pd.concat()` er veldig nyttig om du har flere datasett med de samme kolonnene. En annen vanlig måte å slå sammen data på er at du har datasett med forskjellig informasjon (kolonner) om de samme datapunktene (radene). Da er det mer naturlig å bruke `.merge()`.

La oss si at du vil se på sammenhengen mellom antall turer og hvor lenge en sykkeltur varer. Du har allerede sett hvordan du kan lage en oversikt over antall turer:

Python

```python
antall_turer = (
    data.groupby(["start_station_name", "end_station_name"])
    .size()
    .sort_values(ascending=False)
    .reset_index()
    .rename(columns={0: "num_trips"})
)
```

For å regne ut den vanlige lengden av en tur kan du gjøre en tilsvarende gruppering:

Python

```python
lengde = (
    data.groupby(["start_station_name", "end_station_name"])
    .median()
    .loc[:, "duration"]
    .reset_index()
)
```

Nå har både `antall_turer` og `lengde` de samme kolonnene for start- og endestasjon. Det vil da være ganske greit å slå dem sammen ved å slå opp på disse:

Python

```python
antall_turer.merge(lengde, on=["start_station_name", "end_station_name"])
```

De første radene i dette nye datasettet ser slik ut:

	start_station_name	end_station_name	num_trips	duration
0	Tjuvholmen	Frognerstranda	141	284.0
1	Frognerstranda	Tjuvholmen	131	309.0
2	Kværnerbyen	Sjøsiden ved trappen	87	520.0

Om du vil slå sammen datasett som ikke ligner hverandre like mye, er det fortsatt mulig. `.merge()` og de andre metodene har flere muligheter for å velge hvilke kolonner (eller indekser) som skal brukes, hva som skal skje med manglende verdier og så videre.

Mer informasjon

- Artikkel: *Combining Data in Pandas With merge(), .join(), and concat()*
  - realpython.com/pandas-merge-join-and-concat/

Fra tallene ser du at de mest populære stasjonene brukes mye til småturer på rundt 5 minutter. Hva er de mest populære turene som varer litt lengre (mer enn 10 minutter)?

Python

```python
(
    antall_turer.merge(
        lengde,
        on=["start_station_name", "end_station_name"]
    )
    .query("duration > 600")
    .sort_values(by="num_trips", ascending=False)
)
```

	start_station_name	end_station_name	num_trips	duration
6	Sjøsiden ved trappen	Kværnerbyen	63	720
10	Huitfeldts gate	Huitfeldts gate	60	920
31	Tjuvholmen	Skråninga	44	622.5

Det ser ut til sightseeing rundt omkring Huitfeldts gate er ganske populært!

3 Legg data på kart

I den siste delen av kurset så vi raskt på hvordan vi kan visualisere data på kart. Til dette bruker vi et bibliotek som heter Folium. Dette er bygd på toppen av Javascript-biblioteket LeafletJS.

3.1 Installasjon

Folium er ikke en del av standardinstallasjonen av Anaconda. Du må derfor installere det manuelt. Du kan gjøre dette på to forskjellige måter:

I **Anaconda Navigator** kan du gjøre følgende:

- Klikk **Environments** i listen til venstre
- Klikk **Channels**
- Bruk **Add...** for å legge til kanalen `conda-forge`
- Velg **All** i nedtrekksmenyen og søk etter **folium**
- Marker **folium**
- Klikk **Apply**

(Ana)conda kan ha problemer med brannmur og lignende. Om installasjonen ser ut til å stoppe helt kan det være et problem med at Anaconda ikke kommer seg på nett. Det kan hjelpe å koble seg til et mer åpent nett hvis mulig.

Alternativt kan du installere Folium gjennom et kommandovindu:

- Åpne **Anaconda Prompt** fra startmenyen
- Skriv `conda install folium -c conda-forge`
- Om `conda`-kommandoen gir feilmelding, kan du prøve `python -m pip install folium` i stedet.

3.2 Ditt første kart

Folium lager kart for nettsider. For å se på kartet ditt lagrer du det som en HTML-fil, som du deretter kan åpne med nettleseren din.

La oss først sjekke at vi kan lage kart og åpne dem. Skriv denne koden:

Python

```python
import folium

kart = folium.Map()
kart.save("bysykkel.html")
```

Når du kjører denne skal det lagres en fil som heter `bysykkel.html`. I et utforskervindu kan du dobbelklikke på denne for å åpne den i nettleseren din.

Du burde nå se et verdenskart, som du kan manøvrere rundt i på samme måte som for eksempel Google Maps.

For å gjøre kartet litt mer brukervennlig kan du velge hvor kartet skal sentreres og hvor langt det skal være zoomet inn:

Python

```python
kart = folium.Map(location=[59.9, 10.75], zoom_start=11)
kart.save("bysykkel.html")
```

Du kan nå laste kartet på nytt i nettleseren (med F5), og det bør allerede være sentrert rundt Oslo.

3.3 Data med lengde- og breddegrader

For å legge data på kart trenger du kartkoordinater. I bysykkeldataene ligger det lengde- og breddegrader for hver stasjon, som du kan bruke. Du kan lage en liten oversikt over stasjonene ved å plukke ut de relevante kolonnene og kaste duplikater:

Python

```python
stasjoner = (
    data.loc[:, [
            "start_station_id",
            "start_station_name",
            "start_station_latitude",
            "start_station_longitude",
        ]]
    .drop_duplicates()
    .rename(
        columns={
            "start_station_id": "id",
            "start_station_name": "name",
            "start_station_latitude": "lat",
            "start_station_longitude": "lon",
        }
    )
    .set_index("id")
)
```

id	name	lat	lon
423	Schous plass	59.9203	10.7608
412	Jakob kirke	59.9179	10.7549
407	Sagene bussholdeplass	59.9377	10.7516

Folium støtter forskjellige typer markører. Her skal vi bruke en `CircleMarker` for å tegne sirkler ved hver stasjon.

Prøv først å legge til en markør manuelt:

Python

```
kart = folium.Map(location=[59.9, 10.75], zoom_start=11)
folium.CircleMarker([59.9203, 10.7608], popup="Schous plass").add_to(kart)
kart.save("bysykkel.html")
```

Du vil nå se en sirkel tegnet på kartet litt nordøst for Oslo sentrum:

Du kan klikke på sirkelen for å se informasjonen du skrev inn under `popup`. Du kan også endre størrelsen på sirkelen eller lage litt bakgrunnsfyll (som også gjør det enklere å klikke på markøren):

Python

```
kart = folium.Map(location=[59.9, 10.75], zoom_start=11)
folium.CircleMarker(
    [59.9203, 10.7608],
    popup="Schous plass",
    radius=50,
    fill=True
).add_to(kart)
kart.save("bysykkel.html")
```

For å legge til alle stasjonene kan du bruke en `for`-løkke som gjør kommandoer for alle elementer i en liste:

Python

```python
kart = folium.Map(location=[59.9, 10.75], zoom_start=11)

for id in stasjoner.index:
    stasjon = stasjoner.loc[id]
    posisjon = [stasjon.loc["lat"], stasjon.loc["lon"]]
    folium.CircleMarker(
        posisjon,
        popup=stasjon.loc["name"],
        radius=20,
        fill=True
    ).add_to(kart)

kart.save("bysykkel.html")
```

Til slutt kan vi gjøre samme gruppering som tidligere for å legge til informasjon om hvilke stasjoner som er mest populære.

> **Merk:** I disse eksemplene bruker vi `station_id` for å identifisere stasjoner fordi det faktisk er to forskjellige stasjoner med samme navn.

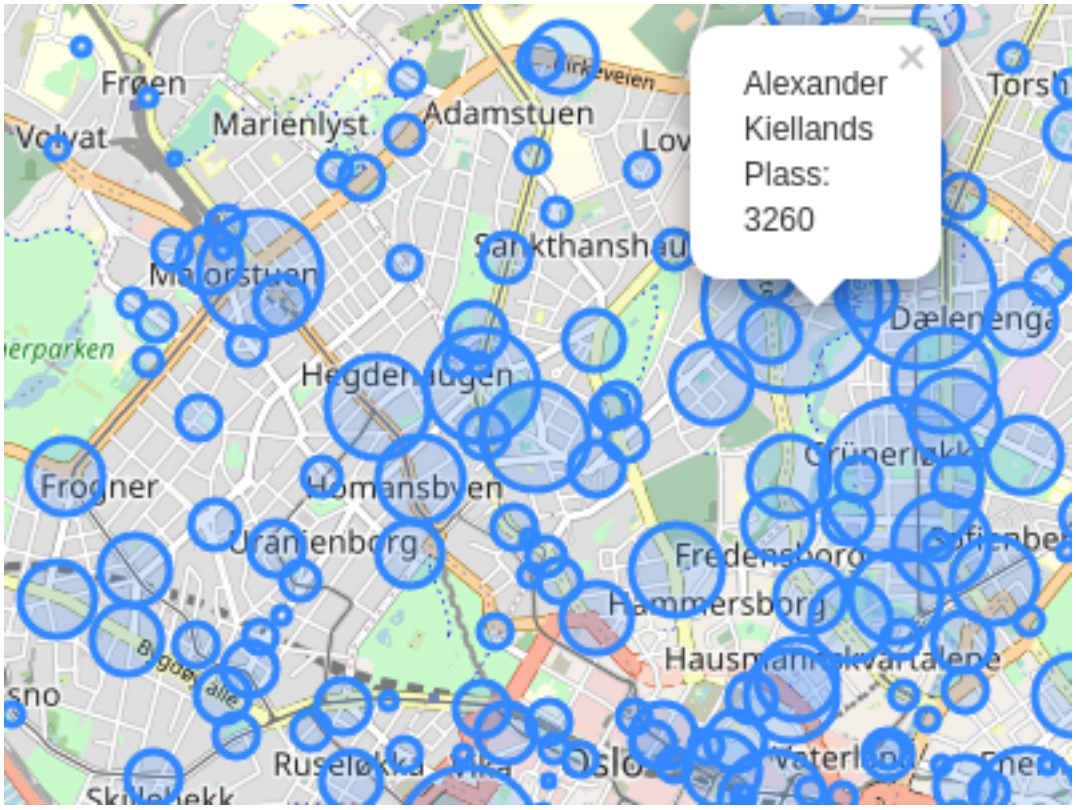
Python

```python
kart = folium.Map(location=[59.9, 10.75], zoom_start=11)
antall_turer = data.groupby("start_station_id").size()

for id in stasjoner.index:
    stasjon = stasjoner.loc[id]
    posisjon = [stasjon.loc["lat"], stasjon.loc["lon"]]
    folium.CircleMarker(
        posisjon,
        popup=f"{stasjoner.loc[id, 'name']}: {antall_turer.loc[id]}",
        radius=antall_turer.loc[id] / 100,
        fill=True
    ).add_to(kart)

kart.save("bysykkel.html")
```

Dette gir deg et kart over alle stasjonene. Størrelsen på sirklene er skalert i forhold til hvor mange turer som har startet ved den stasjonen:



Mer informasjon:

- Folium:
  - python-visualization.github.io/folium/quickstart.html
- Leaflet:
  - leafletjs.com/examples/quick-start/