

PYTHON ØVET

FOR IDA D. 02/10/2022

Omkring mig

- Anders Bensen Ottsen
- Diplomingeniør i Softwareteknologi
 - Fra Syddansk Universitet
- Læser cand.polyt i Computer Science & Engineering
 - Ved Danmarks Tekniske Universitet
 - Skriver pt. speciale i AI og ansigtsgenkendelse
- Arbejder som Machine Learning Engineer ved Weel & Sandvig
- Har undervist python kurser for IDA 8+ gange efterhånden

Omkring Jens

- Jens Kristian Vitus Bering
- BSc. I Software Engineering
 - Fra Syddansk Universitet
- Læser cand.polyt i Software Engineering
 - Ved Syddansk Universitet
- Arbejder som udvikler ved Bankdata
 - Hvor han programmerer finans software

Omkring Loc

- Loc Hoang Thanh Nguyen
- Diplomingeniør i Softwareteknologi
 - Fra Syddansk Universitet
- Læser cand.polyt i Software Engineering
 - Ved Syddansk Universitet
- Arbejder som DevOps ved Weel & Sandvig
 - Hvor han laver infrastruktur & udvikler software

Dagens program

- Funktionel programmering
- SQLite databaser
- REST API
 - Requests
 - Opsætning
- Eksempel på en service
- Hvad man kan gøre herfra og spørgerunde

Dagens program

- Funktionel programmering
 - SQLite databaser
 - REST API
 - Requests
 - Opsætning
 - Eksempel på en service
 - Hvad man kan gøre herfra og spørgerunde
- Et anderledes paradigme

Dagens program

- Funktionel programmering — Et anderledes paradigme
- SQLite databaser — Datalagring & kommunikation
- REST API
 - Requests
 - Opsætning
- Eksempel på en service
- Hvad man kan gøre herfra og spørgerunde

Dagens program

- Funktionel programmering

Et anderledes paradigme

- SQLite databaser

Datalagring & kommunikation

- REST API

Netværkskald (HTTP)

- Requests
- Opsætning

- Eksempel på en service
- Hvad man kan gøre herfra og spørgerunde

Dagens program

- Funktionel programmering

Et anderledes paradigme

- SQLite databaser

Datalagring & kommunikation

- REST API

Netværkskald (HTTP)

- Requests

- Opsætning

- Eksempel på en service

Hvordan man binder det hele sammen

- Hvad man kan gøre herfra og spørgerunde

Dagens program

- Funktionel programmering

Et anderledes paradigme

- SQLite databaser

Datalagring & kommunikation

- REST API

Netværkskald (HTTP)

- Requests

- Opsætning

- Eksempel på en service

Hvordan man binder det hele sammen

- Hvad man kan gøre herfra og spørgerunde

Afrunding

Efter i dag kan I:

- Forstå anderledes paradigmer end imperativt og objektorienteret
- Forstå hvordan computere kan kommunikere igennem Python
 - Og internettets rygrad
- Gemme data & kommunikere med SQL databaser i Python
- Have værktøjer nok til at bygge moderne programmer
- Så hvordan når vi der til?
 - Ved små ”forelæsninger”
 - Med lidt liveprogrammering fra undertegnet
 - Opgaveløsning

Efter i dag kan I:

- Forstå anderledes paradigmer end imperativt og objektorienteret
- Forstå hvordan computere kan kommunikere igennem Python
 - Og internettets rygrad
- Gemme data & kommunikere med SQL databaser i Python
- Have værktøjer nok til at bygge moderne programmer
- Så hvordan når vi der til?
 - Ved små ”forelæsninger”
 - Med lidt liveprogrammering fra undertegnet
 - Opgaveløsning

Meget af tiden går her

Det online format

- Forelæsningerne bliver mig der taler i zoom
 - Spørgsmål? Skriv i chatten, vi skal nok holde øje med den
- I dag er der mere teori end ved Python 1 & 2 (Grundet sværere emner)
 - Jeg starter med at forklare teorien, og viser den så anvendt i Python
- Ved opgaverne bliver I smidt ud i breakout rooms
 - Hvor I har ca. 30 minutter til at løse dem
 - Jens, Loc og jeg hopper rundt og hjælper i breakout rooms
 - ”Ræk hånden op” hvis i sidder fast! Så kommer vi (prøver i hvert fald)
 - I er selvfølgelig ikke tvunget til at tale med de andre, men det plejer at hjælpe!
- **[LINK TIL SLIDES HER](#)**

Visual Studio Code

- IDE'en vi bruger i dag
- Minimalistisk
- Er generelt virkelig god!
- I kan også bruge andre, men vi har mere svært ved at supportere

FUNKTIONEL PROGRAMMIERUNG

Men først, python liste tricks

- Python er et ret unikt sprog når det kommer til liste operationer
 - Pga. "list slicing"
 - Vi "slicer", eller deler, listen op
 - Det er ret praktisk når man arbejder med data (eller AI)
- Så lad os kigge på det

List Slicing

```
1  my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2
3  # Get the last element
4  my_list[-1] # 10
5
6  # Get the list from the 1st index
7  my_list[1:] # [2, 3, 4, 5, 6, 7, 8, 9, 10]
8
9  # Get everything but the last element
10 my_list[:-1] # [1, 2, 3, 4, 5, 6, 7, 8, 9]
11
12 # Get elements from the 4th index to the 6th
13 my_list[4:6] # [5, 6]
14
15 # Get from the 4th index to before the last 4 elements
16 my_list[4:-4] # [5, 6]
17
18 # Get every 2nd element from index 0 to the last element
19 my_list[0:-1:2] # [1, 3, 5, 7, 9]
20
21 # Same as above
22 my_list[::2] # [1, 3, 5, 7, 9]
```

List Slicing

```
1 my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2
3 # Get the last element
4 my_list[-1] # 10
5
6 # Get the list from the 1st index
7 my_list[1:] # [2, 3, 4, 5, 6, 7, 8, 9, 10]
8
9 # Get everything but the last element
10 my_list[:-1] # [1, 2, 3, 4, 5, 6, 7, 8, 9]
11
12 # Get elements from the 4th index to the 6th
13 my_list[4:6] # [5, 6]
14
15 # Get from the 4th index to before the last 4 elements
16 my_list[4:-4] # [5, 6]
17
18 # Get every 2nd element from index 0 to the last element
19 my_list[0:-1:2] # [1, 3, 5, 7, 9]
20
21 # Same as above
22 my_list[::2] # [1, 3, 5, 7, 9]
```

- Vi kan se på lister som slices [*a*:*b*:*c*]
 - Hvor vi går fra et index *a* til et index *b*, og bevæger os i størrelsen *c*

Programmerings paradigmer

- Definition
 - En metode, eller en måde at tænke på, til at løse et givet problem
 - Forskellige paradigmer kan løse det samme problem på forskellige måder
 - Nogle sprog har ”kun” et paradigme, imens mange mere sprog moderne har flere
 - Som f.eks. python
- Imperativt
 - Koden kører i faste sekvenser (skridt for skridt nedad)
- Objektorienteret
 - Vi modellerer koden i klasser og objekter som bruger hinanden
- Funktionelt
 - Vi opdeler vores kode i funktioner der passer til problemet


Funktioner & lambda

- Case: En metode til at tage et tal i anden

Funktioner & lambda

- Case: En metode til at tage et tal i anden
- Som vi kender den

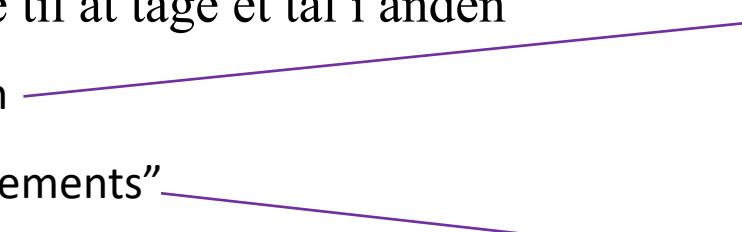
```
1  # The regular way
2  def square(x):
3      return x**2
4  square(2) # 4
```



Funktioner & lambda

- Case: En metode til at tage et tal i anden
- Som vi kender den
- Med "lambda statements"

```
1  # The regular way
2  def square(x):
3      |   return x**2
4  square(2) # 4
5
6  # Using lambda statements
7  square = lambda x : x ** 2
8  square(2) # 4
```



Funktioner & lambda

- Case: En metode til at tage et tal i anden
- Som vi kender den
- Med "lambda statements"
 - Det er funktioner der ikke behøver at have et navn
 - Her til højre har den, men vi får se
 - Kan skrives på en linje, og bliver ofte bare givet som et argument

```
1  # The regular way
2  def square(x):
3      |   return x**2
4  square(2) # 4
5
6  # Using lambda statements
7  square = lambda x : x ** 2
8  square(2) # 4
```

Map

- En specifik metode i python
 - Som man ofte bruger i funktionel programmering
 - Tager to argumenter: *map(function, list)*
- Bruges som regel til at ”mappe” sin data over i et andet format
 - Som f.eks. at tage alle tal i en liste i anden

Map

```
1 my_list = [1,2,3,4]
2
3 def square(x):
4     return x**2
5
6 # Regular way
7 squared_list = [] # [1, 4, 9, 16]
8 for element in my_list:
9     squared_list.append(square(element))
10
11 # Shorter map way
12 list(map(square, my_list)) # [1, 4, 9, 16]
```

- Så map løber igennem hvert element i en liste
 - Og anvender den givne funktion på hvert element
- *map* returnerer et objekt, men her putter man bare list()
rundt om

Map

```
1  my_list = [1,2,3,4]
2
3  def square(x):
4      |   return x**2
5
6  # Regular way
7  squared_list = [] # [1, 4, 9, 16]
8  for element in my_list:
9      |   squared_list.append(square(element))
10
11 # Shorter map way
12 list(map(square, my_list)) # [1, 4, 9, 16]
13
14 # Map with lambda
15 square = lambda x : x**2
16 list(map(square, my_list)) # [1, 4, 9, 16]
17
18 # Map with lambda, one liner
19 list(map(lambda x : x**2, my_list)) # [1, 4, 9, 16]
20
21 ##### DIFFERENT EXAMPLE #####
22
23 # Making all strings upper case - lambda and map
24 list(map(lambda s : s.upper(), ["anders", "jens", "loc"])) # ['ANDERS', 'JENS', 'LOC']
```

- Så map løber igennem hvert element i en liste
 - Og anvender den givne funktion på hvert element
- *map* returnerer et objekt, men her putter man bare list() rundt om
- Så her kan man se hvorfor lambda kan være smart
 - I funktionel programmering giver man ofte bare funktioner med som argument

Filter

- Endnu en ”funktionel programmering” metode i python
- Bruges til at ”filtrere” data væk
 - I stedet for at ”mappe” det
- Tager også en funktion og liste som argument:
 - *filter(function, list)*

Filter

```
1 my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2
3 # Regular way
4 filtered_list = [] # [1, 2, 3]
5 for element in my_list:
6     if (element < 4):
7         filtered_list.append(element)
8
9 # Filter and lambda
10 list(filter(lambda x : x < 4, my_list)) # [1, 2, 3]
```

- Så filter løber igennem hvert element i en liste
 - Og gemmer et element på en betingelse
 - Altså hvis noget er sandt (true)!

Filter

```
1  my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2
3  # Regular way
4  filtered_list = [] # [1, 2, 3]
5  for element in my_list:
6      if (element < 4):
7          filtered_list.append(element)
8
9  # Filter and lambda
10 list(filter(lambda x : x < 4, my_list)) # [1, 2, 3]
11
12
13 ##### DIFFERENT EXAMPLE #####
14
15 # If element is smaller than 4 OR larger than 7 then save it
16 list(filter(lambda x : (x < 4 or x > 7), my_list)) # [1, 2, 3, 8, 9, 10]
17
18 # If string contains character "a" then save it
19 list(filter(lambda s : "a" in s, ["anders", "jens", "loc"])) # ["anders"]
```

- Så filter løber igennem hvert element i en liste
 - Og gemmer et element på en betingelse
 - Altså hvis noget er sandt (true)!

Opgaver pt. 1 (20-30 minutter)

- Betragt listen:** `one_hundred_numbers = list(range(100))`
Brug slicing til at få hver syvende tal mellem index 20 og 80.
 - Det skulle gerne give: `[20, 27, 34, 41, 48, 55, 62, 69, 76]`
- Betragt listen:** `celsius_degrees = [20, 16, 19, 22, 18, 30]`
Brug map metoden og en lambda funktion til at regne hver celsius grad om til fahrenheit.
 - Det skulle gerne returnere: `[68.0, 60.8, 66.2, 71.6, 64.4, 86.0]`
 - Formlen er: `celsius_to_fahrenheit = (1.8 * celsius_degrees) + 32`
- Betragt listen:** `name_list = ['Loc', 'Anders', 'Jens', 'Henriette', 'Hans Erik', 'Andrea']`
Brug filter metoden og en lambda funktion til at få fat i de navne som er mindre end 4 bogstaver langt ELLER hvis sidste bogstave er s i navnet.
 - Det skulle gerne returnere: `['Loc', 'Anders', 'Jens']`
 - Hints:
 - Man kan få længden af en streng som man kan med en liste.
 - Man kan også få det sidste element i en streng som man kan med en liste.
- (Ekstra) Betragt listen:** `noisy_strings = ["python...##.%", "?bad##.%,.", "is,...#%. ", "!not%,,,,", "powerful##,,,,,,"]`
Brug først map metoden til at fjerne alt støjen (.,%#) der er efter hver streng. Brug herefter filter metoden til at gemme en hver streng som ikke starter med "!" og ikke starter med "?".
 - Det skulle gerne give: `["python", "is", "powerful"]`
 - Hints:
 - `"hej!.".strip("!.")` returnerer `"hej"`
 - Man kan også få første element i en streng med `[0]`

SQL & DATABASES

Databaser

- En database er et slags system der gemmer større mængder data/information
 - Herefter kan man så hente/tilføje/ændre i denne data
- Så i virkeligheden er en database bare endnu et program
 - Med det ene formål at stå for den gemte data
- Når man bygger software vil der næsten altid være en database
 - En dedikeret ”server”/computer hvor vores database er, f.eks. I cloud
 - Så snakker vores system så med den her computer



RDBMS

- Relational Database Management System (RDBMS)
 - En specifik slags database baseret på den ”relationelle model”
 - Opfundet af datalogen Edgar F. Codd i ~ 1970 hos IBM
- Ideen er lidt som excel - det er bare tabeller
 - Men tabellerne kan have en relation til hinanden

RDBMS - Brewery

id	name	address	country
1	Albani	Tværgade 2, 5000 Odense	Denmark
2	Carlsberg	Vestre Ringvej 111, 7000 Fredericia	Denmark
3	Heineken	Stadhouderskade 78, 1072 AE Amsterdam	Netherlands

RDBMS - Brewery

id	name	address	country
1	Albani	Tværgade 2, 5000 Odense	Denmark
2	Carlsberg	Vestre Ringvej 111, 7000 Fredericia	Denmark
3	Heineken	Stadhouderskade 78, 1072 AE Amsterdam	Netherlands

- Ovenstående kalder man et "Table"
 - Det her table hedder "Brewery"
 - Som består af 4 kolonner: Id, Name, Address, Country

RDBMS - Brewery

id	name	address	country
1	Albani	Tværgade 2, 5000 Odense	Denmark
2	Carlsberg	Vestre Ringvej 111, 7000 Fredericia	Denmark
3	Heineken	Stadhouderskade 78, 1072 AE Amsterdam	Netherlands

- Ovenstående kalder man et "Table"
 - Det her table hedder "Brewery"
 - Som består af 4 kolonner: Id, Name, Address, Country
- Et table består af kolonner med en type (F.eks. Streng, ints, doubles)

RDBMS - Brewery

id	name	address	country
1	Albani	Tværgade 2, 5000 Odense	Denmark
2	Carlsberg	Vestre Ringvej 111, 7000 Fredericia	Denmark
3	Heineken	Stadhouderskade 78, 1072 AE Amsterdam	Netherlands

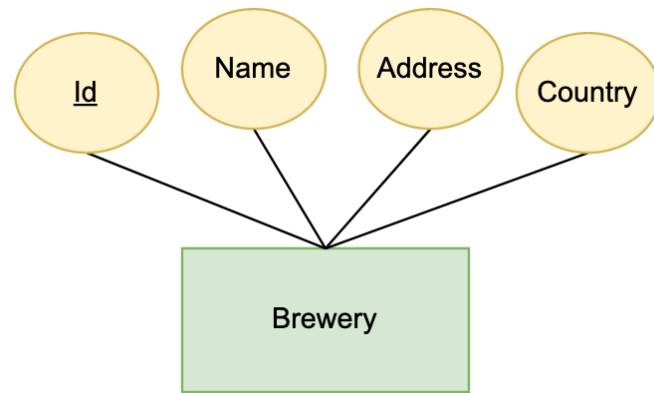
- Ovenstående kalder man et "Table"
 - Det her table hedder "Brewery"
 - Som består af 4 kolonner: Id, Name, Address, Country
- Et table består af kolonner med en type (F.eks. Streng, ints, doubles)
 - Og rækker som er den reelle data

RDBMS - Brewery

id	name	address	country
1	Albani	Tværgade 2, 5000 Odense	Denmark
2	Carlsberg	Vestre Ringvej 111, 7000 Fredericia	Denmark
3	Heineken	Stadhouderskade 78, 1072 AE Amsterdam	Netherlands

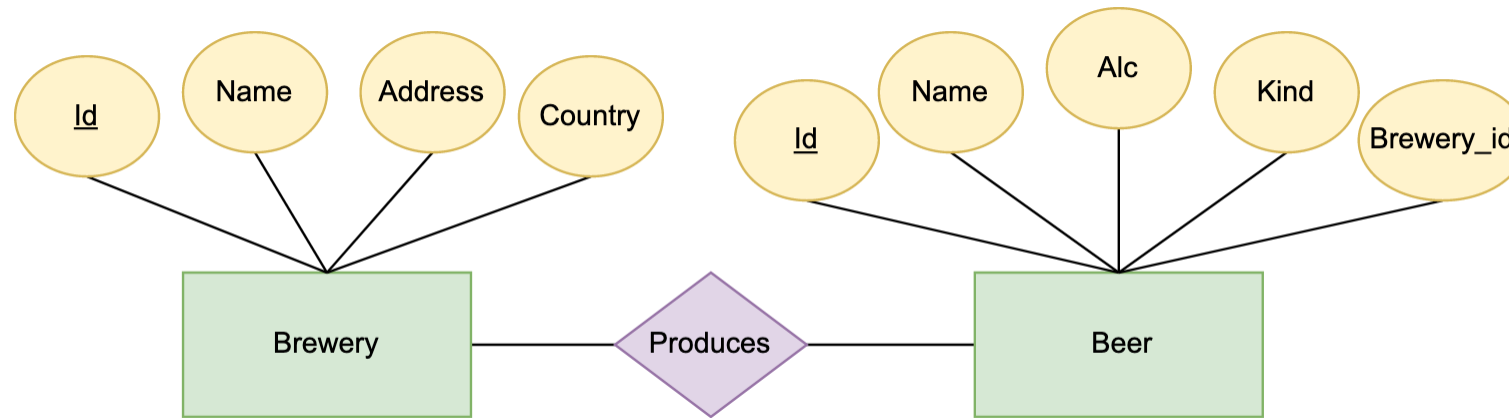
- Ovenstående kalder man et "Table"
 - Det her table hedder "Brewery"
 - Som består af 4 kolonner: Id, Name, Address, Country
- Et table består af kolonner med en type (F.eks. Streng, ints, doubles)
 - Og rækker som er den reelle data
 - Har altid noget til at kendetegne en specifik række (Primary Key)

Diagram over Brewery

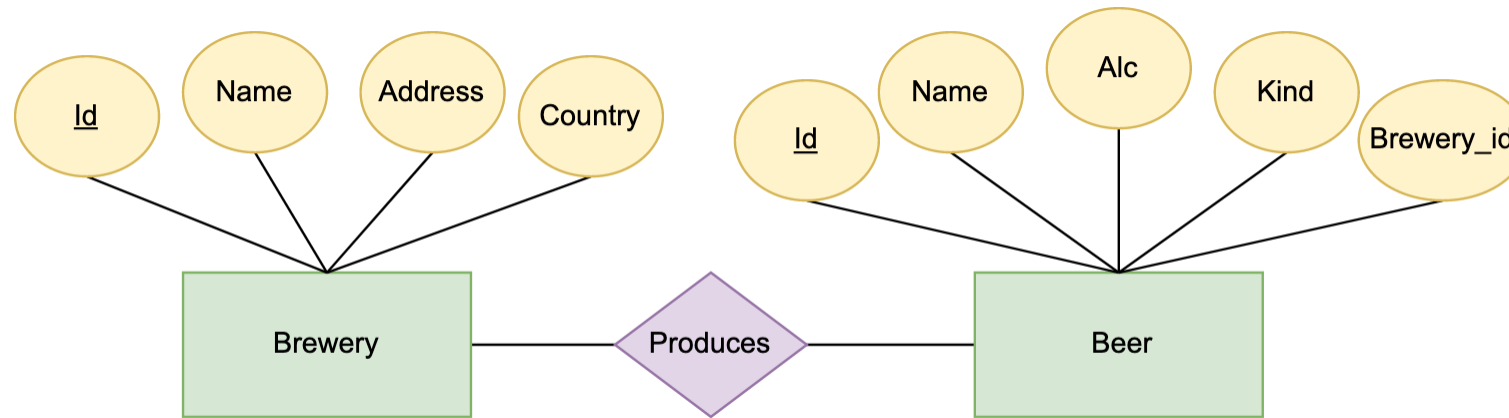


- Typen af diagram hedder et Entity-Relation (ER) diagram
 - Og bliver ofte brugt til at beskrive databaser
- Men vi har endnu ingen relationer!
 - F.eks. Producerer et bryggeri jo nogle øl
 - Den her relation kan vi lave med RBDMS

Brewery + Beer



Brewery + Beer



- Så et bryggeri brygger/producerer øl

RBDMS - Beer

id	name	alc	kind	brewery_id
1	Odense Classic	4.6	Pilsner	1 →
2	Odense Pilsner	4.6	Pilsner	1 →
3	Mosaic IPA	5.7	IPA	1 →
4	Tuborg Pilsner	4.6	Pilsner	2 →
5	Tuborg Classic	4.6	Pilsner	2 →
6	Tuborg Raa	4.5	Organic Pilsner	2 →
7	Red Tuborg	4.3	Pilsner	2 →
8	Baltika Dark Brown Ale	5	Brown Ale	2 →
9	Brewmasters Collection Irish Red Ale	4.6	Ale	2 →
10	Brooklyn Summer Ale	5	Ale	2 →
11	Heineken	4.4	Pilsner	3 →
12	Sol	4.4	Pilsner	3 →

RBDMS - Beer

id	name	alc	kind	brewery_id
1	Odense Classic	4.6	Pilsner	1 →
2	Odense Pilsner	4.6	Pilsner	1 →
3	Mosaic IPA	5.7	IPA	1 →
4	Tuborg Pilsner	4.6	Pilsner	2 →
5	Tuborg Classic	4.6	Pilsner	2 →
6	Tuborg Raa	4.5	Organic Pilsner	2 →
7	Red Tuborg	4.3	Pilsner	2 →
8	Baltika Dark Brown Ale	5	Brown Ale	2 →
9	Brewmasters Collection Irish Red Ale	4.6	Ale	2 →
10	Brooklyn Summer Ale	5	Ale	2 →
11	Heineken	4.4	Pilsner	3 →
12	Sol	4.4	Pilsner	3 →

- Det her er "relationen" til bryggeriet
- Hver øl har også id'et til sit brewery
 - Kaldes en "Foreign Key"

SQL

- Et programmeringssprog man bruger til at skabe RDBMS
 - Og kommunikere med
- Opfundet i ~ 1974 af Donald B. Chamberlin & Raymond F. Boyce, også ved IBM
- I SQL skriver man "Queries"/Forespørgsler
 - Man spørger altså databasen om noget data
 - Et meget specifikt format
- Der findes forskellige SQL sprog, f.eks. PostgreSQL, MySQL osv.
 - De er dog alle sammen ekstremt ens, kan man et - kan man alle

SQL - Datatyper

- serial / INTEGER Autoincrement
 - Det er den vi bruger til id! En int som selv gør sig større
- VARCHAR
 - String
- BIGINT
 - Den største integer, kan også være mindre f.eks. tinyint
- FLOAT
 - Double/Kommatal
- Der er mange flere... https://www.w3schools.com/sql/sql_datatypes.asp

SQL - Create Table Brewery

```
CREATE TABLE brewery (  
    id serial PRIMARY KEY,  
    name VARCHAR(256),  
    address VARCHAR(256),  
    country VARCHAR(256)  
);
```

SQL - Create Table Brewery

```
CREATE TABLE brewery (  
    id serial PRIMARY KEY,  
    name VARCHAR(256),  
    address VARCHAR(256),  
    country VARCHAR(256)  
);
```

- Så vi laver et table "brewery"
- Den har 4 attributter
 - 3 strings af længde MAX 256
 - 1 serial (som er vores "Primary Key")

SQL - Create Table Beer

```
CREATE TABLE beer (  
    id serial PRIMARY KEY,  
    name    VARCHAR(256),  
    alc     FLOAT(4),  
    kind    VARCHAR(256),  
    brewery_id BIGINT,  
    FOREIGN KEY (brewery_id) REFERENCES Brewery(id)  
);
```

- Så vi laver et table "beer"
- Den har 5 attributter
 - 2 strings af længde MAX 256
 - 1 serial (som er vores "Primary Key")
 - 1 float (med 4 decimaler) som er alkohol procent
 - 1 integer, som er referencen til Brewery

SQL – Insert Into, en række

```
INSERT INTO brewery(name, address, country)  
VALUES ('Albani', 'Tværgade 2, 5000 Odense', 'Denmark');
```

SQL – Insert Into, en række

```
INSERT INTO brewery(name, address, country)
VALUES ('Albani', 'Tværgade 2, 5000 Odense', 'Denmark');
```

- Vi indsætter data i brewery
 - I kolonnerne (name, address, country)
 - Ikke i id! Da det er serial holder den selv styr på id'et

SQL – Insert Into, flere rækker

```
INSERT INTO beer(name, alc, kind, brewery_id)
VALUES
('Odense Classic', 4.6, 'Pilsner', 1),
('Odense Pilsner', 4.6, 'Pilsner', 1),
('Mosaic IPA', 5.7, 'IPA', 1),
('Tuborg Pilsner', 4.6, 'Pilsner', 2),
('Tuborg Classic', 4.6, 'Pilsner', 2),
('Tuborg Raa', 4.5, 'Organic Pilsner', 2),
('Red Tuborg', 4.3, 'Pilsner', 2),
('Baltika Dark Brown Ale', 5, 'Brown Ale', 2),
('Brewmasters Collection Irish Red Ale', 4.6, 'Ale', 2),
('Brooklyn Summer Ale', 5, 'Ale', 2),
('Heineken', 4.4, 'Pilsner', 3),
('Sol', 4.4, 'Pilsner', 3);
```

- Her indsætter vi alle de forskellige øl i beer tabellen
 - Det vigtige her er vores reference til brewery!

SQL – SELECT *

```
SELECT *  
FROM BREWERY
```

SQL – SELECT *

SELECT *
FROM BREWERY

Det her betyder ALT data

SQL – SELECT *

SELECT * Det her betyder ALT data
FROM BREWERY Tabellen vi henter fra

SQL – SELECT *

```
SELECT *  
FROM BREWERY
```

	id	name	address	country
1	1	Albani	Tværgade 2, 5000 Odense	Denmark
2	2	Carlsberg	Vestre Ringvej 111, 7000 Fredericia	Denmark
3	3	Heineken	Stadhouderskade 78, 1072 AE Amsterdam	Netherlands

SQL – SELECT

```
SELECT name, country  
FROM BREWERY
```

	name	country
1	Albani	Denmark
2	Carlsberg	Denmark
3	Heineken	Netherlands

SQL – SELECT * FROM WHERE

```
SELECT *  
FROM brewery  
WHERE country = 'Denmark'
```

SQL – SELECT *

SELECT *

FROM brewery

WHERE country = 'Denmark'

Vi henter kun data fra der hvor vores rækker har Country = 'Denmark'

SQL – SELECT *

```
SELECT *  
FROM brewery  
WHERE country = 'Denmark'
```

	id	name	address	country
1	1	Albani	Tværgade 2, 5000 Odense	Denmark
2	2	Carlsberg	Vestre Ringvej 111, 7000 Fredericia	Denmark

SQL JOIN

- Hvis vi henter alt data fra vores øl/beer tabel
 - Så får vi bare id'erne på det pågældende bryggeri
 - Hvad nu hvis vi gerne vil have navnet på bryggeriet med ud?
 - Så bruger vi et join!

SQL JOIN

```
SELECT *  
FROM beer  
JOIN brewery on brewery.id = beer.brewery_id
```

SQL JOIN

```
SELECT *  
FROM beer  
JOIN brewery on brewery.id = beer.brewery_id
```

Så vi "joinder" vores to tabeller, sådan at der hvor brewery_id'et er lig med et brewery.id så henter vi også alt andet data

SQL JOIN

```
SELECT *  
FROM beer  
JOIN brewery on brewery.id = beer.brewery_id
```

	id	name	alc	kind	brewery_id	id	name	address	country
1	1	Odense Classic	4.6	Pilsner	1	1	Albani	Tværgade 2, 5000 Odense	Denmark
2	2	Odense Pilsner	4.6	Pilsner	1	1	Albani	Tværgade 2, 5000 Odense	Denmark
3	3	Mosaic IPA	5.7	IPA	1	1	Albani	Tværgade 2, 5000 Odense	Denmark
4	4	Tuborg Pilsner	4.6	Pilsner	2	2	Carlsberg	Vestre Ringvej 111, 7000 Fredericia	Denmark
5	5	Tuborg Classic	4.6	Pilsner	2	2	Carlsberg	Vestre Ringvej 111, 7000 Fredericia	Denmark
6	6	Tuborg Raar	4.5	Organic Pilsner	2	2	Carlsberg	Vestre Ringvej 111, 7000 Fredericia	Denmark
7	7	Red Tuborg	4.3	Pilsner	2	2	Carlsberg	Vestre Ringvej 111, 7000 Fredericia	Denmark
8	8	Baltika Dark Brown Ale	5	Brown Ale	2	2	Carlsberg	Vestre Ringvej 111, 7000 Fredericia	Denmark
9	9	Brewmasters Collection Irish Red Ale	4.6	Ale	2	2	Carlsberg	Vestre Ringvej 111, 7000 Fredericia	Denmark
10	10	Brooklyn Summer Ale	5	Ale	2	2	Carlsberg	Vestre Ringvej 111, 7000 Fredericia	Denmark
11	11	Heineken	4.4	Pilsner	3	3	Heineken	Stadhouderskade 78, 1072 AE Amsterdam	Netherlands
12	12	Sol	4.4	Pilsner	3	3	Heineken	Stadhouderskade 78, 1072 AE Amsterdam	Netherlands

SQL JOIN, kolonner

```
SELECT beer.name, beer.alc, beer.kind, brewery.name  
FROM beer  
JOIN brewery on brewery.id = beer.brewery_id
```

	name	alc	kind	name
1	Odense Classic	4.6	Pilsner	Albani
2	Odense Pilsner	4.6	Pilsner	Albani
3	Mosaic IPA	5.7	IPA	Albani
4	Tuborg Pilsner	4.6	Pilsner	Carlsberg
5	Tuborg Classic	4.6	Pilsner	Carlsberg
6	Tuborg Raab	4.5	Organic Pilsner	Carlsberg
7	Red Tuborg	4.3	Pilsner	Carlsberg
8	Baltika Dark Brown Ale	5	Brown Ale	Carlsberg
9	Brewmasters Collection Irish Red Ale	4.6	Ale	Carlsberg
10	Brooklyn Summer Ale	5	Ale	Carlsberg
11	Heineken	4.4	Pilsner	Heineken
12	Sol	4.4	Pilsner	Heineken

SQL JOIN, kolonner

```
SELECT beer.name, beer.alc, beer.kind, brewery.name
FROM beer
JOIN brewery on brewery.id = beer.brewery_id
```

- Så nu kan vi blande information fra de to tabeller

	name	alc	kind	name
1	Odense Classic	4.6	Pilsner	Albani
2	Odense Pilsner	4.6	Pilsner	Albani
3	Mosaic IPA	5.7	IPA	Albani
4	Tuborg Pilsner	4.6	Pilsner	Carlsberg
5	Tuborg Classic	4.6	Pilsner	Carlsberg
6	Tuborg Raab	4.5	Organic Pilsner	Carlsberg
7	Red Tuborg	4.3	Pilsner	Carlsberg
8	Baltika Dark Brown Ale	5	Brown Ale	Carlsberg
9	Brewmasters Collection Irish Red Ale	4.6	Ale	Carlsberg
10	Brooklyn Summer Ale	5	Ale	Carlsberg
11	Heineken	4.4	Pilsner	Heineken
12	Sol	4.4	Pilsner	Heineken

SQL JOIN, afgrænsning

```
SELECT beer.name, beer.alc, beer.kind, brewery.name
FROM beer
JOIN brewery on brewery.id = beer.brewery_id
WHERE beer.kind = 'Pilsner'
```

	name	alc	kind	name
1	Odense Classic	4.6	Pilsner	Albani
2	Odense Pilsner	4.6	Pilsner	Albani
3	Tuborg Pilsner	4.6	Pilsner	Carlsberg
4	Tuborg Classic	4.6	Pilsner	Carlsberg
5	Red Tuborg	4.3	Pilsner	Carlsberg
6	Heineken	4.4	Pilsner	Heineken
7	Sol	4.4	Pilsner	Heineken

SQL JOIN, afgrænsning

```
SELECT beer.name, beer.alc, beer.kind, brewery.name  
FROM beer  
JOIN brewery on brewery.id = beer.brewery_id  
WHERE beer.kind = 'Pilsner'
```

- Og vi kan afgrænse den data vi henter

	name	alc	kind	name
1	Odense Classic	4.6	Pilsner	Albani
2	Odense Pilsner	4.6	Pilsner	Albani
3	Tuborg Pilsner	4.6	Pilsner	Carlsberg
4	Tuborg Classic	4.6	Pilsner	Carlsberg
5	Red Tuborg	4.3	Pilsner	Carlsberg
6	Heineken	4.4	Pilsner	Heineken
7	Sol	4.4	Pilsner	Heineken

SQL i python

- Okay, nu har vi lært hvad en database, RDBMS og SQL er for noget
 - Der er meget mere til emnet, men det her er en fin start!
- Nu skal vi se hvordan vi kan køre de her kommandoer fra python
 - Det er de samme principper, men vi køre det bare inde fra python
 - Vha. sqlite3 biblioteket (Som er i python som standard!)
- SQLite betyder at man gemmer dataen i en lokal "db/SQL" fil, og ikke snakker med en ekstern server
 - Det er dog ekstremt nemt at skifte imellem de to, det er stortset samme kode
 - Det bliver ikke en ekstern server i dag fordi det kan give meget bøvl

SQLite3 i Python

```
1  import sqlite3 # Import the sqlite3 library
2
3  con = sqlite3.connect('mydb.db') # We connect to the local database, a file called stock2.db
4  cur = con.cursor() # We create a cursor, which we can use to execute SQL statements
5
6  # We create the brewery table:
7  cur.execute("CREATE TABLE brewery (id INTEGER PRIMARY KEY AUTOINCREMENT, name VARCHAR(256), address VARCHAR(256), country VARCHAR(256));");
8
9  # We populate (insert into) it:
10 cur.execute("INSERT INTO brewery (name, address, country) VALUES"+
11             " ('Albani', 'Tværgade 2, 5000 Odense', 'Denmark')," +
12             " ('Carlsberg', 'Vestre Ringvej 111, 7000 Fredericia', 'Denmark')," +
13             " ('Heineken', 'Stadhouderskade 78, 1072 AE Amsterdam', 'Netherlands')");
14
15 con.commit() # We commit our changes to the database (save it)
16
17 result = cur.execute("SELECT * FROM brewery;") # We can now query our database, here i just SELECT *
18 data = result.fetchall() # We save all the data in an array
```

I stedet for serial! Serial findes ikke i sqlite....

SQLite3 i Python

```
1 import sqlite3 # Import the sqlite3 library
2
3 con = sqlite3.connect('mydb.db') # We connect to the local database, a file called stock2.db
4 cur = con.cursor() # We create a cursor, which we can use to execute SQL statements
5
6 # We create the brewery table:
7 cur.execute("CREATE TABLE brewery (id INTEGER PRIMARY KEY AUTOINCREMENT, name VARCHAR(256), address VARCHAR(256), country VARCHAR(256));");
8
9 # We populate (insert into) it:
10 cur.execute("INSERT INTO brewery (name, address, country) VALUES"+
11             " ('Albani', 'Tværgade 2, 5000 Odense', 'Denmark')," +
12             " ('Carlsberg', 'Vestre Ringvej 111, 7000 Fredericia', 'Denmark')," +
13             " ('Heineken', 'Stadhouderskade 78, 1072 AE Amsterdam', 'Netherlands')");
14
15 con.commit() # We commit our changes to the database (save it)
16
17 result = cur.execute("SELECT * FROM brewery;") # We can now query our database, here i just SELECT *
18 data = result.fetchall() # We save all the data in an array
```

I stedet for serial! Serial findes ikke i sqlite....

SQLite3 i Python

```
1 import sqlite3 # Import the sqlite3 library
2
3 con = sqlite3.connect('mydb.db') # We connect to the local database, a file called stock2.db
4 cur = con.cursor() # We create a cursor, which we can use to execute SQL statements
5
6 # We create the brewery table:
7 cur.execute("CREATE TABLE brewery (id INTEGER PRIMARY KEY AUTOINCREMENT, name VARCHAR(256), address VARCHAR(256), country VARCHAR(256));");
8
9 # We populate (insert into) it:
10 cur.execute("INSERT INTO brewery (name, address, country) VALUES"+
11             "('Albani', 'Tværgade 2, 5000 Odense', 'Denmark')," +
12             "('Carlsberg', 'Vestre Ringvej 111, 7000 Fredericia', 'Denmark')," +
13             "('Heineken', 'Stadhouderskade 78, 1072 AE Amsterdam', 'Netherlands')");
14
15 con.commit() # We commit our changes to the database (save it)
16
17 result = cur.execute("SELECT * FROM brewery;") # We can now query our database, here i just SELECT *
18 data = result.fetchall() # We save all the data in an array
```

- Så man skriver bare sine ”SQL statements” som strenge, og giver dem til .execute metoden!

I stedet for serial! Serial findes ikke i sqlite....

SQLite3 i Python

```
1 import sqlite3 # Import the sqlite3 library
2
3 con = sqlite3.connect('mydb.db') # We connect to the local database, a file called stock2.db
4 cur = con.cursor() # We create a cursor, which we can use to execute SQL statements
5
6 # We create the brewery table:
7 cur.execute("CREATE TABLE brewery (id INTEGER PRIMARY KEY AUTOINCREMENT, name VARCHAR(256), address VARCHAR(256), country VARCHAR(256));");
8
9 # We populate (insert into) it:
10 cur.execute("INSERT INTO brewery (name, address, country) VALUES"+
11             "('Albani', 'Tværgade 2, 5000 Odense', 'Denmark')," +
12             "('Carlsberg', 'Vestre Ringvej 111, 7000 Fredericia', 'Denmark')," +
13             "('Heineken', 'Stadhouderskade 78, 1072 AE Amsterdam', 'Netherlands')");
14
15 con.commit() # We commit our changes to the database (save it)
16
17 result = cur.execute("SELECT * FROM brewery;") # We can now query our database, here i just SELECT *
18 data = result.fetchall() # We save all the data in an array
```

- Så man skriver bare sine "SQL statements" som strenge, og giver dem til .execute metoden!
- Live!

SQLite3 – SQL filer

- Man kan også have deciderede SQL filer, som til højre →
 - Det kan være ret praktisk, specielt når man fylder data
 - Dem kan man så importere og køre direkte i python

```
populate.sql
1 CREATE TABLE brewery (
2     id INTEGER PRIMARY KEY AUTOINCREMENT,
3     name VARCHAR(256),
4     address VARCHAR(256),
5     country VARCHAR(256)
6 );
7
8 CREATE TABLE beer (
9     id INTEGER PRIMARY KEY AUTOINCREMENT,
10    name VARCHAR(256),
11    alc FLOAT(4),
12    kind VARCHAR(256),
13    brewery_id BIGINT,
14    FOREIGN KEY (brewery_id) REFERENCES Brewery(id)
15 );
16
17 INSERT INTO brewery(name, address, country)
18 VALUES
19 ('Albani', 'Tværgade 2, 5000 Odense', 'Denmark'),
20 ('Carlsberg', 'Vestre Ringvej 111, 7000 Fredericia', 'Denmark'),
21 ('Heineken', 'Stadhouderskade 78, 1072 AE Amsterdam', 'Netherlands');
22
23 INSERT INTO beer(name, alc, kind, brewery_id)
24 VALUES
25 ('Odense Classic', 4.6, 'Pilsner', 1),
26 ('Odense Pilsner', 4.6, 'Pilsner', 1),
27 ('Mosaic IPA', 5.7, 'IPA', 1),
28 ('Tuborg Pilsner', 4.6, 'Pilsner', 2),
29 ('Tuborg Classic', 4.6, 'Pilsner', 2),
30 ('Tuborg Raa', 4.5, 'Organic Pilsner', 2),
31 ('Red Tuborg', 4.3, 'Pilsner', 2),
32 ('Baltika Dark Brown Ale', 5, 'Brown Ale', 2),
33 ('Brewmasters Collection Irish Red Ale', 4.6, 'Ale', 2),
34 ('Brooklyn Summer Ale', 5, 'Ale', 2),
35 ('Heineken', 4.4, 'Pilsner', 3),
36 ('Sol', 4.4, 'Pilsner', 3);
```

SQLite3 – SQL filer

```
1  import sqlite3 # We import sqlite3
2
3  with open('populate.sql', 'r') as sql_file: # We open our sql file
4      sql_script = sql_file.read() # We read it into a "script (just a string)"
5
6  db = sqlite3.connect('test.db') # We connect to our database (or create it)
7  cur = db.cursor() # We get the cursor
8  cur.executescript(sql_script) # We execute the entire script
9
10 # We can then execute SQL statements
11 res = cur.execute("SELECT beer.name, beer.alc, beer.kind, brewery.name " +
12                  "FROM beer " +
13                  "JOIN brewery on brewery.id = beer.brewery_id " +
14                  "WHERE beer.kind = 'Pilsner' ")
15
16 # And save our data
17 data = res.fetchall()
```

SQLite3 – SQL filer

```
1  import sqlite3 # We import sqlite3
2
3  with open('populate.sql', 'r') as sql_file: # We open our sql file
4      sql_script = sql_file.read() # We read it into a "script (just a string)"
5
6  db = sqlite3.connect('test.db') # We connect to our database (or create it)
7  cur = db.cursor() # We get the cursor
8  cur.executescript(sql_script) # We execute the entire script
9
10 # We can then execute SQL statements
11 res = cur.execute("SELECT beer.name, beer.alc, beer.kind, brewery.name " +
12                  "FROM beer " +
13                  "JOIN brewery on brewery.id = beer.brewery_id " +
14                  "WHERE beer.kind = 'Pilsner' ")
15
16 # And save our data
17 data = res.fetchall()
```

- Live!

Opgaver pt. 2 (30-40 minutter)

1. Download først vores Beer+Brewery SQL fil:

https://raw.githubusercontent.com/AndersBensen/python_101/main/python3/populate.sql

Der er 3 skridt i den her opgave:

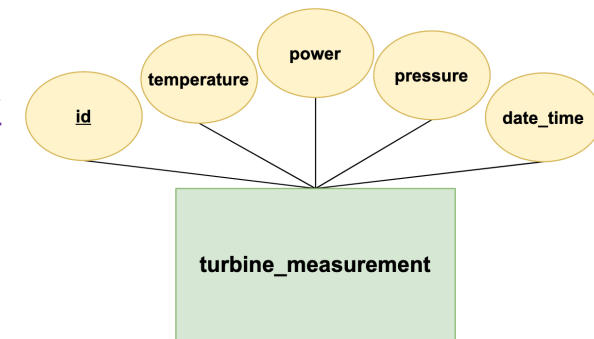
- Først få kørt SQL filen som jeg viste i tidligere et eksempel og sørg for at databasen er "populated".
- Herefter indsæt 3 af dine egne yndlings øl i databasen igennem python. Hvis ikke det tilsvarende bryggeri er der, så indsæt først det (og husk at det får id 4).
- Til sidst hent alle de øl fra databasen der har en procent over 4,5 og print dem ud i konsollen.

2. Lav en Beer klasse og en Brewery klasse i python. Hent herefter alle øl og bryggerier ud fra vores database og gem hver række som et tilsvarende python objekt.

- Klasserne skal sådan set bare bestå af de attributter de har i databasen.
- Lav en string metode i hver klasse (def __str__(self)) som returnerer objektets indhold som en string.
- Gem øl objekterne i en liste, og gem bryggeri objekterne i en anden liste.
- Print nu hver eneste objekt ud i konsollen.

3. (Ekstra) Lav en ny database ud fra ER diagrammet til højre. Fyld den med dataen fra det her link: https://github.com/AndersBensen/python_101/raw/main/python3/turbine_data.txt

- Hent herefter alle temperatur rækker ud fra databasen, gem dem i en liste og plot dem med matplotlib.
- Hints:
 - Filen er komma (",") separeret, du kan f.eks. læse den ind med *open* metoden i python.
 - Så hver række du læser ind fra filen, skal du så tilføje (INSERT INTO) til databasen.
 - Data typerne kan f.eks. Være: INTEGER; FLOAT, FLOAT, INTEGER, VARCHAR



HTTP, REST & API

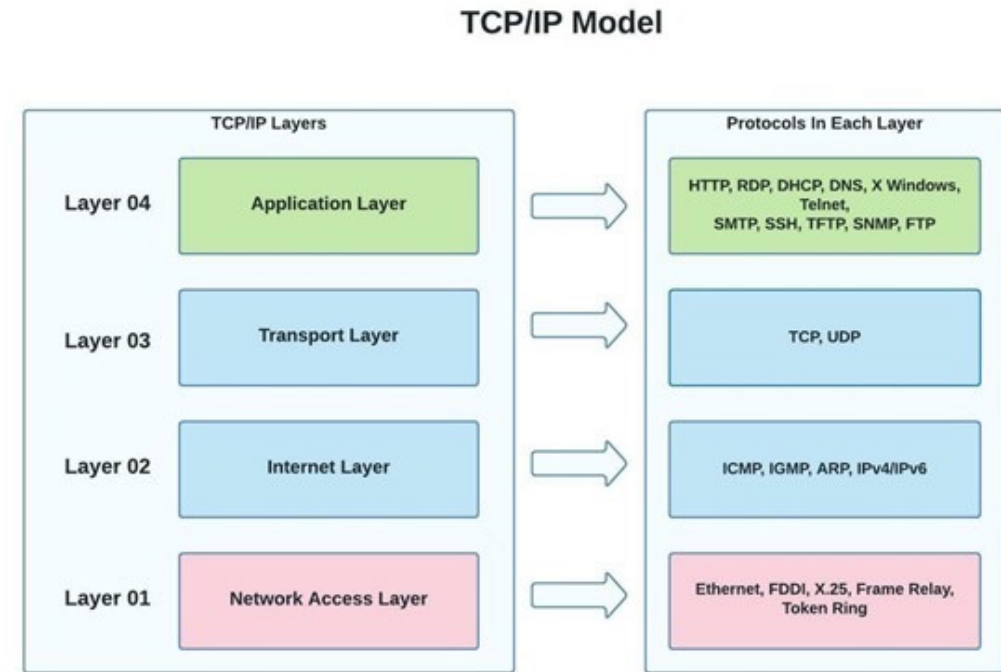
Computer netværk

- ”Et sæt af computere der kommunikerer over protokoller”
 - Protokol -> En standard der beskriver hvordan man kommunikerer
 - ... På plads 3 i en streng af bits (001000) har vi ...
- Internettet og netværk generelt er bygget i lag
 - I det nederste lag sendes der ”fysiske” signaler igennem luften
 - En computer fortolker så de her signaler som 0’er og 1’ere
 - Som så kan oversættes til bogstaver/tal
- De her forskellige lag til sammen hedder f.eks. TCP/IP modellen
- **Enhver computer har en IP adresse TODO**



TCP/IP

- En "model" der beskriver forskellige lag
 - I internettet / et netværk
- Nederste lag (Network Access) er på et "lavt niveau"
 - F.eks. Hvordan skal bits (0,1) sendes og fortolkes
 - Laget efter bygger så oven på det
 - Osv.
- Øverst har vi så "Applikations" laget
 - Meget højt niveau, og her er HTTP
 - F.eks.: [GET http://www.google.com](http://www.google.com)
 - Så når man bruger f.eks. Google bliver det oversat ned igennem lagene
 - Og til sidst er det bare 0'ere og 1'ere, men det er abstraheret væk for os!

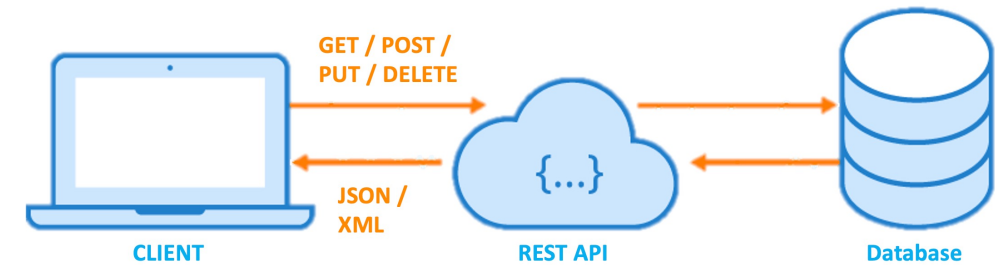


HTTP

- **HyperText Transfer Protocol (HTTP)**
 - Bruges til at hente/sende ressourcer på internettet
- I bruger den formentlig hver dag når i går ind på en hjemmeside
 - Så i har forespurgt (***request***) en ressource (f.eks. <http://www.google.com>)
 - Jeres browser henter så den tilsvarende side (vha. HTTP!) og viser den flot frem
- HTTP er kendt for 8 handlinger (HTTP verbs), de 4 vigtigste:
 - POST
 - GET
 - PUT
 - DELETE

API

- **Application Programmable Interface**
 - En grænseflade man kan programmere op af
- F.eks. Har NemID/MitID et API
 - Vi gider ikke selv verificerer brugeres offentlige identitet
 - Så derfor bruger vi NemID's API til at sikre det!
- Der findes API'er til nærmest alt
 - Facebook, Vejret, Google Maps, Aktiepriser....
 - I dag skal i både kalde API'er fra Python, men også lave jeres eget!
- REST API
 - En slags standard til at bygge HTTP API'er
 - <https://restfulapi.net/>



JSON

- **JavaScript Object Notation**
 - Dog ikke begrænset til programmeringssproget JavaScript
- Et data format der ofte bliver brugt i HTTP/REST API'er
 - Minder lidt om objekter i OOP
- Ofte når man kalder et API får man et svar tilbage i JSON
 - Og hvis man skal give noget data med forventes også JSON

JSON

```
1  {
2      "person": {
3          "name": "Anders",
4          "height": 1.72,
5          "age": 26,
6          "married": false,
7          "favoriteFood": [
8              {"name": "lasagna"},
9              {"name": "pizza"},
10             {"name": "sushi"}
11         ]
12     }
13 }
```

- Objekter er i {}
- Der er de datatyper vi kender
 - int, string, double, boolean
- Attributter har et navn og værdien sættes med :
- Lister er givet ved []
 - Og kan så indeholde flere objekter
- I virkeligheden er det faktisk som et dict i python

JSON

```
1  {
2      "person": {
3          "name": "Anders",
4          "height": 1.72,
5          "age": 26,
6          "married": false,
7          "favoriteFood": [
8              {"name": "lasagna"},
9              {"name": "pizza"},
10             {"name": "sushi"}
11         ]
12     }
13 }
```

```
1  {
2      "person": {
3          "name": "Anders",
4          "height": 1.72,
5          "age": 26,
6          "married": false,
7          "favoriteFood": [
8              "lasagna", "pizza", "sushi"
9          ]
10     }
11 }
```

- Højre er også gyldigt

Requests

- Requests er et bibliotek i python til at lave HTTP requests med
 - Understøtter alle de ting vi har talt om
 - HTTP verbs
 - POST, PUT, GET, DELETE
 - JSON
 - Man kan få sit svar i JSON
- <https://api.coindesk.com/v1/bpi/currentprice.json>
 - Et API til at hente den nuværende pris for Bitcoin (BTC)
 - Kan vi f.eks. gøre i gennem requests

Requests i Python

```
1  import requests
2
3  url = "https://api.coindesk.com/v1/bpi/currentprice.json"
4  response = requests.get(url)
5  response_json = response.json()
```

Requests i Python

```
1 import requests
2
3 url = "https://api.coindesk.com/v1/bpi/currentprice.json"
4 response = requests.get(url)
5 response_json = response.json()
```

```
1 {
2   "time": {
3     "updated": "Sep 20 2022 13: 29: 00 UTC",
4     "updatedISO": "2022-09-20T13: 29: 00+00: 00",
5     "updateduk": "Sep 20 2022 at 14: 29 BST"
6   },
7   "disclaimer": "This data was produced from the CoinDesk Bitcoin
8     Price Index (USD). Non-USD currency data converted using
9     hourly conversion rate from openexchangerates.org",
10  "chartName": "Bitcoin",
11  "bpi": {
12    "USD": {
13      "code": "USD",
14      "symbol": "&#36;",
15      "rate": "18,919.3239",
16      "description": "United States Dollar",
17      "rate_float": 18919.3239
18    },
19    "GBP": {
20      "code": "GBP",
21      "symbol": "&pound;",
22      "rate": "15,808.8357",
23      "description": "British Pound Sterling",
24      "rate_float": 15808.8357
25    },
26    "EUR": {
27      "code": "EUR",
28      "symbol": "&euro;",
29      "rate": "18,430.1837",
30      "description": "Euro",
31      "rate_float": 18430.1837
32    }
33  }
34 }
```

Requests i Python

```
1 import requests
2
3 url = "https://api.coindesk.com/v1/bpi/currentprice.json"
4 response = requests.get(url)
5 response_json = response.json()
6
7 response_json['chartName'] # BitCoin
8 response_json['bpi']
9
10 response_json['bpi']['USD']['rate_float'] # 18919.3239
```

```
1 {
2   "time": {
3     "updated": "Sep 20 2022 13: 29: 00 UTC",
4     "updatedISO": "2022-09-20T13: 29: 00+00: 00",
5     "updateduk": "Sep 20 2022 at 14: 29 BST"
6   },
7   "disclaimer": "This data was produced from the CoinDesk Bitcoin
8     Price Index (USD). Non-USD currency data converted using
9     hourly conversion rate from openexchangerates.org",
10  "chartName": "Bitcoin",
11  "bpi": {
12    "USD": {
13      "code": "USD",
14      "symbol": "&#36;",
15      "rate": "18,919.3239",
16      "description": "United States Dollar",
17      "rate_float": 18919.3239
18    },
19    "GBP": {
20      "code": "GBP",
21      "symbol": "&pound;",
22      "rate": "15,808.8357",
23      "description": "British Pound Sterling",
24      "rate_float": 15808.8357
25    },
26    "EUR": {
27      "code": "EUR",
28      "symbol": "&euro;",
29      "rate": "18,430.1837",
30      "description": "Euro",
31      "rate_float": 18430.1837
32    }
33  }
34 }
```


Requests i Python

```
1 import requests
2
3 url = "https://api.coindesk.com/v1/bpi/currentprice.json"
4 response = requests.get(url)
5 response_json = response.json()
6
7 response_json['chartName'] # BitCoin
8 response_json['bpi']
9
10 response_json['bpi']['USD']['rate_float'] # 18919.3239
```

```
1 {
2   "time": {
3     "updated": "Sep 20 2022 13: 29: 00 UTC",
4     "updatedISO": "2022-09-20T13: 29: 00+00: 00",
5     "updateduk": "Sep 20 2022 at 14: 29 BST"
6   },
7   "disclaimer": "This data was produced from the CoinDesk Bitcoin
8     Price Index (USD). Non-USD currency data converted using
9     hourly conversion rate from openexchangerates.org",
10  "chartName": "Bitcoin",
11  "bpi": {
12    "USD": {
13      "code": "USD",
14      "symbol": "&#36;",
15      "rate": "18,919.3239",
16      "description": "United States Dollar",
17      "rate_float": 18919.3239
18    },
19    "GBP": {
20      "code": "GBP",
21      "symbol": "&pound;",
22      "rate": "15,808.8357",
23      "description": "British Pound Sterling",
24      "rate_float": 15808.8357
25    },
26    "EUR": {
27      "code": "EUR",
28      "symbol": "&euro;",
29      "rate": "18,430.1837",
30      "description": "Euro",
31      "rate_float": 18430.1837
32    }
33  }
34 }
```

Requests i Python

```
1 import requests
2
3 url = "https://api.coindesk.com/v1/bpi/currentprice.json"
4 response = requests.get(url)
5 response_json = response.json()
6
7 response_json['chartName'] # BitCoin
8 response_json['bpi']
9
10 response_json['bpi']['USD']['rate_float'] # 18919.3239
```

```
1 {
2   "time": {
3     "updated": "Sep 20 2022 13: 29: 00 UTC",
4     "updatedISO": "2022-09-20T13: 29: 00+00: 00",
5     "updateduk": "Sep 20 2022 at 14: 29 BST"
6   },
7   "disclaimer": "This data was produced from the CoinDesk Bitcoin
8     Price Index (USD). Non-USD currency data converted using
9     hourly conversion rate from openexchangerates.org",
10  "chartName": "Bitcoin",
11  "bpi": {
12    "USD": {
13      "code": "USD",
14      "symbol": "&#36;",
15      "rate": "18,919.3239",
16      "description": "United States Dollar",
17      "rate_float": 18919.3239
18    },
19    "GBP": {
20      "code": "GBP",
21      "symbol": "&pound;",
22      "rate": "15,808.8357",
23      "description": "British Pound Sterling",
24      "rate_float": 15808.8357
25    },
26    "EUR": {
27      "code": "EUR",
28      "symbol": "&euro;",
29      "rate": "18,430.1837",
30      "description": "Euro",
31      "rate_float": 18430.1837
32    }
33  }
34 }
```

Requests i Python

```
1 import requests
2
3 url = "https://api.coindesk.com/v1/bpi/currentprice.json"
4 response = requests.get(url)
5 response_json = response.json()
6
7 response_json['chartName'] # BitCoin
8 response_json['bpi']
9
10 response_json['bpi']['USD']['rate_float'] # 18919.3239
```

```
1 {
2   "time": {
3     "updated": "Sep 20 2022 13: 29: 00 UTC",
4     "updatedISO": "2022-09-20T13: 29: 00+00: 00",
5     "updateduk": "Sep 20 2022 at 14: 29 BST"
6   },
7   "disclaimer": "This data was produced from the CoinDesk Bitcoin
8     Price Index (USD). Non-USD currency data converted using
9     hourly conversion rate from openexchangerates.org",
10  "chartName": "Bitcoin",
11  "bpi": {
12    "USD": {
13      "code": "USD",
14      "symbol": "&#36;",
15      "rate": "18,919.3239",
16      "description": "United States Dollar",
17      "rate_float": 18919.3239
18    },
19    "GBP": {
20      "code": "GBP",
21      "symbol": "&pound;",
22      "rate": "15,808.8357",
23      "description": "British Pound Sterling",
24      "rate_float": 15808.8357
25    },
26    "EUR": {
27      "code": "EUR",
28      "symbol": "&euro;",
29      "rate": "18,430.1837",
30      "description": "Euro",
31      "rate_float": 18430.1837
32    }
33  }
34 }
```

Requests i Python

```
1 import requests
2
3 url = "https://api.coindesk.com/v1/bpi/currentprice.json"
4 response = requests.get(url)
5 response_json = response.json()
6
7 response_json['chartName'] # BitCoin
8 response_json['bpi']
9
10 response_json['bpi']['USD']['rate_float'] # 18919.3239
```

- Live

```
1 {
2   .... "time": {
3   ....   .... "updated": "Sep 20 2022 13: 29: 00 UTC",
4   ....   .... "updatedISO": "2022-09-20T13: 29: 00+00: 00",
5   ....   .... "updateduk": "Sep 20 2022 at 14: 29 BST"
6   .... },
7   .... "disclaimer": "This data was produced from the CoinDesk Bitcoin
8   ....   Price Index (USD). Non-USD currency data converted using
9   ....   hourly conversion rate from openexchangerates.org",
10  .... "chartName": "Bitcoin",
11  .... "bpi": {
12  ....   .... "USD": {
13  ....     .... "code": "USD",
14  ....     .... "symbol": "&#36;",
15  ....     .... "rate": "18,919.3239",
16  ....     .... "description": "United States Dollar",
17  ....     .... "rate_float": 18919.3239
18  ....   },
19  ....   .... "GBP": {
20  ....     .... "code": "GBP",
21  ....     .... "symbol": "&pound;",
22  ....     .... "rate": "15,808.8357",
23  ....     .... "description": "British Pound Sterling",
24  ....     .... "rate_float": 15808.8357
25  ....   },
26  ....   .... "EUR": {
27  ....     .... "code": "EUR",
28  ....     .... "symbol": "&euro;",
29  ....     .... "rate": "18,430.1837",
30  ....     .... "description": "Euro",
31  ....     .... "rate_float": 18430.1837
32  ....   }
33  .... }
34  .... }
```

Flask

- Et bibliotek til at opstille API'er med i Python
- Understøtter også alt det vi har talt om i dag
- Det bitcoin API vi lige "requestede" kunne sagtens være lavet i Flask

Flask i Python

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/hello/<name>')
6  def hello_world(name):
7      return 'Hello ' + name
8
9  if __name__ == '__main__':
10     app.run(port=8000)
```

Flask i Python

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/hello/<name>')
6  def hello_world(name):
7      return 'Hello ' + name
8
9  if __name__ == '__main__':
10     app.run(port=8000)
```

API "ruten" til metoden, <name> er et argument!

Flask i Python

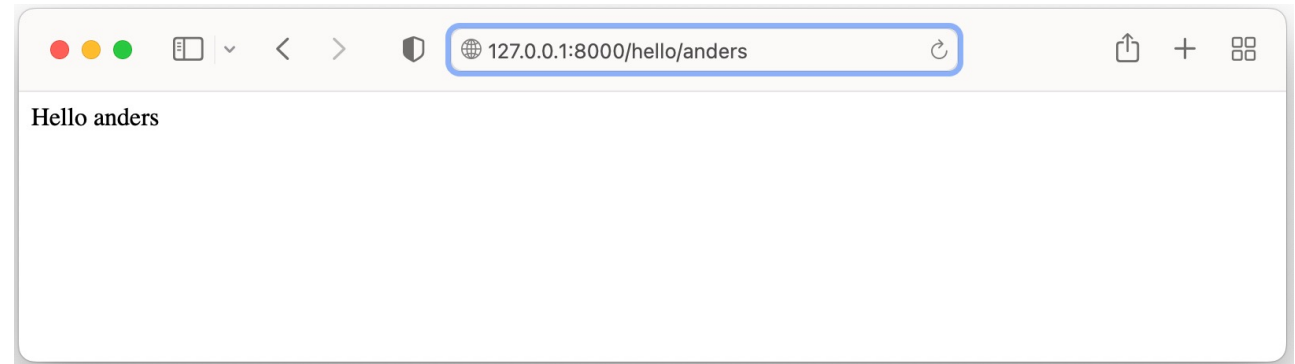
```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/hello/<name>')
6  def hello_world(name):
7      return 'Hello ' + name
8
9  if __name__ == '__main__':
10     app.run(port=8000)
```

API "ruten" til metoden, <name> er et argument!

Starter API'et på en given port

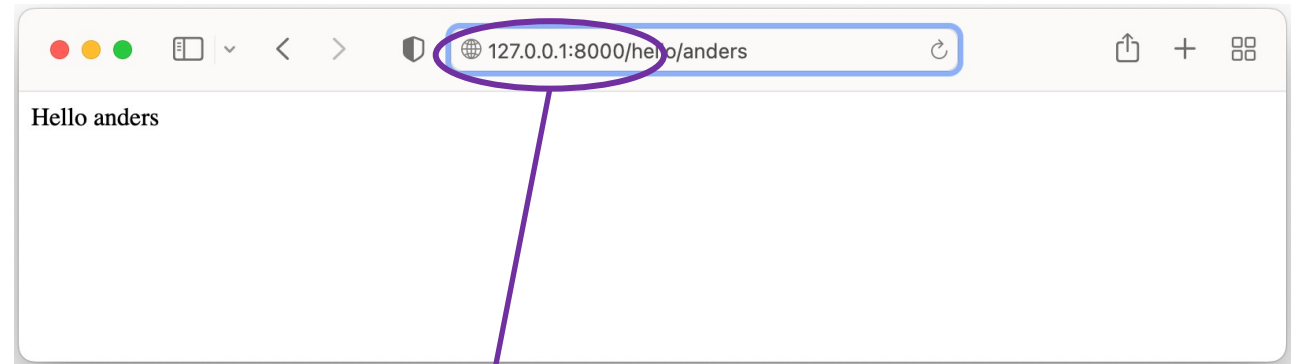
Flask i Python

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/hello/<name>')
6  def hello_world(name):
7      return 'Hello ' + name
8
9  if __name__ == '__main__':
10     app.run(port=8000)
```



Flask i Python

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/hello/<name>')
6  def hello_world(name):
7      return 'Hello ' + name
8
9  if __name__ == '__main__':
10     app.run(port=8000)
```

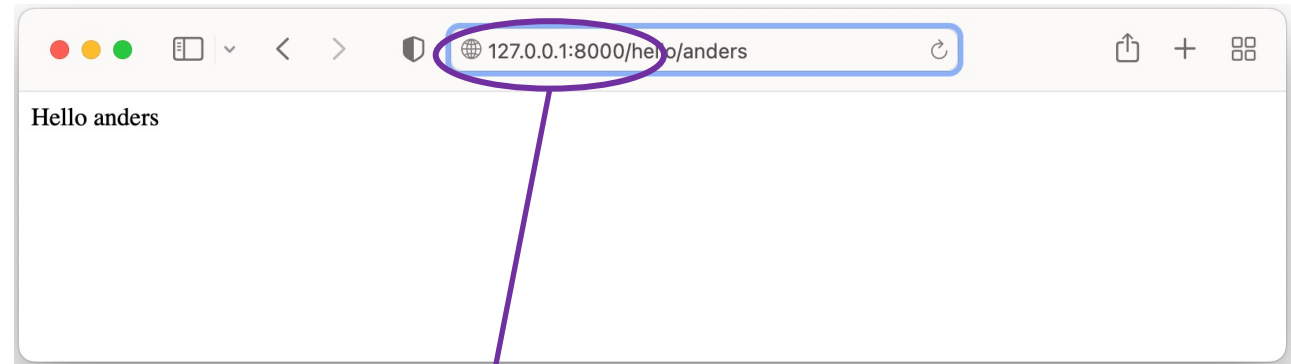


127.0.0.1 er en ip-adresse og 8000 er en port.
Det er sådan cirka bare det samme som et navn på en hjemmeside. Google.dk kører f.eks. på ip adressen 142.250.181.206 og port 80.

Flask i Python

```
1  from flask import Flask
2
3  app = Flask(__name__)
4
5  @app.route('/hello/<name>')
6  def hello_world(name):
7      return 'Hello ' + name
8
9  if __name__ == '__main__':
10     app.run(port=8000)
```

- Live



127.0.0.1 er en ip-adresse og 8000 er en port.
Det er sådan cirka bare det samme som et navn på en hjemmeside. Google.dk kører f.eks. på ip adressen 142.250.181.206 og port 80.

Flask, flere argumenter & retur JSON

```
1  from flask import Flask, request
2
3  app = Flask(__name__)
4
5  @app.route('/listrange', methods=['GET'])
6  def listrange():
7      min = int(request.args.get('min'))
8      max = int(request.args.get('max'))
9      rangelist = list(range(min, max+1))
10
11     return { "range" : rangelist }
12
13 if __name__ == '__main__':
14     app.run(port=8000)
```



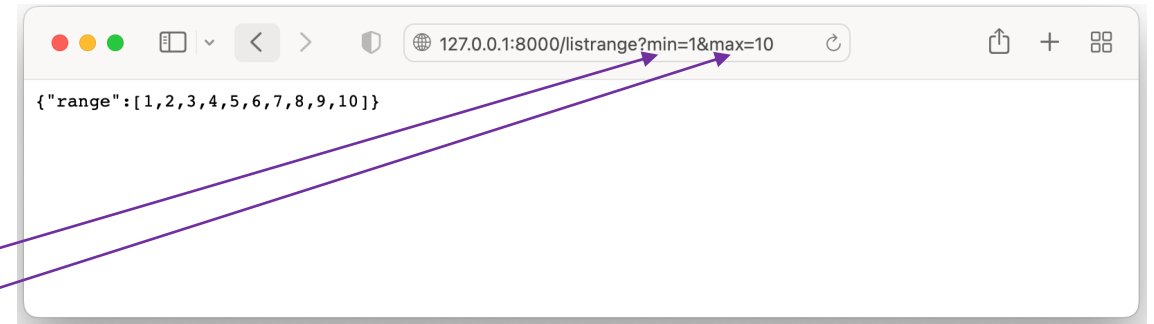
Flask, flere argumenter & retur JSON

```
1 from flask import Flask, request
2
3 app = Flask(__name__)
4
5 @app.route('/listrange', methods=['GET'])
6 def listrange():
7     min = int(request.args.get('min'))
8     max = int(request.args.get('max'))
9     rangelist = list(range(min, max+1))
10
11     return { "range" : rangelist }
12
13 if __name__ == '__main__':
14     app.run(port=8000)
```



Flask, flere argumenter & retur JSON

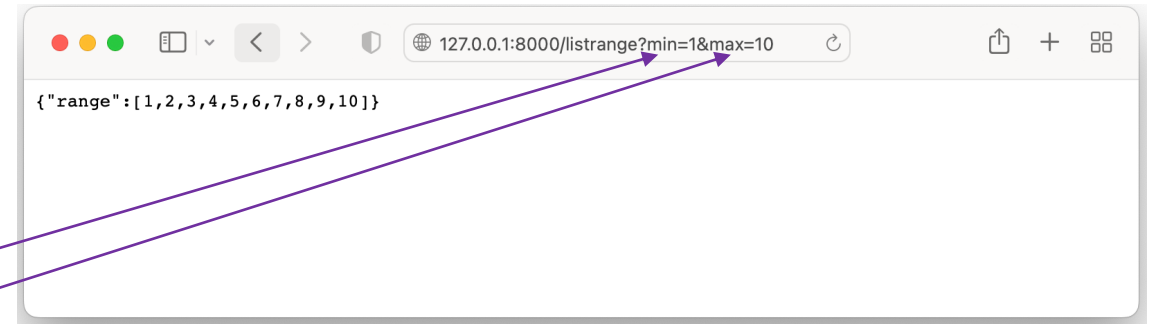
```
1 from flask import Flask, request
2
3 app = Flask(__name__)
4
5 @app.route('/listrange', methods=['GET'])
6 def listrange():
7     min = int(request.args.get('min'))
8     max = int(request.args.get('max'))
9     rangelist = list(range(min, max+1))
10
11     return { "range" : rangelist }
12
13 if __name__ == '__main__':
14     app.run(port=8000)
```



- Så vi kan give flere argumenter
 - Ved at hive dem ud af "requestet"
- Og sætte at metoden skal kunne GET
- Og vi kan returnere med {} så laver Flask det om til JSON for os

Flask, flere argumenter & retur JSON

```
1 from flask import Flask, request
2
3 app = Flask(__name__)
4
5 @app.route('/listrange', methods=['GET'])
6 def listrange():
7     min = int(request.args.get('min'))
8     max = int(request.args.get('max'))
9     rangelist = list(range(min, max+1))
10
11     return { "range" : rangelist }
12
13 if __name__ == '__main__':
14     app.run(port=8000)
```



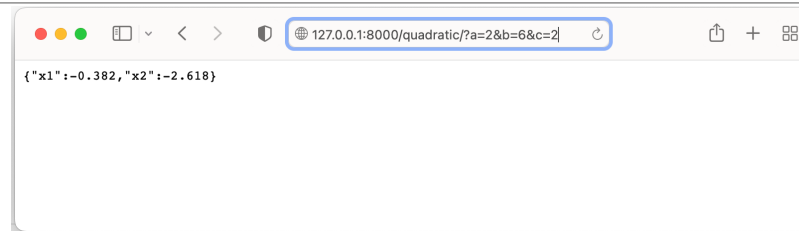
- Live
- Så vi kan give flere argumenter
 - Ved at hive dem ud af "requestet"
- Og sætte at metoden skal kunne GET
- Og vi kan returnere med {} så laver Flask det om til JSON for os

Opgaver pt. 3 (30-40 minutter)

1. Lav et API med Flask til at løse andengradsligninger og returner de to x-skæringer som JSON. Svaret skal se ud som til højre:

- *pip install flask*

- Hints: $x1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$, $x2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$



2. Kald api'et: https://api.open-meteo.com/v1/forecast?latitude=55.403756&longitude=10.402370&hourly=temperature_2m med requests biblioteket. Men udskift 55.40... og 10.40... med din Bys latitude og longitude, de kan f.eks. slås op her: <https://www.countrycoordinate.com/search/?keywords=Odense>) og gem svaret i en variabel som json.

- *pip install requests*

- Gem temperaturen ud i en variabel.

- Hints: ['hourly']['temperature_2m']

- Plot nu temperaturen over de første 24 timer vha. matplotlib.pyplot.

- *pip install matplotlib*

3. (Ekstra) Lav et API ("/beers/") med Flask til at kunne GET alle øl i din database fra opgave 2.1.

- Lav også et API til at hente alle øl af samme type ("/beer/<kind>") i databasen.

- **!VIGTIGT!**, initialiser databasen med det her argument: `db = sqlite3.connect("mydb.db", check_same_thread=False)`

- Hints:

- Lav en liste hvor du gemmer hvert øl som et JSON objekt i listen og returner den.

- Hvis du har et mellemrum i din type af øl, så kan du f.eks. skrive sådan her i URL'en: `/beer/Organic%20Pilsner` (%20 svarer til mellemrum i computer sprog).

At binde det hele sammen

- Et eksempel på hvordan man kan bruge koncepterne fra i dag sammen
 - Og hvor nemt det er at gå fra at connecte til SQLite til at connecte til en ægte database
 - Hvis man bygger det rigtigt

Afrunding

- Beklager de (måske) mange nye koncepter
 - Men i kan muntre jer op med at med de her koncepter kan man komme virkelig langt
 - Alle (99%) software firmaer bruger SQL databaser
 - Stortset alle ”services” bliver udstillet igennem REST/HTTP API’er i dag
- Læs evt. slidesne igennem flere gange og prøv at lave opgaverne igen
 - De færreste forstår det hele første gang
 - Koncepterne kræver tid og arbejde for forståelse
- Løsninger til opgaver og mini programmet kan findes her:
 - https://github.com/AndersBensen/python_101/raw/main/python3/exercises.zip

Spørgsmål?

- anders_bensen@hotmail.com