Python i praksis - Af Henrik Sterner (henrik.sterner@gmail.com)

- Introduktion til Python
- Introduktion til variabler og datatyper i Python
- Brugerinput i Python
- Strengformatering i Python
- Betinget udførelse af kode i Python
- Løkker i Python
- Liste comprehension i Python
- Funktioner i Python
- Højere ordens funktioner i Python

Om brugen af disse slides

- Disse slides forsøger at eksemplifere en lang række af de vigtige begreber i Python
- De må på ingen måde kopieres uden tilladelse fra Henrik Sterner
- De er lavet i markdown og kan derfor nemt konverteres til andre formater
- Ved brug af Visual Studio Code kan de konverteres til HTML, PDF, PowerPoint, Word, LaTeX og mange andre formater
- Slides er tilgængelige på github.com/henriksterner/IntroPython/

Konvertere slides til andre formater

- Fra markdown til pdf ved brug af pandoc:

```
pandoc -t beamer -o slidespython.pdf slidespython.md
```

- Fra markdown til word ved brug af pandoc:

```
pandoc -t docx -o slidespython.docx slidespython.md
```

- Fra markdown til pdf hvor sektioner tilføjes som hovedpunkter i pdf'en:

```
pandoc -t beamer -o slidespython.pdf slidespython.md --toc
```

Øvelser og software brugt

Det er ekstremt vigtigt at lave en masse øvelser. Da det træner brugen af de grundlæggende strukturer. Programmering er en praktisk disciplin - et håndværk. Og man bliver ikke god til et håndværk uden at øve sig.

Øvelserne er lavet så de kan løses i Python ved brug af eksempelvis Jupyter Notebook eller Google Colab.

Undlad at få hjælp fra AI m.m. Det lærer du ikke noget af.

Hvad er Google Colab?

- Google Colab er et interaktivt programmeringsmiljø
- Google Colab er en webapplikation
- Google Colab er gratis
- Google Colab er udviklet i Python
- Google Colab kan bruges til mange forskellige programmeringssprog
- Google colab tilgås via en Google konto og Google Drev
- Alle filer gemmes i Google Drev
- Google Colab kan bruges til at træne maskinlæring og deep learning modeller
- Google Colab kan bruges til at træne modeller på GPU og TPU
- Adressen er: https://colab.research.google.com/

Hvad er Jupyter Notebook?

- Jupyter Notebook er et interaktivt programmeringsmiljø
- Jupyter Notebook er en webapplikation
- Jupyter Notebook er gratis
- Jupyter Notebook er udviklet i Python
- Jupyter Notebook kan bruges til mange forskellige programmeringssprog
- Jupyter Notebook kan tilgås lokalt på computeren
- Jupyter Notebook kan installeres via Anaconda og Visual Studio Code
- Anaconda kan downloades fra: https://www.anaconda.com/products/individual
- Visual Studio Code kan downloades fra: https://code.visualstudio.com/

Installationsvejledning til Anaconda

- Gå til adressen: https://www.anaconda.com/products/individual
- Download Anaconda til dit operativsystem
- Følg installationsvejledningen
- Åbn Anaconda Navigator

Installationsvejledning til Visual Studio Code

- Gå til adressen: https://code.visualstudio.com/
- Download Visual Studio Code til dit operativsystem
- Følg installationsvejledningen

- Åbn Visual Studio Code
- Installer Jupyter-plugin
- Åbn Jupyter Notebook
- Lav en ny notebook
- Skriv noget kode i en celle
- Kør koden i cellen Koden kunne være:

```
print("Hej verden")
```

Introduktion til Python

- Python: Et populært og læsbar programmeringssprog
- Bruges bredt indenfor udvikling, automatisering og dataanalyse
- Udtales "Pai-thon"
- Navngivet efter Monty Python
- Skabt af Guido van Rossum i 1991

Hvad er Python?

- Python er et højniveausprog
- Python er et objektorienteret sprog
- Python er et dynamisk typet sprog
- Python er et open source sprog
- Python er et sprog med mange biblioteker

Datascience biblioteker med Python

- Numpy: Matematik og videnskabelige beregninger
- Pandas: Dataanalyse og datahåndtering
- Matplotlib: Visualisering af data
- Scikit-learn: Maskinlæring
- TensorFlow: Maskinlæring og deep learning
- PyTorch: Maskinlæring og deep learning

Overblik over hvad vi skal igennem

- Introduktion til variabler i Python
- Operationer på variabler i Python
- Datatyper i Python
- Brugerinput i Python
- Strengformatering i Python
- Betinget udførelse af kode i Python

- Løkker i Python
- Funktioner i Python
- Objektorienteret programmering i Python
- Fejlhåndtering i Python
- Filhåndtering i Python

Introduktion til variabler og datatyper i Python

- Variabler gemmer værdier til brug i programmet
- Navngivne lagerpladser til forskellige typer af data
- Python bruger dynamisk typetildeling
- Variabler behøver ikke erklæres med en bestemt type
- Data i variabler kan ændres undervejs i programmet

Variabelnavne

```
navn = "Alice"
alder = 30
point_1 = 95.5
er_student = True
```

Navngivningsregler for variabler

- Variabelnavne kan indeholde bogstaver, tal og understregning
- Variabelnavne må ikke starte med et tal
- Variabelnavne må ikke indeholde mellemrum
- Variabelnavne må ikke indeholde specialtegn
- Variabelnavne må ikke være Python nøgleord
- Variabelnavne bør være sigende
- Variabelnavne kan være på engelsk

Datatyper i Python

```
tekst = "Hej verden"
heltal = 42
decimaltal = 3.14
sandt_eller_falsk = True
```

Streng (Tekst)

```
besked = "Velkommen til Python"
underbesked = besked[8:12]
længde = len(besked)
```

Heltal

```
alder = 25
højde = -160
antal_børn = 3
```

Sandt eller Falsk (Boolean)

```
er_solopgang = True
er_nedbør = False
```

Listetyper

```
farver = ["rød", "grøn", "blå"]
tal = [1, 2, 3, 4, 5]
blandet = ["æble", 3, True]
```

Tupletyper

```
koordinater = (3, 7)
datarække = (1, "tekst", True)
```

Dictionarietyper

```
person = {
    "navn": "Alice",
    "alder": 30,
    "er_student": True
}
navn = person["navn"]
```

Variabelskift

```
a = 5
b = 7
a, b = b, a
print("a:", a)  # Resultat: 7
print("b:", b)  # Resultat: 5
```

Globale og Lokale Variabel

```
global_var = 10
if global_var > 5:
    lokal_var = 5
    print(global_var)
    print(lokal_var)
```

Vi vender snart tilbage til if-betingelser

Typekonvertering

```
alder = 25
alder_tekst = str(alder)
pris = "9.99"
pris_decimal = float(pris)
```

Typen af en Variabel

```
tekst = "Hej verden"
print(type(tekst))  # Resultat: <class 'str'>
alder = 25
print(type(alder))  # Resultat: <class 'int'>
pris = 9.99
print(type(pris))  # Resultat: <class 'float'>
```

Indbyggede Funktioner til Typer

```
længde = len("Hej verden")
maksimum = max([4, 7, 2, 9])
minimum = min([4, 7, 2, 9])
```

Variabler som Referencer

```
liste_a = [1, 2, 3]
liste_b = liste_a
liste_a.append(4)
print(liste_b)  # Resultat: [1, 2, 3, 4]
```

Operationer på Variabler

- Addition, subtraktion, multiplikation og division
- Potens, kvadratrod og modulo
- Sammenligning, logisk og bitvis
- Tildeling, øg og mindsk
- Andre operationer

Addition, Subtraktion, Multiplikation og Division

```
a = 5
b = 2
print(a + b)  # Resultat: 7
print(a - b)  # Resultat: 3
print(a * b)  # Resultat: 10
print(a / b)  # Resultat: 2.5
```

Bemærk at division altid giver et decimaltal. Hvis resultater skal gemmes bør der erklæres variable, som rummer resultatet af den enkelte operation.

Potens, Kvadratrod og Modulo

```
a = 5
b = 2
print(a ** b)  # Resultat: 25
print(b ** 0.5)  # Resultat: 1.4142135623730951
print(a % b)  # Resultat: 1
```

Identitet og Lighed

```
a = [1, 2, 3]
b = [1, 2, 3]
print(a is b)  # Resultat: False
print(a == b)  # Resultat: True
```

Sammenligning og Logisk operatorer

```
a = 5
b = 2
print(a > b)  # Resultat: True
print(a < b)  # Resultat: False
print(a >= b)  # Resultat: True
print(a <= b)  # Resultat: False
print(a == b)  # Resultat: False
print(a != b)  # Resultat: True
```

Typekontrol med isinstance()

```
x = 5
if isinstance(x, int):
    print("x er et heltal")
else:
    print("x er ikke et heltal")
```

Bitvis operatorer i teori

- & (bitvis og) - Hvis begge bits er 1, så er resultatet 1, ellers 0
- | (bitvis eller) - Hvis en af bits er 1, så er resultatet 1, ellers 0
- ^ (bitvis eksklusiv eller) - Hvis bits er forskellige, så er resultatet 1, ellers 0
- ~ (bitvis negation) - Hvis bit er 1, så er resultatet 0, ellers 1
- << (bitvis venstreskift) - Skift bits til venstre
- (bitvis højreskift) - Skift bits til højre

Bitvis operatorer eksempler

```
a = 5
b = 2
print(a & b)  # Resultat: 0
print(a | b)  # Resultat: 7
print(a ^ b)  # Resultat: 7
print(~a)  # Resultat: -6
print(a << 1)  # Resultat: 10
print(a >> 1)  # Resultat: 2
```

Hvad er brugerinput i Python?

- Input fra brugeren
- Input fra tastaturet
- Input fra en fil
- Input fra en database
- Input fra en sensor
- Input fra en anden computer
- Input fra en anden enhed Vi kan bruge kommadoen input() til at få brugerinput i Python

Brugerinput med input()

```
navn = input("Hvad hedder du? ")
alder = int(input("Hvor gammel er du? "))
```

Brugerinput med input() og typekonvertering

```
alder = int(input("Hvor gammel er du? "))
højde = float(input("Hvor høj er du i meter? "))
```

Brugerinput med input() med typetjek og fejlhåndtering

```
while True:
    alder = input("Hvor gammel er du? ")
    if alder.isdigit():
        alder = int(alder)
        break
    else:
        print("Indtast venligst et heltal")
```

Brugerinput bmi-beregner

```
højde = float(input("Hvor høj er du i meter? "))
vægt = float(input("Hvor meget vejer du i kg? "))
bmi = vægt / (højde ** 2)
print("Din BMI er: " + str(bmi))
```

Brugerinput andengradsligning

```
a = float(input("Indtast a: "))
b = float(input("Indtast b: "))
c = float(input("Indtast c: "))
d = b ** 2 - 4 * a * c
if d < 0:
    print("Ligningen har ingen løsninger")
elif d == 0:
    x = -b / (2 * a)
    print("Ligningen har en løsning: " + str(x))
else:
    x1 = (-b + d ** 0.5) / (2 * a)
    x2 = (-b - d ** 0.5) / (2 * a)
    print("Ligningen har to løsninger: " + str(x1) + " og " + str(x2))
```

Vi vender tilbage til if-else om lidt

Brugerinput med Fejlhåndtering

Måske brugeren taster forkert eller indtaster noget ugyldigt. Det skal vi håndtere.

```
while True:
    try:
        alder = int(input("Hvor gammel er du? "))
        break
    except ValueError:
        print("Indtast venligst et heltal")
```

Brugerinput med tal

```
while True:
    alder = input("Hvor gammel er du? ")
    if alder.isdigit():
        alder = int(alder)
        break
    else:
        print("Indtast venligst et heltal")
```

Strenge i Python

- Strenge er en sekvens af tegn
- Strenge er omgivet af anførselstegn
- Strenge kan indeholde bogstaver, tal og specialtegn
- Strenge kan indeholde et hvilket som helst tegn
- Strenge kan indeholde et hvilket som helst antal tegn
- Strenge er en datatype i Python
- Strenge er arrays af tegn

Strenge med anførselstegn

Strenge som tekst:

```
tekst = "Hej verden"
```

Strenge med tal og tekst:

```
tekst = "Hej verden 123"
```

Strengformatering

```
navn = "Alice"
alder = 30
print("Hej, jeg hedder " + navn + " og er " + str(alder) + " år gammel")
print("Hej, jeg hedder %s og er %d år gammel" % (navn, alder))
print("Hej, jeg hedder {} og er {} år gammel".format(navn, alder))
print(f"Hej, jeg hedder {navn} og er {alder} år gammel")
```

Indexing og slicing af strengen S

- S[0] er det første tegn i strengen
- S[i] - Returnerer det i'te tegn i strengen
- Negativt indeks tæller fra slutningen af strengen
- S[-2] henter det næstsidste tegn i strengen.
- S[-2]=S[len(S)-2]

Indexing og slicing af strengen S - eksempel

```
tekst = "Hej verden"
print(tekst[0])  # Resultat: H
print(tekst[1])  # Resultat: e
print(tekst[-1])  # Resultat: n
print(tekst[-2])  # Resultat: e
```

Indexing og slicing af strengen S[i:j] - eksempel

- S[i:j] - Returnerer en delstreng af S fra indeks i til j-1
- S[i:] - Returnerer en delstreng af S fra indeks i til slutningen af strengen
- S[:j] - Returnerer en delstreng af S fra starten af strengen til indeks j-1
- S[:] - Returnerer hele strengen
- S[i:j:k] - Returnerer en delstreng af S fra indeks i til j-1 med skridt k

Indexing og slicing af strengen S[i:j] - eksempel

```
tekst = "Hej verden"
print(tekst[0:3])  # Resultat: Hej
print(tekst[4:9])  # Resultat: verden
print(tekst[:3])  # Resultat: Hej
print(tekst[4:])  # Resultat: verden
print(tekst[:])  # Resultat: Hej verden
print(tekst[0:9:2])  # Resultat: Hjvn
```

Konvertering af karakterer til tal

- Vi kan konvertere karakterer til tal ved brug af ord() funktionen. Funktionen returnerer Unicode værdien af et tegn.
- Vi kan konvertere tal til karakterer ved brug af chr() funktionen. Funktionen returnerer tegnet, der svarer til Unicode værdien.
- Unicode er en international standard, der tillader tegn fra alle sprog og symboler at blive repræsenteret med en unik numerisk kode.
- Unicode værdier er tal i intervallet 0 til 1114111.

- Unicode værdier for almindelige tegn er de samme som ASCII værdierne.

Konvertering af karakterer til tal - eksempel

```
print(ord("A"))  # Resultat: 65
print(chr(65))  # Resultat: A
```

Metoder på strenge

```
besked = "Hej verden"
print(besked.upper())  # Resultat: HEJ VERDEN
print(besked.lower())  # Resultat: hej verden
print(besked.capitalize())  # Resultat: Hej verden
print(besked.replace("Hej", "Hello"))  # Resultat: Hello verden
```

Flere metoder på strenge

```
besked = "Hej verden"
print(besked.startswith("Hej"))  # Resultat: True
print(besked.endswith("Hej"))  # Resultat: False
print(besked.find("verden"))  # Resultat: 4
print(besked.find("ikke"))  # Resultat: -1
```

Brugen af split og join

```
besked = "Hej verden"
ord = besked.split(" ")
print(ord)  # Resultat: ["Hej", "verden"]
ny_besked = " ".join(ord)
print(ny_besked)  # Resultat: Hej verden
```

Brugen af strip

```
besked = "   Hej verden   "
print(besked.strip())  # Resultat: Hej verden
```

Strengformatering med %

- %s - Streng (eller enhver anden objekt)
- %d - Heltal
- %f - Decimaltal
- %e - Videnskabelig notation
- %x - Hexadecimaltal
- %c - Tegn
- %r - Repræsentation af objekt
- %% - Procenttegn
- %10s - Streng med 10 tegn
- %-10s - Streng med 10 tegn til venstre
- %10d - Heltal med 10 tegn

Strengformatering med % eksempel

```
navn = "Alice"
alder = 30
print("Hej, jeg hedder %s og er %d år gammel" % (navn, alder)) # Resultat: Hej,
jeg hedder Alice og er 30 år gammel
print("Hej, jeg hedder %10s og er %10d år gammel" % (navn, alder)) # Resultat:
Hej, jeg hedder      Alice og er         30 år gammel
print("Hej, jeg hedder %-10s og er %-10d år gammel" % (navn, alder)) # Resultat:
Hej, jeg hedder Alice      og er 30        år gammel
```

Brugen af format

```
navn = "Alice"
alder = 30
print("Hej, jeg hedder {} og er {} år gammel".format(navn, alder))
print("Hej, jeg hedder {0} og er {1} år gammel".format(navn, alder))
print("Hej, jeg hedder {navn} og er {alder} år gammel".format(navn=navn,
alder=alder))
print(f"Hej, jeg hedder {navn} og er {alder} år gammel")
```

Brugen af format med tal

```
tal = 3.14159265359
print("{:.2f}".format(tal))  # Resultat: 3.14
print("{:.4f}".format(tal))  # Resultat: 3.1416
print("{:e}".format(tal))  # Resultat: 3.141593e+00
```

```
print("{:d}".format(42))  # Resultat: 42
print("{:b}".format(42))  # Resultat: 101010
print("{:x}".format(42))  # Resultat: 2a
```

Betinget udførsel af kode i Python

- Betinget udførsel også kaldet selektion/forgrening
- Kode udføres kun hvis en eller flere betingelser er opfyldt
- Vi bruger if, elif og else til betinget udførsel af kode

Opbygning af if-sætninger

```
if betingelse:
    instruktioner
```

Betingelse skal evalueres til sand eller falsk. Instruktioner udføres kun hvis betingelsen er sand.

Opbygning af if-else-sætninger

```
if betingelse:
    instruktionerSand
else:
    instruktionerFalsk
```

Betingelse skal evalueres til sand eller falsk. InstruktionerSand udføres hvis betingelsen er sand. InstruktionerFalsk udføres hvis betingelsen er falsk.

Opbygning af if-elif-else-sætninger

```
if betingelse1:
    instruktioner1
elif betingelse2:
    instruktioner2
else:
    instruktioner3
```

Betingelse1 skal evalueres til sand eller falsk. Instruktioner1 udføres hvis betingelse1 er sand. Betingelse2 skal evalueres til sand eller falsk. Instruktioner2 udføres hvis betingelse2 er sand. Instruktioner3 udføres hvis betingelse1 og betingelse2 er falsk.

Betinget udførelse af kode

```
if alder >= 18:
    print("Du er myndig")
else:
    print("Du er ikke myndig")
```

Betinget udførelse af kode med elif

```
if alder < 0:
    print("Ugyldig alder")
elif alder < 18:
    print("Du er ikke myndig")
else:
    print("Du er myndig")
```

Betinget udførelse af kode med flere betingelser

```
if alder < 0:
    print("Ugyldig alder")
elif alder < 18:
    print("Du er ikke myndig")
elif alder < 67:
    print("Du er i arbejde")
else:
    print("Du er på pension")
```

Betinget udførsel med flere instruktioner

```
if alder < 0:
    print("Ugyldig alder")
elif alder < 18:
    print("Du er ikke myndig")
    print("Du må ikke købe alkohol")
elif alder < 67:
    print("Du er i arbejde")
    print("Du må gerne købe alkohol")
else:
    print("Du er på pension")
    print("Du må gerne købe alkohol")
```

Eksempel på if-sætninger i if-sætninger

```
if alder < 0:
    print("Ugyldig alder")
elif alder < 18:
    print("Du er ikke myndig")
    if alder < 15:
        print("Du må ikke køre bil")
    else:
        print("Du må køre knallert")
elif alder < 67:
    print("Du er i arbejde")
    print("Du må gerne køre bil")
else:
    print("Du er på pension")
    print("Du må gerne køre bil")
```

BMI beregner med if-betingelse i if-betingelse

```
højde = float(input("Hvor høj er du i meter? "))
vægt = float(input("Hvor meget vejer du i kg? "))
bmi = vægt / (højde ** 2)
if bmi < 18.5:
    print("Du er undervægtig")
elif bmi < 25:
    print("Du er normalvægtig")
elif bmi < 30:
    print("Du er overvægtig")
else:
    print("Du er svært overvægtig")
    if bmi > 40:
        print("Du er i livsfare")
```

Switch-sætninger findes ikke i Python

- Python har ikke switch-sætninger
- Vi bruger if-elif-else-sætninger i stedet
- Vi kan bruge en dictionary til at simulere en switch-sætning
- Vi kan bruge en funktion til at simulere en switch-sætning

Switch-sætning med dictionary

```
def switch(case):
    return {
        "1": "Du valgte 1",
        "2": "Du valgte 2",
        "3": "Du valgte 3"
    }.get(case, "Ugyldigt valg")
```

Switch-sætning med funktion

```
def case1():
    return "Du valgte 1"
def case2():
    return "Du valgte 2"
def case3():
    return "Du valgte 3"
def switch(case):
    return {
        "1": case1,
        "2": case2,
        "3": case3
    }.get(case, "Ugyldigt valg")()
```

Intro til løkker

- Løkker bruges til at gentage kode
- Løkker bruges til at udføre kode et bestemt antal gange
- Løkker bruges til at udføre kode indtil en betingelse er opfyldt
- Løkker bruges til at udføre kode på hvert element i en liste
- Løkker bruges til at udføre kode på hvert bogstav i en streng
- Løkker bruges til at udføre kode på hvert bogstav i en fil
- Mange andre ting
- Der findes to slags løkker: while-løkker og for-løkker

While løkkers opbygning

```
while betingelse:
    instruktioner
```

Betingelse skal evalueres til sand eller falsk. Instruktioner udføres så længe betingelsen er sand.

For løkkers opbygning

```
for variabel in sekvens:
    instruktioner
```

Variabel er en variabel der bruges til at gemme hvert element i sekvensen. Sekvens er en liste, tupel, streng eller anden sekvens. Instruktioner udføres for hvert element i sekvensen.

While-løkker

```
i = 0
while i < 10:
    print(i)
    i += 1
```

For-løkker

```
for i in range(10):
    print(i)
```

For-løkker med liste

```
farver = ["rød", "grøn", "blå"]
for farve in farver:
    print(farve)
```

For-løkker med streng

```
besked = "Hej verden"
for bogstav in besked:
    print(bogstav)
```

Løkker indeni løkker

```
for i in range(3):
    for j in range(3):
        print(i, j)
```

While løkker indeni while løkker

```
i = 0
while i < 3:
    j = 0
    while j < 3:
        print(i, j)
        j += 1
    i += 1
```

While løkker indeni for løkker

```
for i in range(3):
    j = 0
    while j < 3:
        print(i, j)
        j += 1
```

For løkker indeni while løkker

```
i = 0
while i < 3:
    for j in range(3):
        print(i, j)
    i += 1
```

Løkker med break

```
for i in range(10):
    print(i)
    if i == 5:
        break
```

```
i = 0
while i < 10:
    print(i)
    if i == 5:
        break
    i += 1
```

Løkker med continue

```
for i in range(10):
    if i == 5:
        continue
    print(i)
```

```
i = 0
while i < 10:
    i += 1
    if i == 5:
        continue
    print(i)
```

Løkker med else

```
for i in range(10):
    print(i)
else:
    print("Færdig")
```

```
i = 0
while i < 10:
    print(i)
    i += 1
else:
    print("Færdig")
```

Løkker med else og break

```
for i in range(10):
    print(i)
    if i == 5:
        break
else:
    print("Færdig")
```

break afbryder løkken og else udføres ikke.

```
i = 0
while i < 10:
    print(i)
    if i == 5:
        break
    i += 1
else:
    print("Færdig")
```

break afbryder løkken og else udføres ikke.

Løkker med else og continue

```
for i in range(10):
    if i == 5:
        continue
    print(i)
else:
    print("Færdig")
```

continue springer til næste iteration og else udføres.

```
i = 0
while i < 10:
    i += 1
    if i == 5:
        continue
    print(i)
else:
    print("Færdig")
```

continue springer til næste iteration og else udføres.

Liste comprehension i Python

- Liste comprehension er en smart måde at lave lister på
- Smart i den forstand at det er kortere og hurtigere
- Det er hurtigere fordi det er optimeret i Python
- Det er kortere fordi det er mere kompakt
- Men det er ikke altid nemmere at læse

Listecomprehension med for løkker

```
liste = [i for i in range(10)]
print(liste)  # Resultat: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Listecomprehension med for løkker og betingelse

```
liste = [i for i in range(10) if i % 2 == 0]
print(liste)  # Resultat: [0, 2, 4, 6, 8]
```

Listecomprehension med for løkker og flere løkker

```
liste = [(i, j) for i in range(3)
                for j in range(3)]
print(liste)  # Resultat: [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0),
(2, 1), (2, 2)]
```

Listecomprehension med for løkker og flere løkker og betingelse

```
liste = [(i, j) for i in range(3)                    for j in range(3) if i != j]
print(liste)  # Resultat: [(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
```

Funktioner i Python

- Funktioner er en samling af kode/instruktioenr
- Funktioner kan udføre en bestemt opgave
- Det er smart at bruge funktioner fordi det gør koden mere overskuelig

- Det er smart at bruge funktioner fordi det gør koden mere genbrugelig, mere testbar og mere fejlsikker

Funktioners opbygning

```
def navn(argumenter):
    instruktioner
```

De behøver ikke at have argumenter eller returværdier.

```
navn(argumenter)
```

Funktioner uden argumenter

```
def hilsen():
    print("Hej verden")
hilsen()  # Resultat: Hej verden
```

Funktioner med argumenter

```
def hilsen(navn):
    print("Hej " + navn)
hilsen("Alice")  # Resultat: Hej Alice
hilsen("Bob")  # Resultat: Hej Bob
```

Funktioner med flere argumenter

```
def hilsen(navn, alder):
    print("Hej " + navn + ", du er " + str(alder) + " år gammel")
hilsen("Alice", 30)  # Resultat: Hej Alice, du er 30 år gammel
hilsen("Bob", 25)  # Resultat: Hej Bob, du er 25 år gammel
```

Funktioner med returværdi

```
def hilsen(navn):
    return "Hej " + navn
hilsetekst = hilsen("Alice")
print(hilsetekst)  # Resultat: Hej Alice
```

Funktioner med returværdi og flere argumenter

```
def hilsen(navn, alder):
    return "Hej " + navn + ", du er " + str(alder) + " år gammel"
hilsentekst = hilsen("Alice", 30)
print(hilsentekst)  # Resultat: Hej Alice, du er 30 år gammel
```

Funktioner med returværdi og flere argumenter bestående af flere typer

```
def hilsen(navn, alder, er_student):
    tekst = "Hej " + navn + ", du er " + str(alder) + " år gammel"
    if er_student:
        tekst += " og er studerende"
    return tekst
hilsentekst = hilsen("Alice", 30, True)
print(hilsentekst)  # Resultat: Hej Alice, du er 30 år gammel og er studerende
```

Funktioner med for-løkker

```
def hilsen(navn, antal):
    for i in range(antal):
        print("Hej " + navn)
hilsen("Alice", 3)
```

Funktioner med while-løkker

```
def hilsen(navn, antal):
    i = 0
    while i < antal:
        print("Hej " + navn)
```

```
        i += 1
hilsen("Alice", 3)
```

Funktioner med for-løkker og returværdi

```
def hilsen(navn, antal):
    tekst = ""
    for i in range(antal):
        tekst += "Hej " + navn + "\n"
    return tekst
hilsentekst = hilsen("Alice", 3)
print(hilsentekst)
```

Funktioner med while-løkker og returværdi

```
def hilsen(navn, antal):
    tekst = ""
    i = 0
    while i < antal:
        tekst += "Hej " + navn + "\n"
        i += 1
    return tekst
hilsentekst = hilsen("Alice", 3)
print(hilsentekst)
```

Funktioner med for-løkker og break

```
def hilsen(navn, antal):
    for i in range(antal):
        print("Hej " + navn)
        if i == 1:
            break
hilsen("Alice", 3)
```

Funktioner med prædefinerede argumenter

```
def hilsen(navn="verden"):
    print("Hej " + navn)
```

```
hilsen()  # Resultat: Hej verden
hilsen("Alice")  # Resultat: Hej Alice
```

Funktioner med vilkårligt antal argumenter

```
def hilsen(*navne):
    for navn in navne:
        print("Hej " + navn)
hilsen("Alice", "Bob", "Charlie")
```

Funktioner med vilkårligt antal argumenter og prædefinerede argumenter

```
def hilsen(*navne, hilsen="Hej"):
    for navn in navne:
        print(hilsen + " " + navn)
hilsen("Alice", "Bob", "Charlie", hilsen="Goddag")
```

Funktioner med vilkårligt antal argumenter og prædefinerede argumenter og returværdi

```
def hilsen(*navne, hilsen="Hej"):
    tekst = ""
    for navn in navne:
        tekst += hilsen + " " + navn + "\n"
    return tekst
hilsentekst = hilsen("Alice", "Bob", "Charlie", hilsen="Goddag")
print(hilsentekst)
```

Filinput og filoutput i Python

- Filinput og filoutput bruges til at læse og skrive filer
- Filinput og filoutput bruges til at gemme og hente data
- Data kan være tekst, tal, billeder, lyd, video, osv.

Filinput med open()

```
fil = open("tekstfil.txt", "r")
tekst = fil.read()
print(tekst)
fil.close()
```

Filoutput med open()

```
fil = open("tekstfil.txt", "w")
fil.write("Hej verden")
fil.close()
```

Filinput med open() og with

```
with open("tekstfil.txt", "r") as fil:
    tekst = fil.read()
    print(tekst)
```

with sørger for at filen lukkes efter brug.

Filoutput med open() og with

```
with open("tekstfil.txt", "w") as fil:
    fil.write("Hej verden")
```

with sørger for at filen lukkes efter brug.

Filinput med open() og fejlhåndtering

```
try:
    fil = open("tekstfil.txt", "r")
    tekst = fil.read()
    print(tekst)
except FileNotFoundError:
    print("Filen blev ikke fundet")
finally:
    fil.close()
```

Filoutput med open() og fejlhåndtering

```
try:
    fil = open("tekstfil.txt", "w")
    fil.write("Hej verden")
except PermissionError:
    print("Du har ikke tilladelse til at skrive til filen")
finally:
    fil.close()
```

Filinput med open() og fejlhåndtering og with

```
try:
    with open("tekstfil.txt", "r") as fil:
        tekst = fil.read()
        print(tekst)
except FileNotFoundError:
    print("Filen blev ikke fundet")
```

Læsning af fil med read()

```
with open("tekstfil.txt", "r") as fil:
    tekst = fil.read()
    print(tekst)
```

Læsning af fil med readline()

```
with open("tekstfil.txt", "r") as fil:
    linje = fil.readline()
    while linje:
        print(linje, end="")
        linje = fil.readline()
```

Læsning af fil med readlines()

```
with open("tekstfil.txt", "r") as fil:
    linjer = fil.readlines()
    for linje in linjer:
        print(linje, end="")
```

Skrivning af fil med write()

```
with open("tekstfil.txt", "w") as fil:
    fil.write("Hej verden")
```

Skrivning af fil med writelines()

```
with open("tekstfil.txt", "w") as fil:
    fil.writelines(["Hej\n", "verden\n"])
```

Skrivning af fil med append()

```
with open("tekstfil.txt", "a") as fil:
    fil.write("Hej verden")
```

Skrivning af fil med append og writelines()

```
with open("tekstfil.txt", "a") as fil:
    fil.writelines(["Hej\n", "verden\n"])
```

Skrivning af fil med append og write()

```
with open("tekstfil.txt", "a") as fil:
    fil.write("Hej verden")
```

Fejlhåndtering i Python

- Fejlhåndtering bruges til at håndtere fejl
- Fejlhåndtering bruges til at undgå at programmet går i stå
- Fejlhåndtering bruges til at give brugeren en besked
- Fejlhåndtering bruges til at logge fejl
- Fejlhåndtering bruges til at håndtere uventede situationer

Fejlhåndtering med try og except

```
try:
    print(x)
except NameError:
    print("Variablen findes ikke")
```

Fejlhåndtering med try og except og flere fejl

```
try:
    print(x)
except NameError:
    print("Variablen findes ikke")
except:
    print("Noget gik galt")
```

Fejlhåndtering med try og except og flere fejl og else

```
try:
    print(x)
except NameError:
    print("Variablen findes ikke")
except:
    print("Noget gik galt")
else:
    print("Ingen fejl")
```

Her udføres else hvis der ikke er nogen fejl.

Fejlhåndtering med try og except og flere fejl og else og finally

```
try:
    print(x)
except NameError:
    print("Variablen findes ikke")
except:
    print("Noget gik galt")
else:
    print("Ingen fejl")
finally:
    print("Uanset hvad")
```

Fejlhåndtering med raise

```
x = -1
if x < 0:
    raise Exception("Tallet er mindre end 0")
```

Fejlhåndtering med assert

```
x = "Hej"
assert x == "Hej", "Teksten er ikke
```

Her udføres assert hvis betingelsen er sand ellers udføres en fejlmeddelelse.

Typer af fejl i Python

- SyntaxError: Fejl i syntaksen
- IndentationError: Fejl i indrykning
- NameError: Variabel findes ikke
- TypeError: Forkert type
- ValueError: Forkert værdi
- ZeroDivisionError: Division med nul
- FileNotFoundError: Filen findes ikke
- PermissionError: Manglende tilladelse
- osv.

Typer af exception i Python

- Exception: Generel fejl
- AssertionError: Fejl i assert

- AttributeError: Attribut findes ikke
- ImportError: Modul findes ikke
- IndexError: Indeks findes ikke
- KeyError: Nøgle findes ikke
- KeyboardInterrupt: Bruger afbryder program
- osv.

Indlæsning af csv-fil med fejlhåndtering

```
import csv
try:
    with open("data.csv", "r") as fil:
        data = csv.reader(fil)
        for række in data:
            print(række)
except FileNotFoundError:
    print("Filen blev ikke fundet")
```

Skrivning af csv-fil med fejlhåndtering

```
import csv
try:
    with open("data.csv", "w") as fil:
        data = csv.writer(fil)
        data.writerow(["Navn", "Alder"])
        data.writerow(["Alice", 30])
        data.writerow(["Bob", 25])
except PermissionError:
    print("Du har ikke tilladelse til at skrive til filen")
```

Moduler i Python

- Moduler er en samling af funktioner, klasser og variabler.
- Moduler bruges til at organisere kode.
- Moduler bruges til at genbruge kode.

Import af moduler

```
import math
print(math.pi)  # Resultat: 3.141592653589793
```

Import af moduler med alias

```
import math as m
print(m.pi)  # Resultat: 3.141592653589793
```

Import af moduler med from

```
from math import pi
print(pi)  # Resultat: 3.141592653589793
```

Med from kan vi importere en specifik funktion, klasse eller variabel.

Import af moduler med from og alias

```
from math import pi as p
print(p)  # Resultat: 3.141592653589793
```

Med from kan vi importere en specifik funktion, klasse eller variabel og give den et alias.

Import af moduler med from og *

```
from math import *
print(pi)  # Resultat: 3.141592653589793
```

Med from kan vi importere alle funktioner, klasser og variabler. Og * betyder alle.

Import af moduler med sys.path

```
import sys
print(sys.path)
```

sys.path er en liste af stier hvor Python leder efter moduler.

Import af moduler med sys.path.append()

```
import sys
sys.path.append("C:/bruger/brugernavn")
```

sys.path.append() tilføjer en sti til sys.path.

Import af moduler med sys.path.remove()

```
import sys
sys.path.remove("C:/bruger/brugernavn")
```

sys.path.remove() fjerner en sti fra sys.path.

Import af moduler med sys.path.insert()

```
import sys
sys.path.insert(0, "C:/bruger/brugernavn")
```

sys.path.insert() indsætter en sti i sys.path.

Lav dine egne moduler

```
# minmodul.py
def hilsen(navn):
    print("Hej " + navn)
```

```
import minmodul
minmodul.hilsen("Alice")  # Resultat: Hej Alice
```

Lav dine egne moduler med alias

```
# minmodul.py
def hilsen(navn):
    print("Hej " + navn)
```

```
import minmodul as m
m.hilsen("Alice")  # Resultat: Hej Alice
```

Lav dine egne moduler med from

```
# minmodul.py
def hilsen(navn):
    print("Hej " + navn)
```

```
from minmodul import hilsen
hilsen("Alice")  # Resultat: Hej Alice
```

Lav dine egne moduler med from og alias

```
# minmodul.py
def hilsen(navn):
    print("Hej " + navn)
```

```
from minmodul import hilsen as h
h("Alice")  # Resultat: Hej Alice
```

Lav dine egne moduler med from og *

```
# minmodul.py
def hilsen(navn):
    print("Hej " + navn)
```

```
from minmodul import *
hilsen("Alice")  # Resultat: Hej Alice
```

Zip funktionen i Python

- Den generelle syntaks for zip funktionen er: zip(liste1, liste2, ...)
- Bruges til at kombinere to eller flere lister.
- Lave en liste af tupler, liste af lister, liste af sæt eller liste af dictionaries.
-

Zip funktionen med to lister

```
navne = ["Alice", "Bob", "Charlie"]
alder = [30, 25, 35]
personer = zip(navne, alder)
print(list(personer))  # Resultat: [('Alice', 30), ('Bob', 25), ('Charlie', 35)]
```

Zip funktionen med tre lister

```
navne = ["Alice", "Bob", "Charlie"]
alder = [30, 25, 35]
køn = ["Kvinde", "Mand", "Mand"]
personer = zip(navne, alder, køn)
print(list(personer))  # Resultat: [('Alice', 30, 'Kvinde'), ('Bob', 25, 'Mand'),
('Charlie', 35, 'Mand')]
```

Zip funktionen med to lister og for-løkke

```
navne = ["Alice", "Bob", "Charlie"]
alder = [30, 25, 35]
for navn, alder in zip(navne, alder):
    print(navn, alder)
```

Zip funktionen med to lister og for-løkke og returværdi

```
navne = ["Alice", "Bob", "Charlie"]
alder = [30, 25, 35]
personer = zip(navne, alder)
for person in personer:
    print(person[0], person[1])
```

Enumerate funktionen i Python

- Enumerate funktionen bruges til at tilføje en tæller til en liste.
- Enumerate funktionen bruges til at lave en liste af tupler med tæller og element.
- Enumerate funktionen bruges til at lave en liste af tupler med tæller og element i en for-løkke.
- Enumerate funktionen bruges til at lave en liste af tupler med tæller og element i en liste.
- Den generelle syntaks for enumerate funktionen er: enumerate(liste, start=0)

Enumerate funktionen med en liste

```
navne = ["Alice", "Bob", "Charlie"]
personer = enumerate(navne)
print(list(personer))  # Resultat: [(0, 'Alice'), (1, 'Bob'), (2, 'Charlie')]
```

Enumerate funktionen med en liste og for-løkke

```
navne = ["Alice", "Bob", "Charlie"]
for i, navn in enumerate(navne):
    print(i, navn)
```

Enumerate funktionen med en liste og for-løkke og returværdi

```
navne = ["Alice", "Bob", "Charlie"]
personer = enumerate(navne)
for person in personer:
    print(person[0], person[1])
```

Enumerate funktionen med en liste og for-løkke og returværdi og startværdi

```
navne = ["Alice", "Bob", "Charlie"]
personer = enumerate(navne, start=1)
for person in personer:
    print(person[0], person[1])
```

Højere ordens funktioner i Python

- Højere ordens funktioner er funktioner der tager en eller flere funktioner som argumenter og/eller returnerer en funktion.
- Højere ordens funktioner er en del af funktionel programmering.
- Højere ordens funktioner bruges til at gøre koden mere generisk og mere genbrugelig.
- Højere ordens funktioner bruges til at gøre koden mere kort og mere læsbar.
- Højere ordens funktioner bruges til at gøre koden mere fleksibel og mere testbar.

Funktioner som argumenter

```
def hilsen(navn):
    return "Hej " + navn
def højere_ordens_funktion(funktion, navn):
    return funktion(navn)
tekst = højere_ordens_funktion(hilsen, "Alice")
print(tekst)  # Resultat: Hej Alice
```

Indre funktioner med returværdi som funktion

```
def højere_ordens_funktion():
    def hilsen(navn):
        return "Hej " + navn
    return hilsen
funktion = højere_ordens_funktion()
tekst = funktion("Alice")
print(tekst)  # Resultat: Hej Alice
```

Map funktionen en højere ordens funktion

- Map funktionen bruges til at anvende en funktion på hvert element i en liste.
- Map funktionen bruges til at anvende en funktion på hvert element i flere lister.
- Map funktionen bruges til at anvende en funktion på hvert element i en liste og en anden liste.
- og meget mere...
- Den generelle syntaks for map funktionen er: map(funktion, liste)

Map funktionen med en liste

```
def hilsen(navn):
    return "Hej " + navn
```

```
navne = ["Alice", "Bob", "Charlie"]
tekster = map(hilsen, navne)
print(list(tekster))  # Resultat: ['Hej Alice', 'Hej Bob', 'Hej Charlie']
```

Map funktionen med flere lister

```
def hilsen(navn, alder):
    return "Hej " + navn + ", du er " + str(alder) + " år gammel"
navne = ["Alice", "Bob", "Charlie"]
alder = [30, 25, 35]
tekster = map(hilsen, navne, alder)
print(list(tekster))  # Resultat: ['Hej Alice, du er 30 år gammel', 'Hej Bob, du
er 25 år gammel', 'Hej Charlie, du er 35 år gammel']
```

Map funktionen med flere lister og en funktion

```
def hilsen(navn, alder):
    return "Hej " + navn + ", du er " + str(alder) + " år gammel"
def højere_ordens_funktion(funktion, liste1, liste2):
    return map(funktion, liste1, liste2)
navne = ["Alice", "Bob", "Charlie"]
alder = [30, 25, 35]
tekster = højere_ordens_funktion(hilsen, navne, alder)
print(list(tekster))  # Resultat: ['Hej Alice, du er 30 år gammel', 'Hej Bob, du
er 25 år gammel', 'Hej Charlie, du er 35 år gammel']
```

Filter funktionen en højere ordens funktion

- Filter funktionen bruges til at filtrere elementer i en liste.
- Den generelle syntaks for filter funktionen er: filter(funktion, liste)
- Funktionen skal returnere sand eller falsk.
- Filter funktionen returnerer en iterator.
- Filter funktionen bruges ofte sammen med lambda funktioner, som er anonyme funktioner.

Filter funktionen med en liste

```
def er_over_18(alder):
    return alder >= 18
alder = [30, 15, 25, 40, 10]
```

```
resultat = filter(er_over_18, alder)
print(list(resultat))  # Resultat: [30, 25, 40]
```

Lambda funktioner i Python

- Lambda funktioner er anonyme funktioner. Dvs. funktioner uden navn.
- Lambda funktioner bruges til at lave enkle funktioner.
- Lambda funktioner bruges ofte sammen med højere ordens funktioner.
- Lambda funktioner bruges ofte sammen med map, filter og sort funktioner.
- Den generelle syntaks for lambda funktioner er: lambda argumenter: udtryk
- Lambda funktioner kan have et eller flere argumenter.

Lambda funktioner med et argument

```
f = lambda x: x + 10
print(f(5))  # Resultat: 15
```

Lambda funktioner med flere argumenter

```
f = lambda x, y: x * y
print(f(5, 10))  # Resultat: 50
```

Lambda funktioner med map funktionen

```
navne = ["Alice", "Bob", "Charlie"]
tekster = map(lambda navn: "Hej " + navn, navne)
print(list(tekster))  # Resultat: ['Hej Alice', 'Hej Bob', 'Hej Charlie']
```

Lambda funktioner med filter funktionen

```
alder = [30, 15, 25, 40, 10]
resultat = filter(lambda alder: alder >= 18, alder)
print(list(resultat))  # Resultat: [30, 25, 40]
```

Lambda funktioner med filter funktionen og map funktionen

```
alder = [30, 15, 25, 40, 10]
resultat = map(lambda alder: alder + 10, filter(lambda alder: alder >= 18, alder))
print(list(resultat))  # Resultat: [40, 35, 50]
```

Her filtreres alderen først og derefter lægges 10 til alderen. Dvs. at alderen tjekkes først om den er over 18 og derefter lægges 10 til alderen.

Reduce funktionen i Python

- Reduce funktionen bruges til at reducere en liste til et enkelt element.
- Reduce funktionen bruges til at anvende en funktion på hvert element i en liste og akkumulere resultatet.
- Reduce funktionen bruges til at anvende en funktion på hvert element i en liste og akkumulere resultatet med en startværdi.
- Reduce funktionen bruges til at anvende en funktion på hvert element i en liste og akkumulere resultatet med en startværdi og en anden funktion.
- Syntax: reduce(funktion, liste, startværdi)

Reduce funktionen med en liste

```
from functools import reduce
def add(x, y):
    return x + y
tal = [1, 2, 3, 4, 5]
resultat = reduce(add, tal)
print(resultat)  # Resultat: 15
```

Reduce funktionen med en liste og startværdi

```
from functools import reduce
def add(x, y):
    return x + y
tal = [1, 2, 3, 4, 5]
resultat = reduce(add, tal, 10)
print(resultat)  # Resultat: 25
```

Reduce funktionen med en liste og startværdi og en anden funktion

```
from functools import reduce
def add(x, y):
    return x + y
def sub(x, y):
    return x - y
tal = [1, 2, 3, 4, 5]
resultat = reduce(add, tal, 10)
print(resultat)  # Resultat: 25
resultat = reduce(sub, tal, 10)
print(resultat)  # Resultat: -13
```

Imperativ programmering vs. deklarativ programmering

- Imperativ programmering som Python bruger er en programmeringsparadigme hvor programmereren beskriver hvordan programmet skal udføre opgaven.
- Deklarativ programmering er en programmeringsparadigme hvor programmereren beskriver hvad programmet skal gøre.
- Imperativ programmering bruger for-løkker, while-løkker, if-else, osv.
- Deklarativ programmering bruger map, filter, reduce, osv.

Imperativ programmering med for-løkker

```
tal = [1, 2, 3, 4, 5]
resultat = []
for x in tal:
    resultat.append(x * 2)
print(resultat)  # Resultat: [2, 4, 6, 8, 10]
```

Her bruger vi en for-løkke til at gange hvert element i en liste med 2.

Deklarativ programmering med map funktionen

```
tal = [1, 2, 3, 4, 5]
resultat = map(lambda x: x * 2, tal)
print(list(resultat))  # Resultat: [2, 4, 6, 8, 10]
```

Her bruger vi map funktionen til at gange hvert element i en liste med 2. lambda funktionen er en anonym funktion som tager et argument x og returnerer x * 2. Denne funktion bruges som argument til map funktionen.

Højere ordens funktioner i Python

- map er eksempel på en højere ordens funktion. Dvs. en funktion der tager en anden funktion som argument.
- Typisk er Højere ordens funktioner der tager en eller flere funktioner som argumenter og/eller returnerer en funktion.
- De er en del af funktionel programmering.
- De bruges til at gøre koden mere generisk og mere genbrugelig.
- Koden mere kort og mere læsbar.
- Højere ordens funktioner bruges til at gøre koden mere fleksibel og mere testbar.

Eksempler på højere ordens funktioner: zip

Zip-funktionen er en højere ordens funktion, fordi den tager to eller flere lister som argumenter og returnerer en liste af tupler.

```
navne = ["Alice", "Bob", "Charlie"]
alder = [30, 25, 35]
personer = zip(navne, alder)
print(list(personer))  # Resultat: [('Alice', 30), ('Bob', 25), ('Charlie', 35)]
```

Eksempler på højere ordens funktioner: enumerate

Enumerate-funktionen er en højere ordens funktion, fordi den tager en liste som argument og returnerer en liste af tupler med tæller og element.

```
navne = ["Alice", "Bob", "Charlie"]
personer = enumerate(navne)
print(list(personer))  # Resultat: [(0, 'Alice'), (1, 'Bob'), (2, 'Charlie')]
```

Eksempler på højere ordens funktioner: filter

Filter-funktionen er en højere ordens funktion, fordi den tager en funktion som argument og returnerer en iterator.

```
alder = [30, 15, 25, 40, 10]
resultat = filter(lambda alder: alder >= 18, alder)
```

```
print(list(resultat))  # Resultat: [30, 25, 40]
```

Eksempler på højere ordens funktioner: map

Map-funktionen er en højere ordens funktion, fordi den tager en funktion som argument og returnerer en iterator.

```
navne = ["Alice", "Bob", "Charlie"]
tekster = map(lambda navn: "Hej " + navn, navne)
print(list(tekster))  # Resultat: ['Hej Alice', 'Hej Bob', 'Hej Charlie']
```

Eksempler på højere ordens funktioner: reduce

Reduce-funktionen er en højere ordens funktion, fordi den tager en funktion som argument og returnerer en værdi.

```
from functools import reduce
tal = [1, 2, 3, 4, 5]
resultat = reduce(lambda x, y: x + y, tal)
print(resultat)  # Resultat: 15
```

Reduce-funktionen bruges til at reducere en liste til et enkelt element.

Anonyme funktioner med lambda og map funktionen

```
navne = ["Alice", "Bob", "Charlie"]
tekster = map(lambda navn: "Hej " + navn, navne)
print(list(tekster))  # Resultat: ['Hej Alice', 'Hej Bob', 'Hej Charlie']
```

Anonyme funktioner med lambda og filter funktionen

```
alder = [30, 15, 25, 40, 10]
resultat = filter(lambda alder: alder >= 18, alder)
print(list(resultat))  # Resultat: [30, 25, 40]
```

Anonyme funktioner med lambda og reduce funktionen

```
from functools import reduce
tal = [1, 2, 3, 4, 5]
resultat = reduce(lambda x, y: x + y, tal)
print(resultat)  # Resultat: 15
```

Decorators i Python

- Decorators bruges til at ændre en funktion uden at ændre dens kode.
- Decorators bruges til at tilføje funktionalitet til en eksisterende funktion.
- Decorators bruges til at gøre koden mere generisk og mere genbrugelig.
- Decorators bruges til at gøre koden mere kort og mere læsbar.

Decorators med en funktion

```
def dekoration(funktion):
    def wrapper():
        print("Før")
        funktion()
        print("Efter")
    return wrapper
def hilsen():
    print("Hej verden")
hilsen = dekoration(hilsen)
hilsen()
```

Resultat: Før Hej verden Efter. Hvad der sker her er at vi laver en funktion der tager en anden funktion som argument og returnerer en ny funktion. Den nye funktion udfører noget før og efter den gamle funktion.

Decorators med en funktion og @

```
def dekoration(funktion):
    def wrapper():
        print("Før")
        funktion()
        print("Efter")
    return wrapper
@dekoration
def hilsen():
```

```
    print("Hej verden")
hilsen()
```

Resultat: Før Hej verden Efter.