Variable

En **variabel** i Python er en enhed, der har en værdi – mere præcist er en variabel et navn, der peger på en værdi af data

Eksempler på variable og tildelinger

```
alder = 32
fornavn = 'Peter'
efternavn = 'Sørensen'
navn = fornavn + ' ' + efternavn
prisMedMoms = prisUdenMoms * 1.25
```

Ovenfor ses fem forskellige tildelinger (= assignments), hvor

- variablen `alder` tildeles værdien 32
- variablen `fornavn` tildeles værdien 'Peter'
- variablen `efternavn` tildeles værdien 'Sørensen'
- variablen navn tildeles værdien af `fornavn + ' ' + efternavn`, hvilket er 'Peter Sørensen'
- variablen `prisMedMoms` tildeles værdien af `prisUdenMoms * 1.25`, hvilket kræver, at variablen `prisUdenMoms` tidligere i programmet har fået tildelt en værdi, hvilket ikke fremgår af ovenstående

Generelt gælder der om tildelinger

```
variabel = udtryk
```

at udtrykket til højre for lighedstegnet først skal evalueres, dvs at værdien heraf skal bestemmes, hvorefter variablen tildeles denne værdi og bliver dermed en pegepind til den pågældende værdi

Der gælder i Python, at variable ikke på forhånd skal typeerklæres, som tilfældet er i andre programmeringssprog. Variabelnavne skal starte med enten en underscore _ eller et bogstav og herefter kan følge et vilkårligt antal underscores, bogstaver eller tal. Der er forskel på store og små bogstaver, og vi siger, at Python er case-sensitive

Det er vigtigt med velvalgte navne til konkrete variable, og hvis der er flere ord i navnet som f.eks. i prisMedMoms kan man vælge mellem tre forskellige notationer

- `prisMedMoms` som kaldes Camel Case
- `pris_med_moms` som kaldes Snake Case
- `PrisMedMoms` som kaldes Pascal Case

Bemærk, at notationen

```
pris med moms
```

ikke er tilladt, da mellemrum ikke tillades

Værdien af en variabel udskrives typisk i Python med en print-sætning

```
fornavn = 'Peter'
efternavn = 'Sørensen'
navn = fornavn + ' ' + efternavn
print(navn)
```

```
Peter Sørensen
```

Datatyper

Der er forskellige indbyggede datatyper i Python, og her vil vi omtale de vigtigste

Tal

Heltal kaldes for `integers` og som type i Python `int`, og decimaltal kaldes for `floats` og som type `float`

```
alder = 32
karaktersnit = 8.1
befolkningstalDK = 5.98e6
type(alder), type(karaktersnit), type(befolkningstalDK)
```

```
(int, float, float)
```

Ovenfor lægger vi mærke til, at variablen `alder` automatisk får typen `int`, mens variablene

`karaktersnit`
`befolkningstalDK`

får typen `float`
Decimaltal skrives med punktum og ikke komma, og 10'er potenser skrives med e (eller E)

Strenge

Strenge kaldes for strings og `str` som type i Python, og strenge er omgivet af anførselstegn – enten enkelte eller dobbelte

Tegnene i en streng kan tilgås via nulindekseringen, hvilket betyder, at første tegn i strengen har indeks 0, andet tegn i strengen har indeks 1 osv.

```
sport = 'gymnastik'
print(sport[0])
print(sport[5])
print(sport[8])
print(sport[-1])
```

```
g
s
k
k
```

Som det også ses ovenfor, er det tilladt med negativ indeksering, hvor indekset -1 giver det sidste tegn i strengen, og indekset -2 giver det næstsidste tegn i strengen osv.

Python har i modsætning til andre programmeringssprog ikke en speciel type for et tegn (= char) – i stedet opfattes et tegn i Python som en streng bestående af kun ét tegn

Det er muligt at beskære strenge på følgende måde

```
tekst = 'Programmering er spændende'

tekst[3:7]
```

```
'gram'
```

```
tekst[17:]
```

```
'spændende'
```

```
tekst[:13]
```

```
'Programmering'
```

Nedenfor ses, hvad der sker, når to strenge lægges sammen (sammenkædes eller konkateneres), og når en streng ganges med et helt positivt tal

```
'fod'+'bold'
```

```
'fodbold'
```

```
3*'Ho'
```

```
'HoHoHo'
```

Der er en række indbyggede metoder, der er knyttet til strenge, og nogle af de vigtigste er vist nedenfor

Bogstaverne i en streng kan skrives udelukkende med store bogstaver eller med små bogstaver:

```
fag = 'Informatik'
fag.upper()
```

```
'INFORMATIK'
```

```
fag.lower()
```

```
'informatik'
```

Det kan undersøges, om en given streng er en del af hele strengen, og hvis det er tilfældet, returneres indekset, hvor delstrengen starter, eller returneres –1

```
fag.find('form')
```

```
2
```

```
fag.find('on')
```

```
-1
```

Antallet af forekomster af delstrenge kan tælles

```
fag.count('i')
```

```
1
```

Læg mærke til ovenfor, at der i strengen `'Informatik'` kun er et `'i'`, som er det andet sidste tegn, mens det første tegn er `'I'`, og der er forskel på store og små bogstaver – Python er case-sensitive

Dele af en streng kan erstattes af en ny streng

```
fag.replace('Infor', 'Mate')
```

```
'Matematik'
```

En vigtig pointe ved metoder, der hører til strenge, er, at de oprindelige strenge ikke er ændret. Ovenfor har variablen fag hele tiden værdien 'Informatik'.

Strenge i Python siges at være uforanderlige (på engelsk immutable), men derfor kan variablen fag godt forandres

```
fag = fag.replace('Infor', 'Mate')
print(fag)
```

```
Matematik
```

Det er vigtigt med brugervenlige udskrifter, og her giver Python fine muligheder med f-strenge og escape-tegn

```
kondital = (2800 - 504.9)/44.73
vurdering = 'højt'

print(f'Konditallet er {kondital:.1f} som vurderes til at være {vurdering.upper()}')
```

```
Konditallet er 51.3 som vurderes til at være HØJT
```

Ovenfor ses brugen af en f-streng, hvor bogstavet `f` står foran en streng, som kan indeholde variabler i krøllede parenteser. Variablene kan indgå i udtryk, der skal evalueres inden udskriften

Nedenfor ses eksempler på, hvordan escape-tegn kan bruges i udskrifter

```
print('Pizzariaet kaldes John\'s bedste pizzaer - kom gerne forbi!')
```

```
Pizzariaet kaldes John's bedste pizzaer - kom gerne forbi!
```

Escape-tegnet backslash \ betyder ovenfor, at det efterfølgende anførselstegn skal udskrives og dermed ikke opfattes som et anførselstegn, der hænger sammen med det første anførselstegn foran `Pizzariaet`…

Backslash kan også bruges til at gennemtvinge en ny linje i en udskrift

```
print('Pizzariaet kaldes John\'s bedste pizzaer\n- kom gerne forbi!')
```

```
Pizzariaet kaldes John's bedste pizzaer
- kom gerne forbi!
```

Der kan laves passende tabuleringer i udskrifter med backslash t

```
print('Ida \t15 points')
print('Peter \t12 points')
print('Ole \t11 points')
print('Marie \t10 points')
```

```
Ida     15 points
Peter   12 points
Ole     11 points
Marie   10 points
```

Escape-tegn og f-strenge kan desuden kombineres

```
produkt = 'Kaffe'
pris = 56
print(f'Produkt:\t{produkt}\nPris:\t\t{pris} kr\nStatus:\t\t\'Tilgængelig\'')
```

```
Produkt:        Kaffe
Pris:           56 kr
Status:         'Tilgængelig'
```

Samlinger

Der er fire indbyggede datatyper i Python til opbevaring af samlinger af data

- **liste**
- **tuple**
- **dictionary**
- **mængde**

De har alle nogle kvaliteter i forskellige sammenhænge

Lister

Datatypen **liste** er en følge af endeligt mange elementer i en bestemt rækkefølge

```
['Æble', 'Banan', 'Appelsin']

[1, 3, 7, 12, 19]

[2, 'bold', 17, True, 5, [2, 4, 6]]
```

Vi bemærker de firkantede parenteser, der omkranser listen samt kommaerne, der adskiller de enkelte elementer

I de tre ovenstående eksempler er den første en liste bestående af strenge, mens den anden er en liste af tal. Det tredje eksempel viser, at det er muligt i Python at lave en liste af elementer med forskellige datatyper, selvom det i de færreste tilfælde vil være en hensigtsmæssig ide

En vigtig egenskab ved lister er, at de er indekseret, hvorved man kan tilgå de enkelte elementer i en list med indeks. Der er tale om nul-indeksering, hvilket betyder, at første element i en liste har indeks 0, andet element har indeks 1 og så videre

```
Frugter = ['Æble', 'Banan', 'Appelsin']
Frugter[0]
```

```
'Æble'
```

```
Frugter[2]
```

```
'Appelsin'
```

En typisk fejl ved lister er indeks-fejl

```
Frugter[3]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[7], line 1
----> 1 Frugter[3]

IndexError: list index out of range
```

Python understøtter negativ indeksering, idet indekset –1 refererer til det sidste element i listen, –2 det næstsidste element i listen og så videre

```
Frugter[-1]
```

```
'Appelsin'
```

```
Frugter[-2]
```

```
'Banan'
```

I programmering er der ofte behov for at bruge lister bestående af heltallene fra 0 og op til et bestemt tal, og i den sammenhæng bruger Python den indbyggede funktion `range()`

```
range(8)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

Vi lægger mærke til, at `range(8)` giver listen, der starter med 0 og slutter med 7 og ikke 8. Til gengæld har listen præcist 8 elementer

Vi kan tvinge `range()` til at starte med et andet tal end 0 ved eksplicit at skrive start-tallet

```
range(3, 9)
```

```
[3, 4, 5, 6, 7, 8]
```

Her ser vi, at listen starter ved 3 som angivet i `range(3, 9)`, og at listen slutter ved 8 og ikke 9

I de to eksempler på `range()` har efterfølgerne været 1 større, men det kan vi ændre på med en tredje parameter

```
range(1, 12, 2)
```

```
[1, 3, 5, 7, 9, 11]
```

Lister kan beskæres på samme måde som strenge, som vi tidligere har set

```
venner = ['Anne', 'Peter', 'Ole', 'Ida', 'Eva', 'Søren', 'Mette']
venner[1:]
```

```
['Peter', 'Ole', 'Ida', 'Eva', 'Søren', 'Mette']
```

```
venner[2:6]
```

```
['Ole', 'Ida', 'Eva', 'Søren']
```

```
venner[:4]
```

```
['Anne', 'Peter', 'Ole', 'Ida']
```

Det er vigtigt at pointere, at den oprindelige liste ikke ændres ved beskæring, men til gengæld kan lister ændres på følgende måde

```
venner[3] = 'Signe'
venner
```

```
['Anne', 'Peter', 'Ole', 'Signe', 'Eva', 'Søren', 'Mette']
```

```
venner.append('Bo')
venner
```

```
['Anne', 'Peter', 'Ole', 'Signe', 'Eva', 'Søren', 'Mette', 'Bo']
```

```
venner.remove('Ole')
venner
```

```
['Anne', 'Peter', 'Signe', 'Eva', 'Søren', 'Mette', 'Bo']
```

I de to sidste eksempler er metoderne `append()` og `remove()` anvendt, og der er en række metoder knyttet til lister, hvor de vigtigste fremgår af følgende

Metode	Beskrivelse
`append()`	Tilføjer et element i slutningen af listen
`clear()`	Fjerner alle elementer fra listen
`copy()`	Returnerer en kopi af listen
`count()`	Returnerer antallet af elementer med den specificerede værdi
`extend()`	Tilføjer elementer fra en liste til slutningen af listen
`index()`	Returnerer indekset af det første element med den specificerede værdi
`insert()`	Tilføjer et element i den specificerede position
`pop()`	Fjerner elementet i den specificerede position
`remove()`	Fjerne elementet med den specificerede værdi
`sort()`	Sorterer listen

se desuden https://www.w3schools.com/python/python_lists_methods.asp

En vigtig egenskab ved lister er, at lister kan gennemløbes med en `for`–løkke

```
venner = ['Signe', 'Peter', 'Eva']

for ven in venner:
    print(ven)
```

```
Signe
Peter
Eva
```

List comprehension er en kort og elegant måde at oprette lister på i Python

I stedet for at bruge en for-løkke og append, kan man skrive det mere kompakt, og for de erfarne programmører er det mere læsbart og hurtigere at skrive

```
a = [1, 2, 3, 4, 5]

resultat = []
for tal in a:
    resultat.append(tal * 2)

resultat
```

```
[2, 4, 6, 8, 10]
```

Ovenstående kan erstattes af en list comprehension

```
a = [1, 2, 3, 4, 5]

resultat = [tal * 2 for tal in a]

resultat
```

```
[2, 4, 6, 8, 10]
```

```
venner = ['Ida', 'Peter', 'Signe', 'Eva', 'Agnes']

rigtigeVenner = []
for ven in venner:
    if 'a' in ven:
        rigtigeVenner.append(ven.upper())

rigtigeVenner
```

```
['IDA', 'EVA']
```

Ovenstående kan erstattes af

```
venner = ['Ida', 'Peter', 'Signe', 'Eva', 'Agnes']

rigtigeVenner = [ven.upper() for ven in venner if 'a' in ven ]

rigtigeVenner
```

```
['IDA', 'EVA']
```

Generelt er list comprehension opbygget på følgende måde

```
nyListe = [udtryk for element in liste if betingelse]
```

som erstatning for

```
nyListe = []
for element in liste:
    if betingelse:
        nyListe.append(udtryk)
```

Lister kan sammenkædes på samme måde som strenge

```
[2, 3, 7] + [1, 8, 4]
```

```
[2, 3, 7, 1, 8, 4]
```

```
3 * [5, 1]
```

```
[5, 1, 5, 1, 5, 1]
```

Tupler

Datatypen **tuple** er ligesom datatypen lister en følge af endeligt mange elementer i en bestemt rækkefølge, men der er forskelle

```
('Æble', 'Banan', 'Appelsin')

(1, 3, 7, 12, 19)

(2, 'bold', 17, True, 5, (2, 4, 6))
```

Vi ser, at tupler skrives med almindelige parenteser, hvor lister skrives med firkantede parenteser

I modsætning til eksisterende lister kan tupler ikke ændres

```
venner = ('Anne', 'Peter', 'Ole', 'Ida', 'Eva', 'Søren', 'Mette')
venner[3] = 'Signe'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[8], line 2
      1 venner = ('Anne', 'Peter', 'Ole', 'Ida', 'Eva', 'Søren', 'Mette')
----> 2 venner[3] = 'Signe'

TypeError: 'tuple' object does not support item assignment
```

Samtidig er de fleste af metoderne, der hører til lister, ikke mulige ved tupler. Tupler kan ikke forandres, mens lister godt kan, og vi siger, at lister er foranderlige (= mutable), og tupler er uforanderlige (= immutable)

Det kan være hensigtsmæssigt at bruge datatypen tuple i stedet for liste, når man vil undgå utilsigtede ændringer

Dictionaries

Datatypen **dictionary** er en samling af endeligt mange par af nøgler og værdier

```
{'forret': 'suppe', 'hovedret': 'steg', 'dessert': 'is'}
{'en': 57, 'mand': 12, 'løber': 3}
{'bord': 'table', 'hvid': 'white', 'bold': 'ball'}
```

Dictionaries skrives med krøllede parenteser, hvor de enkelte par af nøgler og værdier er adskilt af kommaer, og hvor der er kolon mellem nøgle og værdi

Det kan være mere overskueligt at skrive et dictionary over flere linjer, hvilket Python tillader

```
retter = {
    'forret': 'suppe',
    'hovedret': 'steg',
    'dessert': 'is'
}
```

Et dictionary er en generaliseret liste, idet nøglerne til et dictionary kan vælges som tallene 0, 1, 2, 3 og 4 svarende til indeks i en liste

```
venneListe = ['Ida', 'Peter', 'Tea', 'Morten', 'Eva']
venneDictionary = {0: 'Ida', 1: 'Peter', 2: 'Tea', 3: 'Morten', 4: 'Eva'}
```

Ligesom ved lister er det en vigtig egenskab ved dictionaries, at de kan gennemløbes ved en `for`–løkke

```
danskEngelsk = {'bord': 'table', 'bold': 'ball', 'stol': 'chair'}
for x in danskEngelsk:
    print(x, end = ' ')
```

```
bord bold stol
```

Vi bemærker, at `for`–løkken

- `for element in dict:`

gennemløber nøglerne ved, at `element` refererer til nøglerne, og hvis vi skal referere til værdierne, kan det klares i udskriften

```
for x in danskEngelsk:
    print(danskEngelsk[x], end = ' ')
```

```
table ball chair
```

Det er dog mere hensigtsmæssigt at gennemløbe indholdet i et dictionary på følgende generelle måde

```
for noegle, vaerdi in dictet.items():
```

og i vores eksempel

```
for x, y in danskEngelsk.items():
    print(x, y)
```

```
bord table
bold ball
stol chair
```

hvor vi har gjort brug af metoden `items()` hørende til dictionaries

De enkelte værdier i dictionary kan bestemmes

```
print(danskEngelsk['stol'])
```

```
chair
```

og nyt indhold i et dictionary kan tilføjes

```
danskEngelsk['hvid'] = 'white'
danskEngelsk
```

```
{'bord': 'table', 'bold': 'ball', 'stol': 'chair', 'hvid': 'white'}
```

og eksisterende kan fjernes med metoden `pop()`

```
danskEngelsk.pop('bold')
danskEngelsk
```

```
{'bord': 'table', 'stol': 'chair', 'hvid': 'white'}
```

Der er forskellige metoder tilknyttet dictionaries

Metode	Beskrivelse
`clear()`	Fjerner alle elementer fra dictionary
`copy()`	Returnerer en kopi af dictionary
`items()`	Returnerer en liste af nøgle og værdi par
`keys()`	Returnerer en liste af nøgler i dictionary
`pop()`	Fjerner elementet med den specificerede nøgle
`values()`	Returnerer en liste af værdier i dictionary

se desuden https://www.w3schools.com/python/python_dictionaries_methods.asp

Mængder

En mængde er en samling af endeligt mange forskellige elementer

```
{'Æble', 'Banan', 'Appelsin'}

{1, 3, 7, 12, 19}

{2, 'bold', 17, True, 5, (2, 4, 6)}
```

Mængder skrives med krøllede parenteser, hvor de forskellige elementer er adskilt af kommaer, og mængder er ikke indekseret, selvom de er skrevet op i en rækkefølge

```
venner = {'Ole', 'Anne', 'Peter', 'Ida'}
venner[1]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[1], line 2
      1 venner = {'Ole', 'Anne', 'Peter', 'Ida'}
----> 2 venner[1]

TypeError: 'set' object is not subscriptable
```

Mængder kan gennemløbes ved en `for`-løkke, men rækkefølgen er ikke gennemskuelig

```
for ven in venner:
    print(ven)
```

```
Ole
Peter
Anne
Ida
```

Elementer i en mængde kan fjernes og tilføjes med metoderne `remove()` og `add()`

```
venner.remove('Ida')
venner.add('Lea')
venner
```

```
{'Anne', 'Lea', 'Ole', 'Peter'}
```

For en samlet oversigt over metoder hørende til mængder se
https://www.w3schools.com/python/python_sets_methods.asp

Sandhedsværdier

Sandhedsværdier (= booleans) repræsenterer én af to værdier `True` og `False`

```
10 > 7
```

```
True
```

```
'bord' < 'abe'
```

```
False
```

I det første eksempel afgøres det, at det er sandt, at 10 er større end 7, og i det andet ses det, at det er falskt, at ordet 'bord' kommer før ordet 'abe' i forhold til en alfabetisk ordning

Variable kan få tildelt sandhedsværdier, dvs enten `True` eller `False`

```
resultat = 5 < 2
print(resultat)
```

```
False
```

Variablen resultat er af typen `bool`

```
type(resultat)
```

```
bool
```

Et andet eksempel på brugen af sandhedsværdier

```
snit = 7.3
optaget = snit > 6
print(optaget)
```

```
True
```

Udtryk

Udtryk er sammensat af forskellige dele, der tilsammen evaluerer til en konkret værdi, og vi skal gennemgå nogle eksempler, hvor det første indbefatter tal

```
pris = 120
antal = 4
moms = 0.25
samletPris = antal * pris * (1 + moms)
print(samletPris)
```

```
600.0
```

ovenfor er

```
antal * pris * (1 + moms)
```

et udtryk, der evaluerer til et decimaltal (= `float`)

Det næste eksempel handler om strenge

```
fornavn = 'Mads'
efternavn = 'Olsen'
hilsen = 'Hej ' + fornavn + ' ' + efternavn
print(hilsen)
```

```
Hej Mads Olsen
```

hvor

```
'Hej ' + fornavn + ' ' + efternavn
```

evaluerer til en streng (= `str`)

Det sidste eksempel

```
temperatur = 22
regn = False
solskin = True
resultat = (temperatur > 20 and solskin) and not regn
print(resultat)
```

```
True
```

indeholder sandhedsværdier (= `booleans`), hvor

`(temperatur > 20 and solskin) and not regn`

er et udtryk, der giver sandhedsværdien `True`

Kontrolstrukturer

I Python findes der ligesom i andre programmeringssprog tre typer kontrolstrukturer

- **sekvens**, som betyder, at instruktionerne i et program udføres én efter én i den rækkefølge, de står skrevet
- **forgrening**, som gør det muligt at vælge mellem forskellige blokke af instruktioner baseret på en betingelse
- **gentagelse**, som bruges til at udføre de samme instruktioner flere gange – enten et bestemt antal gange eller så længe en betingelse er opfyldt

Sekvens

En sekvens består af et endeligt antal instruktioner, der udføres én efter én i den rækkefølge, de står skrevet

```
instruktion 1
instruktion 2
instruktion 3
...
...
instruktion n
```

Eksempel på en sekvens

```
b = float(input('Indtast bredden af rektanglet: '))
h = float(input('Indtast højden af rektanglet: '))
areal = b * h
omkreds = 2 * (b + h)
print('Arealet er', areal, 'og omkredsen er', omkreds)
```

Forgrening

Der er tre typer forgreninger i Python

`if` forgrening

```
if betingelse:
    blok af instruktioner
```

`if-else` forgrening

```
if betingelse:
    blok af instruktioner
else:
    blok af instruktioner
```

`if-elif-else` forgrening

```
if betingelse:
    blok af instruktioner
elif betingelse:
    blok af instruktioner
else:
    blok af instruktioner
```

Bemærk, at

- `betingelse` er et udtryk, der er af typen `bool`, og dermed har værdien **`True`** eller **`False`**
- `blok af instruktioner` kan indeholde alle typer af instruktioner, herunder også forgreninger
- der kan indgå flere **`elif`** blokke

Eksempler på forgreninger

Ved køb af varer på en webshop gives der 10% i rabat, hvis der samlet er købt for mindst 500 kr.

```
if samletPris >= 500:
    rabat = 0.1 * samletPris
```

Betingelsen
`samletPris >= 500`
kræver, at variablen `samletPris` tidligere i programmet har fået tildelt en værdi, der er et heltal, `int`, eller et decimaltal, `float`

Næste eksempel udskriver, om den pågældende person er myndig baseret på alderen, og hvis alderen er mindst `18`, udskrives 'Du er myndig', og ellers udskrives 'Du er ikke myndig'. Som i det første eksempel skal variablen `alder` inden forgreningen have fået tildelt en talværdi

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

I tredje eksempel bruges **`elif`**, som kort står for else if, i forbindelse med vurdering af konditallet

```
if kondital < 39:
    print('Lavt')
elif kondital < 44:
    print('Under middel')
elif kondital < 52:
    print('Middel')
elif kondital < 57:
    print('Godt')
else:
    print('Rigtig godt')
```

Ovenstående **`if-elif-else`** forgrening medfører, at der udskrives 'Lavt', hvis konditallet er under `39`, og der udskrives 'Under middel', hvis konditallet er mindst `39`, men under `44` osv.

Gentagelse

Der er to typer gentagelser i Python

`for` løkke

```
for variabel in sekvens:
    blok af instruktioner
```

Bemærk, at

- `sekvens` skal opfattes som en iterabel datastruktur f.eks. `range(5)` og ikke som en kontrolstruktur
- `blok af instruktioner` kan indeholde alle typer af instruktioner, herunder også andre gentagelser
- en **`for`** løkke sikrer som udgangspunkt, at gentagelsen foregår et bestemt antal gange

`while` løkke

```
while betingelse:
    blok af instruktioner
```

Bemærk, at

- `betingelse` er et udtryk, der er af typen `bool`, og dermed har værdien **`True`** eller **`False`**
- gentagelsen af `blok af instruktioner` fortsætter, mens `betingelse` er **`True`**

- `blok af instruktioner` kan indeholde alle typer af instruktioner, herunder også andre gentagelser

I mange programmeringssprog findes der også en **`repeat-until`** løkke, men det gør der ikke i Python – i stedet må det omformuleres til en **`while`** løkke

Eksempler på løkker

```
for tal in range(4):
    print(tal*tal)
```

```
0
1
4
9
```

I dette eksempel gennemløber variablen `tal` tallene 0, 1, 2, 3 og udskriver kvadrattallene 0, 1, 4, 9

```
venner = ['Ole', 'Ida', 'Mia', 'Mads']
for ven in venner:
    print(ven)
```

```
Ole
Ida
Mia
Mads
```

Ovenfor gennemløber variablen ven elementerne i listen `venner` og udskriver vennerne

```
adgangskode = ''
while adgangskode != 'h3mm3l1g':
    adgangskode = input('Indtast adgangskode: ')
print('Så kom du igennem...')
```

Her gentages indlæsningen af adgangskoden, indtil den rigtige adgangskode er indtastet. Ofte tillades kun et bestemt antal forsøg med indtastning, hvilket ovenstående ikke tager højde for

I næste eksempel kastes der med en terning, indtil der kommer en 6'er

```
import random

kast = 0
while kast != 6:
    kast = random.randint(1, 6)
    print("Du slog: ", kast)
```

Funktioner

Definition

Funktioner er selvstændige enheder i et program med en specifik opgave, hvor funktionen kommunikerer med andre dele af programmet, herunder andre funktioner, gennem veldefinerede grænseflader

Funktioner defineres i Python på følgende overordnede måde

```
def funktionens navn (liste af formelle parametre):
    kroppen af funktionen
```

Bemærk, at

- `def` er et reserveret ord i Python, som fortæller, at en funktion bliver defineret
- navnet på funktionen skal opfylde Pythons regler for navngivning
- de formelle parametre er adskilt af kommaer
- listen af formelle parameter kan være tom, dog skal parenteserne under alle omstændigheder medtages, og der afsluttes altid med et kolon
- kroppen af funktionen er bygget op af Python-sætninger, der er indrykket

Eksempel på en funktion, der bestemmer summen af to tal

```
def summen(tal1, tal2):
    total = tal1 + tal2
    return total
```

Her er der to **formelle parametre** `tal1` og `tal2`, og kroppen af funktionen består af en tildeling og en `return`-sætning, der betyder, at funktionen returnerer værdien af variablen `total`, dvs summes af de to tal.
Bemærk, at `return` er et reserveret ord i Python, som udelukkende bruges i funktioner

Funktionen kan nu bruges af det øvrige program ved et **funktionskald**

```
summen(5, 12)
```

```
17
```

hvor tallene `5` og `12` er **aktuelle parametre**, som også kaldes for **argumenter**, og funktionskaldet returnerer i dette eksempel værdien 17

I ovenstående funktionskald er de aktuelle parametre konkrete tal `5` og `12`, men det kunne også være udtryk:

```
tal = 13
nytTal = summen(5*tal, 60)
print(nytTal)
```

```
125
```

bemærk, at

- variablen `tal` tildeles værdien `13`
- variablen `nytTal` tildeles et udtryk
  `summen(5*tal, 60)`
- den første aktuelle parameter
  `5*tal`
  er et udtryk, der skal evalueres, hvilket giver 65
- den formelle parameter `tal1` sættes til 65
- den formelle parameter `tal2` sættes til `60`
- den lokale variabel `total` tildeles værdien 125
- funktionen returnerer værdien 125
- variablen `nytTal` tildeles værdien 125
- værdien 125 udskrives

Når en funktion returnerer en værdi, kan et funktionskald indgå i et udtryk:

```
resultat = summen(5,7) * summen(6,4)
print(resultat)
```

```
120
```

som medfører, at variablen `resultat` tildeles værdien `120`

For at kunne indgå i et udtryk skal funktionen indeholde en **`return`**-sætning, der betyder, at funktionen returnerer den pågældende værdi, som fremgår af **`return`**-sætningen. Bemærk, at der godt kan være flere **`return`**-sætninger i kroppen af funktionen, og at funktionen stopper efter en **`return`**-sætning, som dermed vil være den første, som funktionen møder

Varierende parameterliste

Ovenfor kiggede vi på en funktion, der bestemmer summen af to tal

```
def summen(tal1, tal2):
    total = tal1 + tal2
    return total
```

På samme måde kan vi definere en funktion, der bestemmer summen af tre tal

```
def summen(tal1, tal2, tal3):
    total = tal1 + tal2 + tal3
    return total
```

og en funktion, der bestemmer summen af fire tal

```
def summen(tal1, tal2, tal3, tal4):
    total = tal1 + tal2 + tal3 + tal4
    return total
```

Mere generelt kan vi definere en funktion, der bestemmer summen af en række tal uden på forhånd at kende antallet af tal

```
def summen(*talliste):
    resultat = 0
    for tal in talliste:
        resultat += tal
    return resultat
```

Bemærk, at tegnet `*` foran den formelle parameter `talliste` indikerer, at der er tale om en liste (mere præcist en tupel) af tal uden at kende længden af listen (tuplen)

Med denne funktion kan vi lave følgende funktionskald

```
summen(2, 6), summen(4, 1, 7, 12), summen(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

```
(8, 24, 45)
```

Default parameterværdier

Nedenfor ses en funktion, der beregner momsbeløbet, når momssatsen er 25%, og herefter et funktionskald, der returnerer momsen af 120 kr.

```
def beregnMoms(udenMoms):
    return udenMoms * 0.25
```

```
beregnMoms(120)
```

```
30.0
```

Hvis vi nu ønsker at eksperimentere med forskellige momssatser, kan momssatsen inddrages som en formel parameter, hvilket vises nedenfor, og efterfølgende er der to funktionskald for at tjekke forskellige momssatser

```
def beregnMoms(udenMoms, momsSats):
    return udenMoms * momsSats
```

```
beregnMoms(120, 0.25), beregnMoms(120, 0.15)
```

```
(30.0, 18.0)
```

Hvis det er i meget specielle tilfælde, at vi ønsker at anvende en alternativ momssats end 25%, kan vi bruge `0.25` som default momssats på følgende måde

```
def beregnMoms(udenMoms, momsSats = 0.25):
    return udenMoms * momsSats
```

```
beregnMoms(120), beregnMoms(120, 0.25), beregnMoms(120, 0.15)
```

```
(30.0, 30.0, 18.0)
```

Her ser vi, at momssatsen automatisk sættes til `0.25` – en default værdi – medmindre der angives en alternativ momssats

Specifikation af funktioner

Det er vigtigt med en præcis specifikation til de enkelte funktioner – specielt når der er tale om funktioner på mange linjer, og et eksempel på en specifikation ses her

```
def summen(*talliste):
    """ Forudsætter at talliste er en liste af tal
        Returnerer summen af tallene, og hvis listen er tom, returneres 0 """
    resultat = 0
    for tal in talliste:
        resultat += tal
    return resultat
```

hvor der er indskrevet kommentarer om, hvad funktionen forudsætter, og hvad den returnerer. Kommentarerne står mellem """ og """
Det kaldes en `docstring`, og med en gennemtænkt `docstring` kan brugere af funktionen nøjes med at læse denne fremfor at dykke ned i selve koden, hvilket selvfølgelig kræver, at funktionen er gennemtestet. Indholdet af `docstring` kan også findes ved at anvende den indbyggede funktion `help`

```
help(summen)
```

```
Help on function summen in module __main__:

summen(*talliste)
    Forudsætter at talliste er en liste af tal
    Returnerer summen af tallene, og hvis listen er tom, returneres 0
```

Docstrings erstatter ikke de hensigtsmæssige kommentarer ved de enkelte sætninger

Bemærk, at `help` også kan bruges på indbyggede funktioner i Python

Specielle funktioner

Lambda-funktioner er små, såkaldte anonyme funktioner, som kan gøre koden mere kompakt, hvilket til gengæld også kan betyde, at koden bliver sværere at læse

Nedenfor ses kvadrat-funktionen defineret på sædvanligvis som en almindelig funktion og efterfølgende som en lambda-funktion

```
def kvadrat(x):
    return x*x

print(kvadrat(4))
```

```
16
```

```
kvadrat = lambda x: x*x
print(kvadrat(4))
```

```
16
```

Funktionen
`sorted()`
kan bruges til at sortere en liste, en tuple eller en dictionary

```
navne = ['Anna', 'Ib', 'Charlotte', 'Peter']

sorteret = sorted(navne)
print(sorteret)
```

```
['Anna', 'Charlotte', 'Ib', 'Peter']
```

Bemærk, at elementerne er strenge, hvorved sorteringen er alfabetisk, og hvis der i stedet havde været tal, ville disse sorteres efter størrelse, og man kan også sortere fra 'størst' til 'mindst' ved at bruge en default parameter `reverse`

```
navne = ['Anna', 'Ib', 'Charlotte', 'Peter']

sorteret = sorted(navne, reverse=True)
print(sorteret)
```

```
['Peter', 'Ib', 'Charlotte', 'Anna']
```

Sorteringsmåden kan også fastsættes ved hjælp af funktioner som fx `len()`

```
navne = ['Anna', 'Ib', 'Charlotte', 'Peter']

sorteret = sorted(navne, key=len)
print(sorteret)
```

```
['Ib', 'Anna', 'Peter', 'Charlotte']
```

Når dictionaries skal sorteres, skal der tages stilling til, om der skal sorteres efter nøgle eller efter værdi
Først sorterer vi efter nøgle

```
frugter = {'æble': 3, 'banan': 2, 'citron': 1, 'pære': 4}

sorteret = sorted(frugter.items())
print(sorteret)
```

```
[('banan', 2), ('citron', 1), ('pære', 4), ('æble', 3)]
```

Vi bemærker, at resultatet af sorteringen er en liste og ikke et dictionary, men det kan vi nemt ændre på ved at anvende `dict()`–funktionen

```
frugter = {'æble': 3, 'banan': 2, 'citron': 1, 'pære': 4}

sorteret = dict(sorted(frugter.items()))
print(sorteret)
```

```
{'banan': 2, 'citron': 1, 'pære': 4, 'æble': 3}
```

Skal vi sortere efter værdi, kan vi bruge en **`lambda`**-funktion

```
frugter = {'æble': 3, 'banan': 2, 'citron': 1, 'pære': 4}

sorteret = dict(sorted(frugter.items(), key=lambda x: x[1]))
print(sorteret)
```

```
{'citron': 1, 'banan': 2, 'æble': 3, 'pære': 4}
```

hvor **`lambda`**-funktionen returnerer værdien med `x[1]`, idet x er et element i dictionary, dvs (nøgle, værdi)

Vi kan også sortere efter værdier med de største først ved at bruge
`reverse = True`

```
dict(sorted(frugter.items(), key=lambda x: x[1], reverse= True))
```

```
{'pære': 4, 'æble': 3, 'banan': 2, 'citron': 1}
```

eller med et minus i **`lambda`**-funktionen

```
dict(sorted(frugter.items(), key=lambda x: -x[1]))
```

```
{'pære': 4, 'æble': 3, 'banan': 2, 'citron': 1}
```

Når vi har en liste bestående af en række elementer, og vi ønsker at bruge den samme funktion på alle elementerne, kan vi gennemløbe listen i en for-løkke og løbende danne de nye elementer til den nye liste

```
distancer = [2800, 2550, 2790, 3050, 2180, 3200]

def kondital(dist):
    return round((dist-505)/45, 1)

resultater = []
for distance in distancer:
    resultat = kondital(distance)
    resultater.append(resultat)

print(resultater)
```

```
[51.0, 45.4, 50.8, 56.6, 37.2, 59.9]
```

En smartere måde at gøre ovenstående er ved at bruge `map()`–funktionen

```
distancer = [2800, 2550, 2790, 3050, 2180, 3200]

def kondital(dist):
    return round((dist-505)/45, 1)

resultater = list(map(kondital, distancer))

print(resultater)
```

```
[51.0, 45.4, 50.8, 56.6, 37.2, 59.9]
```

og med kondital-funktionen som en lambda-funktion

```
distancer = [2800, 2550, 2790, 3050, 2180, 3200]

resultater = list(map(lambda x: round((x-505)/45, 1), distancer))

print(resultater)
```

```
[51.0, 45.4, 50.8, 56.6, 37.2, 59.9]
```

Endelig er der muligheden med list comprehension

```
[round((x-505)/45, 1) for x in distancer]
```

```
[51.0, 45.4, 50.8, 56.6, 37.2, 59.9]
```