Linux – Friheden til at programmere i Python

Version 1.0.20040516 – 2020-12-31

Alfred Jensen

Linux – Friheden til at programmere i PythonVersion 1.0.20040516 – 2020-12-31

af Alfred Jensen

En indføring i programmering i Python på og til Linux.

Indholdsfortegnelse

Figurliste

Forord

1. Forord

Denne bog er skrevet som en indføring i programmering i Python.

2. Linux-bøgerne

Bogen er en del af en serie, som kan findes på http://www.linuxbog.dk/

- *Linux – Friheden til at vælge installation* – Om at installere Linux.
- *Linux – Friheden til at lære Unix* – Om hvordan man bruger Linux' (og Unix') kommandolinjeværktøjer.
- *Linux – Friheden til at vælge grafisk brugergrænseflade* – Om alle de grafiske brugergrænseflader, der findes til Linux.
- *Linux – Friheden til at vælge programmer* – Om de programmer du kan få til Linux.
- *Linux – Friheden til systemadministration* – Om at administre sit eget linuxsystem.
- *Linux – Friheden til at programmere* – Programmering på Linux
- *Linux – Friheden til at programmere i C* – Om at programmere i sproget "C".
- *Linux – Friheden til at programmere i Java* – Om at programmere i sproget "Java".
- *Linux – Friheden til sikkerhed på internettet* – Om at sikre dit Linuxsystem mod indbrud fra internettet.
- *Linux – Friheden til egen webserver* – Om at sætte en webserver med databaser, CGI-programmer og andet godt op.
- *Linux – Friheden til at skrive dokumentation* – Om at skrive dokumentation (og andet) i SGML/DocBook, LaTeX eller andre formater.
- *Linux – Friheden til at vælge kontorprogrammer* – Kontorfunktioner på et Linux/KDE/OpenOffice.org-system.
- *Linux – Friheden til at vælge IT-løsning* – Om muligheder, fordele og ulemper ved at bruge Linux i sin IT-løsning.
- *Linux – Friheden til at vælge OpenOffice.org* – Om at bruge OpenOffice.org, både på Linux og på andre styresystemer.
- *Linux – Friheden til at vælge digital signatur* – Digital signatur på Linux.

3. Ophavsret

Denne bog er skrevet af Linux-brugere til Linux-brugere. Store dele af bogen er skrevet eller redigeret af enkelte forfattere, hvilket er nævnt i revisions-historien til bogen.

Bogen kan findes i opdateret form på http://www.linuxbog.dk/, mens prøve-udgaver kan findes på http://cvs.linuxbog.dk/.

Figur 1. ÅDL

Kommentarer, ris og ros og specielt fejl og mangler bedes sendt til linuxbog@sslug.dk (mailto:linuxbog@sslug.dk), men er du medlem af SSLUG kan du i stedet for med fordel skrive til sslug-bog@sslug.dk (mailto:sslug-bog@sslug.dk).

4. Om forfatterne og bogens historie

Denne bog er skrevet af Alfred Jensen som en tilføjelse til bogserien »Linux - Friheden til ...«. Bogen er som resten af serien en fri bog der kan læses enten i smådele eller som en helhed på http://www.linuxbog.dk/.

5. Vi siger tak for hjælpen

Vi har haft stor glæde af mange SSLUG-medlemmers støtte, rettelser og forslag til forbedringer – bliv ved med dette. Specielt vil vi nævne:

- Gitte Wange har oversat bogen fra LaTeX til Docbook/XML, så den kunne blive en del af »Linux – Friheden til ...«-serien.

Du kan i Appendiks A finde en liste over alle de ændringer som bogen har været igennem.

Hvis du støder på ord du ikke forstår, og de ikke findes i bogens ordliste, så kan http://www.whatis.com/ være et nyttigt opslagsværk. Desværre er det kun på engelsk. Du må meget gerne også skrive til redaktionen (mailto:linuxbog@sslug.dk), så vi kan få ordet med i ordlisten i næste udgave.

6. Typografi

Vi vil afslutte indledningen med at nævne den anvendte typografi.

- Navne på filer og kataloger skrives som `foo.bar`.
- Kommandoer du udfører ved at skrive dem på en kommandolinje, skrives som **help**.
- Der er flere steder i bogen hvor vi viser hvad brugeren taster, og hvad Linux svarer. Det vil se ud som:

  ```
  hven% Dette taster brugeren
  Dette svarer Linux.
  ```

- Der er tilsvarende flere steder i bogen hvor vi viser hvad systemadministratoren (root) taster, og hvad Linux svarer. Det vil se ud som:

  ```
  hven# Dette taster systemadministratoren
  Dette svarer Linux.
  ```

  Det vigtige her er at kommandofortolkeren bruger nummertegnet (#) til at markere at man har systemadministratorrettigheder.

Kapitel 1. Python for begyndere

Figur 1-1. Python undervisning

1.1. Hvad er Python?

Python er et kraftfuldt computersprog, der er ekstrem effektivt til udvikling af internet og webbaserede programmer.

Dets udvikling begyndte i det sene efterår i 1989. I foråret 1991 blev Python almindelig tilgængelig.

Den første bog overhovedet om sproget udkom i begyndelsen af 1996. Herværende bog er mig bekendt den første på dansk og på de skandinaviske sprog i det hele taget. I 1999 regnede en ledende industriobservatør med at der var omkring 300.000, der brugte Python. Det tal må skønnes at være meget større i dag.

Bl.a. er man meget hurtig til at frigive eventuelle rettelser, ligesom det er særdeles let at få besvaret spørgsmål, såfremt man mener at have konstateret fejl i sproget. Fejl der meget let kan vise sig at være

reelle forbedringer i forbindelse med overgangen til bl.a. unicode (de større tegntabeller).

Jeg nævner dette allerede her, fordi der kan være meget stor forskel på, hvordan koden skulle skrives før version 2.2 og efter. Der kom nemlig en meget stor forbedring af sproget. I skrivende stund er det vist mere end svært at finde større fejl, selv behandlingen af de dansk/norske specialtegn er kommet næsten på plads.

Med version 2.3 skulle Python nu kunne håndtere 64 bits tegn - i den første ASCII tegntabel kunne et tegn, en karakters nummer være op til 8 bit lang. Fra og med Python version 2.4 vil Python droppe 8 bits tegntabellen totalt.

1.2. Hvorfor bruge Python?

Hvorfor i al verden skulle man interessere sig for Python, når der i forvejen er en lang stribe af rigtig gode gamle og velfungerende computersprog at gå til? Lad mig som eksempler nævne:

1. Python findes til alle de velkendte styresystemer herunder Linux, Mac og Windows.
2. Python er let at flytte fra et styresystem til et andet som f.eks. fra Linux til Windows og omvendt.
3. Er meget let at lære - også for begyndere indenfor brugen af computersprog i det hele taget.
4. Python kombinerer bemærkelsesværdig styrke med en meget klar syntaks (læs: måde at gøre tingene på).
5. I Python kan du lære objekt-orienteret programmering (programmering og anvendelse af vinduer, knapper og den slags) uden af forstå mange af de komplekse detaljer forinden. Senere kan du så om ønsket sætte dig ind i teknikken bag konstruktionen.
6. Python er et begyndervenligt computersprog, der automatisk styrer mange af de komplekse detalier, der foregår bag scenen. Det gør det muligt for dig at bruge kræfterne på de overordnede ting i dit projekt uden at skulle grave i dybden for hvert eneste skridt. Lad os også her foregribe begivenhederne lidt og se følgende eksempel, der nok også kan interessere den erfarne Java, C++ eller Delphi programmør.

```
>>> from Tkinter import Label
>>> widget = Label()
```

Egentlig burde der nok have været 1 programlinje mere i det viste eksempel. Den skulle i så fald hedde widget.pack() Funktionen pack() stiller objektets bestanddele på plads på udskriftsenheden (skærmen). Men i dette tilfælde er der ikke noget at stille på plads, så de 2 programlinjer er tilstrækkelige. Du kan lave nøjagtig samme vindue med:

med Entry:

```
>>> from Tkinter import Entry
>>> barn = Entry()
```

```
>>> barn.pack()
```

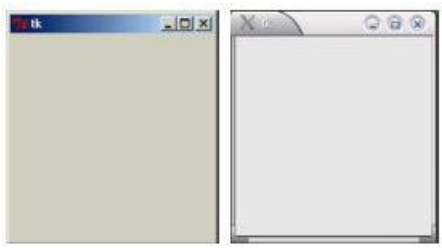
eller med Text (og en lang række andre objekter):

```
>>> from Tkinter import Text
>>> barn = Text()
>>> barn.pack()

>>> from Tkinter import *
>>> barn = Frame().pack(expand = YES, fill = BOTH)
```

Læg allerede her mærke til forskellen på 2 ting. 1: Det du formentlig havde forventet from Tkinter import Frame er blevet til from Tkinter import * (her sker det for at kunne udnytte parameterværdierne i pack) og 2: funktionen barn.pack() er blevet flyttet til Frame() og har fået 2 parametre (argumenter). Det sker for at give Frame arvingen barn mulighed for at kunne udvides (sker med expand = YES) og det i både vandret og lodret retning eller med andre ord til at kunne udfylde hele skærmen. Frame objektet er en kontainer, hvori alle andre grafiske objekter som tekstbokse, knapper, labels og mange andre kan indsættes. Dem er der jo ingen af i eksemplet her, så Frame().pack(expand = YES, fill = BOTH) virker helt på samme måde, som når du klikker på knappen længst til venstre i vinduets øverste højre hjørne. Den får vinduet til at klappe sammen til dets mindst mulige størrelse. expand = YES giver rammen (the frame - kontaineren) mulighed for at kunne udvides efter behov, men her der ikke behov (der skal ikke sættes andre komponenter ind i kontaineren). Hvis der senere sættes et eller flere af den nævnte og tilsvarende objekter ind i rammen (the Frame), vil den blive udvidet efter behov lige så længe der er plads til objekterne i det givne skærmareal. Er der ikke det, vil objekterne ganske enkelt ikke blive viste.

Figur 1-2. Eksempel på vinduer



Naturligvis har de to vinduer nøjagtig de samme åbne/lukke muligheder m.v. som du kender dem fra f.eks. internet browsere i Windows. Det kan da kaldes objektorienteret programmering, så det forslår. Alle vinduer, knapper og andre normale objekter er forprogrammerede, så du direkte kan bruge dem og selvfølgelig tilpasse dem efter dit behov i det givne øjeblik. I C++ skal du skrive masser af programlinjer for at opnå det tilsvarende. Det er i øvrigt meget let at integrere Python og C++ i hinanden.

Noget af det fornemmeste ved OOP (objekt orienteret programmering) er genbrugen. Med de tre viste eksempler har du allerede set, at Python har fremragende egenskaber også på dette område, for

naturligvis kan koden til et fuld færdigt vindue umuligt samles og udføres i de 2-3 programlinjer, det enkelte eksempel indeholder. Der skal mange gange så megen koden til. Hvis ikke Python besad evnen til genbrug, så skulle det meget store program, der skal til for at oprette det fuldt færdige vindue udvikles hver eneste gang, vinduet skulle bruges - altså 3 gange alene til de viste eksempler. Det ville blive kæmpeprogrammer, der i virkeligheden ville være næsten uanvendelige grundet det enorme omfang. Koden til det viste eksempel og til en række andre er indeholdt i Widget klassen. Label, Entry, Text og Frame er alle subklasser i Widget.

7. Python er rygraden i Jython. Det betyder, at en større eller mindre del at et Jython program er og fortsat vil være Python. I Jython er det ofte lettere at udvikle eksempelvis Java applets end det er i selve Java. Applets kan bruges som integrerede dele af HTML, XHTML, XML sprogene til hjemmesider.
8. I Linux taler man ofte om hele programpakken med de førnævnte programpakker som Linux. Det er forkert. Linux er kernen (centralenheden/det styrende program. I Python er der også en kerne. Den er lille, hvilket den også bør være. Til denne styrende enhed er der et meget stort bibliotek (library). Det betyder, at meget af det, du får brug for, allerede er udviklet og testet for dig. Din opgave vil være at skrive den kode, der kombinerer nævnte biblioteks komponenter og udvikle nye egenskaber efterhånden som der bliver brug for det, men du kan i høj grad også selv udvide Python med dine egne funktioner (metoder), klasser, moduler m.v.

1.3. Hvad koster Python, og hvor får jeg det fra?

Python er gratis og tilgængelig for langt de fleste styresystemer. Har du ikke den nyeste version, kan du altid hente den på http://www.python.org. Det er særdeles let at installere Python, hvis sproget ikke allerede er installeret på din computer. Det er det dog normalt hvis du bruger SuSE, RedHat eller Mandrake.

Der er en meget stor udvikling af Python i gang. Alene de ca. 3 måneder, det har taget mig at skrive herværende bog, er der kommet 2 fulde opdateringer af sproget (de udkomne versioner er 2.2 og 2.3). Men ikke alene det, der kommer også opdateringer til sidstnævnte, uden at versionsnavnet ændres. Lige nu den 26. januar 2004 er version 2.3.3 den aktuelle. I version 2.4 skulle der ske ret store ændringer, så Python kan endnu mere (bl.a. behandle 64 bits karakterkode).

1.4. Guide til Python dokumentation

Hvis man er helt ny udi programmering kan det være svært at forstå hvordan dokumentationen af et programmeringssprog er skruet sammen. Men når man først kan gennemskue det, er det tit og ofte meget nemmere at hente hjælp i dokumentationen af sproget end at skulle spørge om hjælp til en specifik funktion på en mailing-liste.

Dokumentation til python kan du finde online på adressen: http://www.python.org/doc/. Her kan du desuden finde guides og HowTo'er til diverse emner. Dokumentationen er delt op i 7 kategorier: Modul oversigt, tutorial, biblioteks reference, Macintosh reference, sprog reference, udvidelser af sproget og et Python/C API. Forklaring på kategorierne:

- **Modul oversigt** - Her er den en oversigt over alle de moduler, der kommer med en release af Python. For hvert modul findes desuden dokumentation af det.
- **Tutorial** - Her finder du en gennemgribende tutorial på den mest basale brug af Python. Et godt sted at starte hvis man i forvejen kan programmere, men bare ikke kender Python.
- **Biblioteks reference (Library reference)** - En gennemgang af de mest brugte biblioteker i Python samt mere gennemgribende eksempler på brug end du finder i modul oversigten.
- **Macintosh reference** - Gennemgang af de biblioteker der er specifikke for Macintosh.
- **Sprog reference (Language reference)** - Gennemgang af sprogets brug og opbygning (syntax, funktioner, klasser m.m.)
- **Sprog udvidelser (Extending and embedding)** - En guide i hvordan du kan udvide Python med C/C++, embedde Python i andre applikationer m.m.
- **Python/C API** - En gennemgang af et C API til Python, der lader C programmører bruge Python i deres applikationer.

Kapitel 2. Sprog reference

2.1. GUI og IDLE - hvad er det?

GUI er en forkortelse af Graphic User Interface og er de programmer, som dels opretter de virtuelle vinduer, knapper, menuer m.v. Dette giver brugeren mulighed for ved klik m.v. at kalde de med nævnte elementer forbundne funktioner, hvis formål kan være at sende en tekst til en tekstboks, hente en fil ind eller sende en ud på et eksternt lager som f.eks. en harddisk, tegne grafik og meget andet. Koden til GUI komponenterne findes bl.a. i Pythons Tk bibliotek. At koden til vinduerne kommer fra Tk biblioteket kan ses af ordet Tk øverst i grundvinduet.

IDLE er en forkortelse for Integrated DeveLopment Environment for Python. IDLE er en udviklings IDE til Python som man kan bruge til at skrive simple programmer i.

```
>>> from Tkinter import Label
>>> barnAfLabel = Label()
```

Da der som nævnt findes en grundmodel af GUI komponenterne i Tk, kan værkføreren Tkinter sende bud ud i Tk, om at koden til opførelsen af grundvinduet skal gøres tilgængelig til opførelse af en kopi af Label(), som jeg i dette tilfælde har valgt at kalde barnAfLabel, fordi det som andre børn har sit ophav, hvorfra det arver i moderen, en betegnelse, der også ses brugt i andre danske bøger om computersprog. På engelsk kaldes moderen for the parent (forælderen). I Python bruges også betegnelsen the main class eller the basic class.

2.2. Immediate (interaktiv) mode

Fra gamle DOS kender du muligvis dosprompten c:\> klarmeldingen, til at computerne kan modtage dine kommandoer. I Python anvendes >>> som en tilsvarende klarmelding. En melding om at Python er i immediate eller, som den også kaldes, interaktiv mode. Det er herfra, du begynder din Python programmering, men bestemt også måden den erfarne Python programmør vender tilbage til og anvender f.eks. ved test af nye programstumper.

2.3. Kommentarer

Når du programmerer har du fra tid til anden brug for at indsætte kommentarer - dele af programmet, som du og andre kan læse, men som computersproget springer over. I Python markerer \# (hashtegnet -

på dansk kaldet havelåge eller bøftegn). En kommentar begynder ved havelågen og fortsætter programlinjen ud. Eks. 1:

```
>>> # Python springer over kommentarer
>>> print "Kommentarer er til dig og ikke til computeren."
Kommentarer er til dig og ikke til computeren.
```

Det er også muligt at udskrive hash-tegnet, hvis det er en del af en tekst-streng. Eks. 2:

```
>>> print "Her udskriver vi et hash-tegn #"
Her udskriver vi et hash-tegn #
```

I dette eksempel var der INGEN kommentarer. Bøftegnet er her en del af en teksstreng. En sådan skal Python meget gerne reagere på og ikke forbigå. Læg mærke til at bøftegnet udskrives her, hvad det ikke gjorde i eksemplet, hvor det tjente sit rette formål.

2.3.1. Kommentarer i anførselstegns par

Strenge kan også være indesluttet i 3 anførselstegn (enkelte eller dobbelte). Eks. 1:

```
>>> print """
... Navn:       Klippeøen Bornholm
... Befolkning: 44.500
... """

Navn:    Klippeøen Bornholm
Befolkning:     44.500
```

Eks. 2:

```
>>> def Funktion():
...     """Indskudt tekst"""
...     return "Normalt bruges 1 tegnsæt"
...
>>> Funktion()
'Normalt bruges 1 tegns\xe6t'
```

Eks. 3:

```
>>> class Klasse:
...     """ Bruges ofte, hvor teksten ikke skal udskrives. """
...     print "Normal tekst til normal udskrift."
...
Normal tekst til normal udskrift.
```

Eks. 4:

```
>>> def inaktiv():
...     """Vær passiv, men dokumemter det.
... Nej, der sker virkelig intet."""
...     pass
```

```
...
>>> inaktiv()
```

Bemærk at tre anførselstegn (enkelte eller dobbelte efter dit ønske) lader Python springe over alt mellem de første (indledende) anførselstegn og de sidste, hvad du også kan se, af at Nej ... ikke behøver den ellers krævede blokindrykning.

2.4. Aritmetiske operatorer

- Addition + eks: 3 + 4 = 12
- Subtraktion - eks: 3 - 4 = -1
- Multiplikation * eks: 3 * 4 = 12
- Eksponent notation ** eks: 3 ** 4 = 81
- Division / eks: 3 / 4 = 0 (afrunder nedad til nærmeste hele tal)
- Division // eks: 3 // 4 = 0 (afrunder nedad til nærmeste hele tal)
- Modulus % eks: 3.0 % 4.0 = 0.75

FARE: Pas på ved heltalsdivisioner

```
>>> 25 / 24
 1
```

25 divideret med 24 burde give 1.0416666666666667, men Python runder automatisk af nedad til nærmeste heltal, som i dette tilfælde er 1.

```
>>> 25 / 24.0
1.0416666666666667
```

For at få decimaldelen returneret (udskrevet) er det nødvendigt, at mindst det ene af tallene er et decimaltal (et tal med flydende komma).

I Python er der fuld støtte for flydende tal (floating point numbers). Operatorer med operanter af blandet type konverterer heltals (integer) operanten til tal med flydende komma (på engelsk punktum):

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

```
>>> print "simpelt eksempel " * 3
 simpelt eksempel simpelt eksempel simpelt eksempel
```

I de traditionelle computersprog kan det ikke gøres så enkelt. Der kræves noget som:

```
print "simpelt eksempel " + "simpelt eksempel " + "simpelt eksempel "
```

2.5. Variabler

En variabel er en pegepind, der peger på et antal adresser i computerens arbejdslager (RAM), hvor data, der kan være flere ting f.eks. tal og tekststrenge, placeres. Variablen skal tildeles et navn, der ikke må indeholde de "ikke-engelske" specialtegn ÆØÅ og æøå. Det skal lige bemærkes, at der i visse underversioner af version 2.3 har været muligheder for at bruge de nævte tegn.

2.5.1. Erklæring af variabler

Når programmørerne skulle foretage beregninger på Dask, Danmarks første computer, måtte de tænke i nuller og et taller. Det er meget ubekvemt for mennesker så tallene blev erstattet med ord - i første omgang ord på 3 bogstaver. Det var en stor lettelse for programmørerne. Nu kan variabel- og andre navne normalt have op til 256 bogstaver eller tal blot bør første tegn i et variabelnavn være et bogstav, der ikke må være et dansk/norsk specialtegn æøå og ÆØÅ. I Python er der for så vidt ikke noget i vejen, for at første tegn kan være et understreg tegn, men det kan være en rigtig dårlig løsning, da fordefinerede navne m.v. herved let vil kunne overskrives. I Python skal variabler ikke erklæres. Erklæringen sker automatisk. MEN BEMÆRK Python gør forskel på store og små bogstaver, d.v.s. at variablerne a og A er to forskellige variabler og vil blive fortolket som sådanne. Det samme gælder naturligvis også alle andre navne (på lister, tuples m.v.)

```
>>> # Først tildeles x værdien nul, så tildeles y værdien i x og z værdien i y.
>>> x = y = z = 0
>>> x
0
>>> y
0
>>> z
0
```

BEMÆRK: I Python anvendes == for lig med og = for tildel. Der er således markant forskel på x == y og x = y

2.5.2. Variabler kan overskrives

Globale variabler kan overskrives:

```
>>> e = 1000 # definering af global variabel
>>> def nix():
...     global e
...     e = 25
...
>>> e
1000
>>> nix()
>>> e
25
```

Globale navne er flere ting, så pas på:

```
>>> liste = [1,2,3,4,5]
>>> def funktion(l):
...     global liste
...     liste.append(12345)
...
>>> liste
[1, 2, 3, 4, 5]
>>> funktion(liste)
>>> liste
[1, 2, 3, 4, 5, 12345]
>>>
```

Indledningsvis oprettes listen [1,2,3,4,5]. Det første kald viser, at såfremt listen kaldes, inden den i funktionen funktion erklærede globale liste overtager 1: den udenfor funktionen definerede liste og 2: tildeler den et element (12345) yderligere. Her markeres det, at en global variabel erklæret i en funktion overstyrer en global variabel erklæret udenfor funktionen.

2.5.3. Variablers adresse

Variablers type kan findes med type:

```
>>> a = 25
>>> type(a)
<type 'int'>
>>> a = 12.4
>>> type(a)
<type 'float'>
>>> a = "Rønne"
>>> type(a)
<type 'str'>
```

2.5.4. Variablers adresse

Variablers adresse i RAM kan findes med id

```
>>> a = 12.4
>>> id(a)
136095956
>>> a = 23
>>> id(a)
135591272
>>> a = "Rønne"
>>> id(a)
1078892096
```

Af følgende udskrift fremgår det, at samme bogstav giver samme adresse:

```
>>> s = "Dette er en streng."
>>> for i in range(0,18):
...     print s[i], id(s[i])
...
D 1078892160
e 1076630176
t 1076645984
t 1076645984
e 1076630176
  1078892192
e 1076630176
r 1076518048
  1078892192
e 1076630176
n 1076614816
  1078892192
s 1076613536
t 1076645984
r 1076518048
e 1076630176
n 1076614816
g 1076690208
```

```
>>> s = "AABBCCaabbcc"
>>> for i in range(0,len(s)):
...     print s[i], id(s[i])
...
```

```
A 1078892448
A 1078892448
B 1078892384
B 1078892384
C 1078892608
C 1078892608
a 1076679360
a 1076679360
b 1076679584
b 1076679584
c 1076690176
c 1076690176
```

2.5.5. Variabler i Python

At Python er et ret nyt computsprog, det markerer sig bl.a. m.h.t. variablers egenskaber og anvendelsesmåde. I de traditionelle computersprog som C, C++, Pascal, Delphi, Visual Basic, Java og andre skal variable være det man kalder strongly typed dvs. de skal defineres til at indeholde ganske bestemte værdityper f.eks. heltal eller tekststrenge, og kun de således definerede typer må gemmes i de aktuelle variabler. Sådan er det ikke i Python. Her flyttes variablers indhold automatisk til andre adresser i lageret, hvis omdefinering er nødvendig, bl.a. fordi alle former for variabler ikke fylder lige meget i lageret.

Eks. 1:

```
>>> # Her sker der 2 ting: 1: variablen a erklæres (is declared) og
>>> # 2: tildeles samtidig værdien "Tekststreng" d.v.s. den sættes til
>>> # at pege på de adresser i computerens lager, hvor "Tekststreng"
>>> # opbevares.
>>> a = "Tekststreng"
>>> a
Tekststreng
>>> # Nu tildeles samme variabel uden videre et heltal (integer) som værdi.
>>> a = 25
>>> print a
25
```

Eks. 2:

```
>>> a = "Tekststreng " + str(25)
>>> a
'Tekststreng 25'
```

Python er ikke strongly typed som C, Pascal og mon ikke de fleste andre computersprogs variabler skal være det. At et sprog er strong typed betyder, at en variabel ene og alene kan indeholde den type af værdi, den er oprettet (defineret) til at indeholde. C, Pascal og de andre sprog holder kort sagt styr på de adresser i computerens RAM lager, variablerne er sat til at pege på.

2.6. Boolske udtryk/variabler

Boolske udtryk/variabler kan indtage/tildeles 2 værdier og kun 2. De 2 værdier er sand (true) og falsk (false). De 2 værdier burde i alle computersprog repræsenteres med 0 for falsk og 1 for sand, men sådan er det desværre ikke altid. Til alt held benytter Python også her det logiske altså: 0 for falsk og 1 for sand, hvad følgende eksempel kan påvise:

Eks. 1:

```
>>> 2 == 3
0
>>> 2 == 2
1
```

Udtryk & Resultat

- true and true & true
- true and false & false
- false and true & false
- false and false & false
- not true & false
- not false & true
- true or true & true
- true or false & true
- false or true & true
- false or false & false

Eks. 2 Vi begynder med at lade en variabel, pege på en adresse i lageret, hvor værdien 5 lagres. Af nemhedsgrunde vælger vi at give variablen det absolut intetsigende navn a. Det var en ordentlig smøre for at beskrive, hvad der reelt sker. Når man ved, hvad det, der sker og, for at det hele ikke skal blive absurd og mekanisk, kan vi tillade os at sige, at vi tildeler variablen a værdien 5, værdien b værdien 7 og variablen c værdien 9. De samme forhold bør altid gælde for den seriøse programmør. Man kan tildele en variabel en værdi f.eks. a = 25 og b = 15 + 10 med tildelingstegnet = og eksempelvis betinge, at såfremt

den værdi en variabel peger på opfylder de samme betingelser som f.eks. indeholder samme talsum, som i dette tilfælde så er a == b.

Nu kan vi gå videre og teste de tildelte værdiers sandhedsværdier i matematiske udtryk:

```
>>> a == 5
1
>>> a == 7
0
>>> b == 7
1
>>> b == 5
0
>>> a == 6 and b == 7
0
>>> a == 7 and b == 7
0
>>> not a == 7 and b == 7
1
>>> a == 7 or b == 7
1
>>> a == 7 or b == 6
0
>>> a == 6 or b == 7
1
>>> not ( a == 7 and b == 6)
1
```

2.7. Indskriv interaktivt

I interaktiv mode kan tal indskrives med 2 forskellige funktioner: raw_input, hvor tal modtages som var det tekst, og input der modtager tal som tal. I førstnævnte vil den modtagne værdi ofte skulle konverteres til tal, ellers kan det give uheldige resultater eks:

```
>>> tal = raw_input("Skriv et tal: " )
Skriv et tal: 730
>>> tal * 4
'730730730730'
>>>
Det går langt bedre med:
>>> tal = input("Skriv et tal: " )
Skriv et tal: 730
>>> tal
730
```

```
integer = raw_input( "Skriv et helt tal:\n" )
integer = int( integer )
if integer < 0:
 print "%d er mindre end nul" % integer
else:
 print "%d er større end nul" % integer
```

```
>>> # Fra tekst til tal
>>> heltal = raw_input("Skriv et helt tal: ")
Skriv et helt tal: 25
>>> "Det indtastede tal var " + heltal
'Det indtastede tal var 25'
```

```
>>> # Fra tal til tekst
>>> heltal = input("Skriv et helt tal: ")
Skriv et helt tal: 25
>>> # og konverteres til streng
>>> "Det indtastede tal var " + str(heltal)
'Det indtastede tal var 25'
```

2.8. Scripts

Hvis og når du kommer ud for at skulle udvikle større projekter i Python, bliver du nødt til at gemme din Python-kode som individuelle filer du kan bruge igen og igen. Hertil findes der adskillige tekst editorer til Linux at vælge imellem. (Indsæt her et link til beskrivelse af tekst editorer i FTAV). Brugen af editoren er slet ikke så svært, som det kan virke i første omgang, hvad du let kan konstatere, hvis du har skrevet et eksempel i immediate mode, så marker hele teksten og kopier det hele til den editor, du vil bruge. Når du har koden inde i editoren, så lad editoren foretage en find og erstat og fjern alle

```
>>>
```

meldingerne. Hvis du har bevaret koden med de tilhørende blokkoloner m.v., så er det, du nu har tilbage et script (Pythons benævnelse af et program). Scriptet kan du gemme under et lovligt navn og med typebetegnelsen py som eksempelvis mitScript.py. Herefter kan du afvikle (køre) programmet lige så tit du lyster. Desuden er det meget let at udvide og ændre. Det sker ved at hente det ind i editoren igen og skrive videre på det. Det er naturligvis ikke meningen, at du hver gang skal begynde dine projekter i interaktiv mode for derfra at flytte dem ud i editoren og fortsætte der. De større projekter er som antydet bøvlede at behandle i interaktiv mode. Der skriver du det hele i editoren og gemmer det på nævnte måde. Når du har skrevet et program (et script) og ønsker at afvikle det, sker det ved fra det bibliotek, hvori skriptet findes at taste python programmetsNavn.py

I Linux kan Python-scripts gøres direkte kørbare ved at indsætte

```
#! /usr/bin/env python
```

som første linje i programmet. "#!" (havelåge og udråbstegn) SKAL være de to første tegn i programfilen. Scriptet kan tildeles en kørbar mode eller tilladelse ved anvendelse af kommandoen:

```
$ chmod +x myscript.py
```

Det er muligt at anvende andre styrekoder end ASCII i Python kildetekster. Den bedste fremgangsmåde til udførelse af dette er at indsætte endnu en speciel kommandolinje umiddelbart efter den allerede viste, således:

```
#! /usr/bin/env python
# -*- coding: iso-8859-1 -*-
```

Hvis du ønsker at afvikle en opstartsfil mere fra det aktuelle bibliotek, kan du skrive en global opstartsfil ved at bruge koden:

```
if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py')
```

Hvis du ønsker at anvende opstartsfilen i et program, må programmet yderligere indeholde følgende kode (aktuelt tilpasset til dit program):

```
import os
filnavn = os.environ.get('PYTHONSTARTUP')
if filnavn and os.path.isfile(filnavn):
    execfile(filnavn)
```

2.9. If, elif og else

Her er et eksempel på hvorledes man kan teste for forskellige betingelser, og få ens funktioner til at gøre nogle forskellige ting.

```
>>> def findMax(a,b,c):
...     max = a
...     if b > a:
...             max = b
...     if c > b:
...            max = c
...     return max
...
>>> findMax(37, 2 * 19 ,43)
43
```

```
>>> tal = 17 ** 4
>>> if tal < 50000:
...     print "Tallet er mindre end 50000"
... else:
...     print "Tallet er",tal
...
Tallet er 83521
```

If sætningen er måske den bedst kendte sætning (statement)

```
>>> x = int(raw_input("Skriv et helt tal: "))
>>> if x < 0:
...      x = 0
...      print 'Negativt tal ændret til nul'
... elif x == 0:
...      print 'Nul'
... elif x == 1:
...      print 'Et'
... else:
...      print 'Større end 1'
...
```

Der kan være nul eller flere elif blokke, else kan vælges/udelades efter ønske. Nøgleordet "elif" er en forkortelse af "else if" og er praktisk, da der derved er muligt at undgå alt for mange blokindrykninger (de fylder enormt i bredden).

```
>>> streng = "wqexmjupiolåærdtce45rft6g8ijuplbkilmæø3we45rft6g8ijupl"
>>> if len(streng) > 50:
...     print "Længden af strengen er større end 50"
... elif len(streng) > 45:
...     print "Længden af strengen er større end 45"
... elif len(streng) > 40:
...     print "Længden af strengen er større end 40"
... else:
...     print "Længden af strengen er:", len(streng)
...
Længden af strengen er større end 50
```

Simulering af terningekast: ALLE variabler i Python SKAL være erklærede, inden de bruges første gang, I det følgende eksempel skal der bruges 6 fordefinerede variabler. Her har Python en speciel og særdeles konstruktiv måde at gøre det på:

```
f6 = f5 = f4 = f3 = f2 = f1 = 0
I andre om ikke alle andre computersprog skulle det gøres nogenlunde således:
f1 = 0
f2 = 0
f3 = 0
f4 = 0
f5 = 0
f6 = 0
Simulering af terningekast:
>>> import random
>>> f6 = f5 = f4 = f3 = f2 = f1 = 0
>>> for kast in range( 1, 10001 ):       # 10.000 terningekast
...     udfald = random.randrange( 1, 7 )
...     if udfald == 1:
...             f1 += 1
...     elif udfald == 2:
...             f2 += 1
...     elif udfald == 3:
...             f3 += 1
...     elif udfald == 4:
...             f4 += 1
...     elif udfald == 5:
...             f5 += 1
...     else:
...             f6 += 1
print "Udfald:"
print "Antal enere:  ", f1
print "Antal toere:  ", f2
print "Antal treere:  ", f3
print "Antal firere:  ", f4
print "Antal femmere: " ,f5
print "Antal seksere: ", f6
import random
def kastMed2():
   terning1 = random.randrange( 1, 7 )
   terning2 = random.randrange( 1, 7 )
   sumTotal = terning1 + terning2
   print "Spiller slog%d + %d = %d" % ( terning1, terning2, sumTotal )
   return sumTotal
sum = kastMed2()       # første omgang kast
if sum == 7 or sum == 11:
   gameStatus = "VANDT"
elif sum == 2 or sum == 3 or sum == 12:
   gameStatus = "TABTE"
else:                                    # husker points
   gameStatus = "FORTSÆT"
   minePoints = sum
   print "Spillers points er ", minePoints
while gameStatus == "FORTSÆT":   # fortsæt spillet
```

```
   sum = kastMed2()
   if sum == minePoints:              # vandt med følgende resultat
      gameStatus = "VANDT"
   elif sum == 7:                   # vandt ved at slå/kaste summen 7
      gameStatus = "TABTE"
if gameStatus == "VANDT":
   print "Spilleren vandt"
else:
   print "Spilleren tabte"
```

2.10. Break og continue

```
>>> for i in range(1,101):
...     print i,
...     if i == 6:
...             break
...
1 2 3 4 5 6
```

2.11. Løkker

En computer er god til at huske (hvis ikke strømmen afbrydes) og god til at gentage. Når en kommando ønskes gentaget, kan det ske i en eller anden form for løkke, der køres i ring sådan at fortolkningen begynder umiddelbart efter blokkolon, fortsætter blokken ud og gentager samme proces til kommandoen i løkkens øverste linje er opfyldt. Her vil vi se på løkkestrukturer.

2.12. While

```
>>> i = 0  # i skal være defineret inden løkken gennemløbes
>>> while i < 11:
...     print i,
...     i += 1
...
0 1 2 3 4 5 6 7 8 9 10
```

Fibonacci tal Fibonaccis tal er en uendelig talrække, hvor det enkelte tal fremkommer som summen af de to foregående eks: 1, 1, 2,3,5,8,13...

```
>>> def fib(n):
...     a, b = 0, 1
...     while b < n:
...             print b,
...             a, b = b, a + b
...
>>> fib(100)
1 1 2 3 5 8 13 21 34 55 89
>>> def fib2(n):
...     resultat = []
...     a, b = 0, 1
...     while b < n:
...             resultat.append(b)
...             a,b = b, a + b
...     return resultat
...
>>> fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

2.13. For løkker

Se også under range. Alle range løkker med 1 eller flere parametre kan anvendes i forbindelse med for løkker.

```
>>> for i in range(1,11):
...      print i
...
1 2 3 4 5 6 7 8 9 10
```

```
Kartotek = []     # liste af Pythonbøger
print "Indskriv bogtitel.\n"
for i in range( 5 ):
   titel = raw_input( "Indskriv bog %d: " % ( i + 1 ) )
   Kartotek.append( titel )
print "\nVis indhold"
for i in range( len( Kartotek ) ):
   Indskriv bogtitel.
```

```
Indskriv bog 1: "Programming Python"
Indskriv bog 2: "Python Cookbook"
Indskriv bog 3: "Python How To Program"
Indskriv bog 4: "Jython Essentials"
Indskriv bog 5: "Learn to Program"


Vis indhold
        1     "Programming Python"
        2     "Python Cookbook"
        3     "Python How To Program"
        4     "Jython Essentials"
        5     "Learn to Program"
```

Kvadrattal:

```
>>> def Kvadrat(k):
...     return k * k
...
>>> for i in range(1,11):
...     print Kvadrat(i),
...
1 4 9 16 25 36 49 64 81 100
```

Tilfældige tal:

```
>>> import random
>>> for i in range( 1, 21 ):
...     print random.randrange( 1, 7 )
...
5 2 2 3 1 5 4 5 3 4 2 1 2 5 6 6 1 3 4 6
```

Mål længden af nogle tekststrenge:

```
>>> a = ['Ugleenge', 'Murergade', 'Galløkken']
for l in a:
        print l, len(l)
>>>
Ugleenge 8
Murergade 9
Galløkken 9
>>>
```

For-løkker og slicing:

```
>>> for x in a[:]: # opret en slice copy af den aktuelle liste:
>>>     if len(l) > 6: a.insert(0, l)
>>> a
['Galløkken', 'Ugleenge', 'Murergade', 'Galløkken']
>>>
>>> for i in range(len(a)):
>>>         print i, a[i]...
...
0 Galløkken
1 Ugleenge
2 Murergade
3 Galløkken
>>>
```

2.14. Pass sætningen

pass sætningen bruges til at gøre ingen ting. Kan bruges når en sætning syntaktisk er nødvendig, mens programmet ikke kræver nogen aktion. For eksempel:

```
Eks. 1
>>> class Klasse:
        pass...
...
>>>
```

```
Eks. 2
while True:
 pass # vent på tastaturklik
```

2.15. Funktionsdefinering og -kald

Under udviklingen af et program kan der være brug for en hensigtsmæssig måde at opdele programmet på, så vi kan nøjes med at beskæftige os med en del af programmet ad gangen. Vi kan ønske at putte mere trivielle rutiner hen i et hjørne af programmet, så de ikke forstyrrer os i den øvrige programskrivning. Her er funktioner (functions) eller som de samme også ofte kaldes metoder (methods) en velkendt hjælp i de fleste computersprog om ikke alle de nuværende. I det oprindelige BASIC kendtes

begrebet ikke, hvorfor vi brugte "GOTO" i massevis. I Pascal og Delphi er der ud over funktionerne nogle til funktioner forholdsvis svarende "pakkesamlere", der kaldes procedurer. Procedurer kan returnere en værdi, men behøver det ikke. Alt efter computersprog kan en funktion ubetinget skulle returnere en værdi (gælder i C++), i andre computersprog som Pascal, Delphi, Visual Basic og Python kan en funktion returnere en værdi, men behøver det ikke nødvendigvis. Python funktioner bliver derfor, hvad C++ programmører nok vil kalde en procedure. Men lad os nu skrive en Python funktion af hver slags - først en funktion, der returnerer en værdi (altså den "rigtige" funktion).

Når du er færdig med defineringen skal du trykker 2 gange på Enter-tasten dels for at fortælle Python, at defineringen er færdig og dels for at komme tilbage til interaktiv mode markeringen

```
>>>
Funktionsdefinering:
>>> def funktion():-
...     a = 4 * 5
...     return a
...
Funktionskald:
>>> funktion()
20
```

Som du kan se ovenfor, skal nøgleordet def bruges for at fortælle Python, at det kommende er en funktionsdefinering. Efter nøgleordet følger funktionsnavnet, der kan være alle lovlige navne som i variabler. Efter funktionsnavnet følger en parameterliste, der som her kan være tom. Den omgives af runde parenteser og efterfølges af det kolon, som du med garanti vil glemme mange gange, inden du er inde i rutinen med at fortælle Python, at det kommende er en sammenhørende blok (her funktionskroppen). Python reagerer ved at rykke de efterfølgende programlinjer en tabulatorbredde til højre. Hvis du kopierer kode fra eksempelvis en editor til Python, er det nødvendigt, at få nævnte indrykninger med evt. ved selv at indsætte dem ved at trykke på tastaturets tabulatortast. Der er ofte flere blokke i en funktion. Er der det, skal de programlinjer, der hører til den pågældende blok rykkes endnu et tabulatorstop til højre.

Returværdien kan også indgå i eksempelvis en anden regneoperation:

```
>>> print funktion() * 7
140
```

Funktion med et parameter (kaldes undertiden et argument):

>>> def f(a): ... return a * 1.25 ... >>> f(75) 93.75

Funktion med mere end et parameter:

>>> def funktionsnavn(s1,s2): ... return s1 + s2

Funktionskald:

>>> funktionsnavn("Streng1 ", "streng2.") 'Streng1 streng2.

Samme funktion, men ved tildeling af numeriske værdier:

>>> funktionsnavn(10,20.50) 30.5

Brugerdefineret funktion kalder anden brugerdefineret funktion:

>>> def f1(): ... print f2() ... >>> def f2(): ... print "Funktion f2 er blevet kaldt." ... >>> f1() Funktion f2 er blevet kaldt. None

Brugerdefineret funktion kalder fordefineret funktion:

>>> from sys import exit >>> def afslut(): ... sys.exit() ... >>> afslut() ajbo@linux:~>

>>> m = input("Skriv denne maaneds nummer: ") Skriv nummeret (pladsen i kalenderen) for denne maaned: 12 >>> maanedsnavne = ["januar","februar","marts","april","maj","juni","juli",\ ... "august","september","oktober","november","december"] >>> if 1 <= m <= 12: ... print "Denne maaneds navn er",maanedsnavne[m - 1]

Denne maaneds navn er december

Liste kan modtages som argument i funktion: >>> liste [] #opretter tom global liste >>> def ul(l): # argumentet l erklæres her som lokalt navn ... liste.append(l) # udvider den globale liste ... >>> ul(1) >>> ul("Hasle") >>> ul(2) >>> ul("Nyker") Kørselsresultat: >>> liste [1, 'Hasle', 2, 'Nyker']

>>> liste = [1,2,3,4] >>> def udvL(l = []): # argumentet l erklæres her som lokalt navn ... for i in range(5,11): ... l.append(i) ... print l ... Kørselsresultat: >>> udvL(liste) [1, 2, 3, 4, 5] [1, 2, 3, 4, 5, 6] [1, 2, 3, 4, 5, 6, 7] [1, 2, 3, 4, 5, 6, 7, 8] [1, 2, 3, 4, 5, 6, 7, 8, 9] [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] >>>

def kR(lengde = 1, bredde = 1, hojde= 1): return lengde * bredde * hojde

print "Forvalgte kasserumfang:", kR() print "Rumfang af kasse med længden 10 er: ", kR(10) print "Rumfang af kasse med højden 10 er: ", kR(1 ,10) print "Rumfang af kasse med længde og højde 10 er:

", kR(10 ,10) print "Rumfang af kasse med længde 12, bredde 4 og højde 10 er: ", kR(12,4 ,10)

Find primtal: >>> for i in range(2, 10): ... for j in range(2, i): ... if i % j == 0: ... print i, 'er lig med', j, '*', i/j ... break ... else: ... print i, 'er et primtal' ... 3 er et primtal 4 er lig med 2 * 2 5 er et primtal 5 er et primtal 6 er lig med 2 * 3 7 er et primtal 7 er et primtal 7 er et primtal 7 er et primtal 8 er lig med 2 * 4 9 er et primtal 9 er lig med 3 * 3 >>>

Beregn fakultet ved rekursion: def fakultet(tal): if tal <= 1: return 1 else: return tal * fakultet(tal - 1) # rekursivt kald

for i in range(1,11): print "%2d! = %d" % (i, fakultet(i))

ajbo@linux:~> python fakultet.py 1! = 1 2! = 2 3! = 6 4! = 24 5! = 120 6! = 720 7! = 5040 8! = 40320 9! = 362880 10! = 3628800

```
>>> def F():
>>>  F()
...
>>> F.ekstra
23
>>>
```

```
def OK(prompt, rundgang=4, svar='Svar ja eller nej!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('JA', 'Ja', 'ja'): return 1
        if ok in ('NEJ', 'Nej', 'nej'): return 0
        rundgang = rundgang - 1
        if rundgang < 0: raise IOError, 'vanskelig bruger'
        print svar
Kan kaldes med eksempelvis:
OK('Ønsker du at afslutte?')
og:
OK('Ønskes filen lukket?', 2)
```

```
>>> i = 5
>>> def f(arg = i):
...     print i
...
```

```
>>> f()
5
>>>
```

```
>>> def f(a, L=[]):
    L.append(a)
    return L... ...
...
Et kørselsresultat:
>>> for i in range(10):
...     print f(i)
...
[0]
[0, 1]
[0, 1, 2]
[0, 1, 2, 3]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 8]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

```
>>> for i in range(10):
...     print f(i),
...
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
>>>
```

```
Returner variablen None:
>>> def f():
...     return
```

```
...
```

```
>>> print f(56)
None
>>>
```

Nøgleords argumenter (keyword arguments). Du kan vælge at lade din funktion have et antal fordefinerede og forud værditildelte nøgleord. Sådanne nøgleord må ikke forveksles med de i Python forud definerede variabler. Vi skal se et eksempel:

```
>>> def funktion(computer,kvalitet, styresystem = "Linux", computersprog = "Python"):
...       print "En",kvalitet,computer, "anvender", styresystem
...
>>> funktion("PC","veludstyret")
En veludstyret PC anvender Linux
>>>
```

Argumenterne computer og kvalitet er tomme, hvorfor de skal tildeles værdi ved funktionskald. Det skal nøgleordsargumenterne ikke, da de allerede er tildelt en sådan. Det samme gælder nøgleordet computersprog.

Som andre variabler, kan variablen kvalitet modtage en tom værdi, ellers kan du få et lidt ulogisk resultat som:

```
>>> funktion(0,0)
En 0 0 anvender naturligvis Linux
>>>
>>> funktion("PC","")
En  PC anvender Linux
>>>
>>> funktion("","veludstyret")
En veludstyret  anvender Linux
>>>
>>> funktion("PC","nogenlunde",styresystem = "Windows")
En nogenlunde PC anvender Windows
>>>
```

Når en funktions sidste parameter er af formen **navn (med 2 foranstillede stjerner), kan den modtage en ordliste, hvis nøgleord ikke findes i parameterlisten. Det kan kombineres med en formel parameter af formen *navn (med 1 foranstillet stjerne).*navn skal komme før **navn:

```
>>> def f(vareart, *argumenter, **noegleord):
...     pass
...
>>>
```

Funktionskald af den type bruges meget i Python, så det er vigtigt at forstå virkning m.v, hvorfor jeg viste første eksempel med danske navne, ellers er det mest praktist at holde sig til de engelske, da det er dem, du normalt vil se:

```
>>> def f(programnavn, *arguments, **keywords):
...     return programnavn, arguments, keywords
...
>>> f("mitProgram.py")
('mitProgram.py', (), {})
>>>
```

Som du ser, returneres der en tuple indeholdende 1: argumentet 2: en tuple og 3: en ordliste. Hvis du ønsker at læse den eksterne fil mitProgram.py, kan du anvende følgende indledning:

```
>>> f("mitProgram.py","r")
('mitProgram.py', ('r',), {})
>>>
```

>>> f("mitProgram.py","r","r2","r3") ('mitProgram.py', ('r', 'r2', 'r3'), {}) >>>

>>> f("mitProgram.py","r","r2","r3", regnskabsaar = 2003) ('mitProgram.py', ('r', 'r2', 'r3'), {'regnskabsaar': 2003}) >>>

>>> f("mitProgram.py","r","r2","r3",regnskabsmaaned = "januar") ('mitProgram.py', ('r', 'r2', 'r3'), {'maaned': 'januar'}) >>> >>> f("mitProgram.py","r","r2","r3",maaned = "februar") ('mitProgram.py', ('r', 'r2', 'r3'), {'maaned': 'februar'}) >>> >>> f("mitProgram.py","r","r2","r3",maaned = "januar",md2 = "februar") ('mitProgram.py', ('r', 'r2', 'r3'), {'md2': 'februar', 'md': 'januar'}) >>>

Argumentlisten behøver ikke nødvendigvis at være af et omfang som en dansk stil:

>>> def f(v,*a, **k): ... return v,a,k ... >>> f("Vareart","cykel",navn = "Christiania") ('Vareart', ('cykel',), {'navn': 'Christiania'}) >>> >>> f("Vareart","cykel",navn2 = "Christiania") ('Vareart', ('cykel',), {'navn2': 'Christiania'}) >>>

>>> def f(v,*a, **k): ... print a, k ... >>> f("Vareart","cykel",navn = "Christiania") (’cykel’,) {’navn’: ’Christiania’}

>>> def f(a,*b,**c): ... c = {1:"en",2:"to",3:"tre"} ... return a,b,c ... >>> f(1,2) (1, (2,), {1: ’en’, 2: ’to’, 3: ’tre’}) >>>

Bemærk: boglisten kræver de krøllede parenteser: >>> def f(a,*b,**c): ... return a,b,c ... ...

>>> f(1,"fire","fem","seks",7:"syv",8:"otte",9:"ni") File "<stdin>", line 1 f(1,"fire","fem","seks",7:"syv",8:"otte",9:"ni") ^ SyntaxError: invalid syntax >>> f(1,"fire","fem","seks",{7:"syv",8:"otte",9:"ni"}) (1, (’fire’, ’fem’, ’seks’, {8: ’otte’, 9: ’ni’, 7: ’syv’}), {}) >>>

>>> def f(a,*b,**c): ... for arg in b : print arg ... >>> f(1,"to","tre","fire") to tre fire >>>

>>> args = [1,10] >>> range(*args) [1, 2, 3, 4, 5, 6, 7, 8, 9] >>>

>>> args = [3, 6] >>> range(*args) [3, 4, 5]

```
Lamda arbejder i baggrunden:
>>> def svindel(n):
...     return lambda x: x + n
...
>>> f = svindel(1.25)
>>> f(100)
101.25
>>>
```

2.16. Lister

Lister er variabler med nul til flere sammenhørende rum. Listen kendes på dens firkantede parenteser. Tænker du dig dem taget væk, har du en tuple. Den kendes på dens kommaer.

```
Listens og dens muligheder:
>>> # Opret tom liste
>>> listen = []
>>> # udvid listen med et ekstra element
>>> listen.append("Ypnasted")
>>> # udvidelsen kan også ske således:
```

```
>>> listen[len(listen):] = "Teglkaas"
>>> listen[len(listen):] = ["Hellig Peder"]
>>> listen
['Ypnasted', 'T', 'e', 'g', 'l', 'k', 'a', 'a', 's', 'Hellig Peder']
>>> # udvidelsen kan også ske således:
>>> listen.extend(["Vang","Hammeren"])
>>> listen
['Ypnasted', 'T', 'e', 'g', 'l', 'k', 'a', 'a', 's', 'Hellig Peder', 'Vang', 'Hammeren']
>>> # Listen kan vendes om:
>>> listen.reverse()
>>> listen
['Hammeren', 'Vang', 'Hellig Peder', 's', 'a', 'a', 'k', 'l', 'g', 'e', 'T', 'Ypnasted']
>>> # fjerner sidste element i listen:
>>> listen.pop()
'Ypnasted'
>>> listen
['Hammeren', 'Vang', 'Hellig Peder', 's', 'a', 'a', 'k', 'l', 'g', 'e', 'T']
>>> for i in range(0,8):
...     listen.pop()
...
'T'
'e'
'g'
'l'
'k'
'a'
'a'
's'
>>> listen
['Hammeren', 'Vang', 'Hellig Peder']
>>> # sorterer listen
>>> listen.sort()
>>> listen
['Hammeren', 'Hellig Peder', 'Vang']
>>> # fjerner angivne element fra listen:
>>> listen.remove("Hellig Peder")
>>> listen
['Hammeren', 'Vang']
>>> # returnerer givne indeks (plads) i listen
>>> listen.index("Hammeren")
0
>>> listen.index("Vang")
1
>>> # finder listens største element:
>>> max(listen)
'Vang'
>>> # finder listens mindste element:
>>> min(listen)
'Hammeren'
>>> # opdeler listen i enkeltelementer
>>> zip(listen,[1,2])
[('Hammeren', 1), ('Vang', 2)]
>>> zip(listen,[8,7])
```

```
[('Hammeren', 8), ('Vang', 7)]
>>> # multiplicerer listen:
>>> lister = listen * 2
>>> lister
['Hammeren', 'Vang', 'Hammeren', 'Vang']
>>> # multiplicerer listen:
>>> lister = zip(listen,[1,2]) * 2
>>> lister
[('Hammeren', 1), ('Vang', 2), ('Hammeren', 1), ('Vang', 2)]
>>>
```

Returnering af større eller mindre dele af en liste kaldes slicing. Slicing af en given del af en liste sker ved at indsætte start- og slutindeks i de fra listen velkendte firkantede parenteser:

>>> l = [1,2,3,3,4,5,6,7,8,9] >>> print l[0] # returner listens første element 1 >>> print l[0:4] # returner listens 4 første elementer [1, 2, 3, 3] >>> print l[3:6] # returner listens 4. til 6. element [3, 4, 5] >>> print l[-4] # returner listens 4. sidste element 6 >>> print l[-1] # returner listens sidste element 9 >>>

Opret liste med lige tal <= 20 # Søg i liste efter integer (heltal) liste = range(0, 21, 2) #listeindhold: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20] print liste skalFindes = int(raw_input("Indtast heltal <= 20: ")) if skalFindes in liste: print "Fundet på indeks:", liste.index(skalFindes) else: print "Elementet blev ikke fundet"

Indtast heltal <= 20: 12 Fundet på indeks: 6 Indtast heltal <= 20: 13 Elementet blev ikke fundet

Listen og tuplen har flere lighedspunkter: >>> l = [] >>> t = { } >>> t = 1,2,3,5 >>> t (1, 2, 3, 5) >>> l = t >>> l (1, 2, 3, 5) >>> l[1] 2 >>> t[1] 2 >>> l = [6,7,8,9] >>> l [6, 7, 8, 9] >>> t = l >>> t [6, 7, 8, 9]

liste = [] # opret tom liste # indsæt elementer i listen for indeks in range(1, 11): liste += [indeks]

print "Listens indhold:", liste

print # indsætter tom linje

for element in liste: print element,

print

listetilgang via indeks (rumnummer) print "\nVælg elementer efter deres indeks:" print "Listens indhold:"

for i in range(len(liste)): print "%6d %3d" % (i, liste[i])

print "\nOpdatering af listeelementer..." print "Listens indhold før opdateringen:", liste liste[0] = -100 liste[-3] = "bornholmere" print "Listens indhold efter opdateringen:", liste

ajbo@linux:~> python anvendt_liste.py Listens indhold: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

1 2 3 4 5 6 7 8 9 10

Vælg elementer efter deres indeks: Listens indhold: 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10

Opdatering af listeelementer... Listens indhold før opdateringen: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Listens indhold efter opdateringen: [-100, 2, 3, 4, 5, 6, 7, 'bornholmere', 9, 10]

liste = [] # opretter tom liste

indsæt 10 heltal via brugerindtastninger print "Skriv 10 heltal:"

for i in range(10): nytElement = int(raw_input("Skriv helt tal: %d: " % (i + 1))) liste += [nytElement]

Udskrift af lister i liste: liste = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

print "Elementerne i listen:" for i in liste: for element in i: print element, print

Kørselsresultat: python row.py Elementerne i listen: 1 2 3 4 5 6 7 8 9

>>> liste = range(11) >>> liste [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Lister kan flettes (indeholde andre lister) f.eks: >>> liste = [[]] >>> liste [[]] >>> liste * 3 [[], [], []]

>>> liste_1 = [2, 3] >>> liste_2 = [1,liste_1, 4] >>> len(liste_2) 3 >>> liste_2[1] [2, 3] >>> liste_2[1][0] 2 >>> liste_2[1].append('ekstrapost') >>> liste_2 [1, [2, 3, 'ekstrapost'], 4] >>> liste_1 [2, 3, 'ekstrapost'] >>>

>>> stack = [3, 4, 5] >>> stack.append(6) >>> stack.append(7) >>> stack [3, 4, 5, 6, 7] >>> stack.pop() 7 >>> stack [3, 4, 5, 6] >>> stack.pop() 6 >>> stack.pop() 5 >>> stack [3, 4]

>>> liste = ["Ugleenge","Sæne","Bækkely","Stampen"] >>> for i, v in enumerate(liste): ... print i,v ... 0 Ugleenge 1 Sæne 2 Bækkely 3 Stampen >>>

Der kan også dannes løkke over to eller flere samtidige sekvenser: >>> person = ’Ole’, ’120’, ’naturen’] >>> svar = ’navn:’, ’alder:’, ’hobby:’] >>> for i, j in zip(person,svar): ... print j,i ... navn: Ole alder: 120 hobby: naturen >>>

```
Lister kan sammenlignes:
>>> [1, 2, 3] < [1, 2, 4]
True
>>>
>>> [1, 2, 3, 4] < [1, 2, 4]
True
>>>
>>> [2,3,4] <> [2.0,3.0,4.0]
True
>>>
>>> [1, 2] < [1, 2, -1]
True
>>>
>>> [1, 2, 3] == [1.0, 2.0, 3.0]
True
>>>
>>> [1, 2, 3,4] <>[1, 2, 3,"p"]
True
>>>
>>> [1, 2, ['aa', 'ab']] < [1, 2, ['abc', 'a'], 4]
```

Lister anvendt som stakke

I computersproget FORTH benyttes begrebet stakke meget. Der sammenligner man normalt en stak med en stabel tallerkener. I en sådan stabel tallerkener, vil den nederste under normale forhold blive placeret i bunden at stakken eller stabelen. I FORTH arbejder man normalt med begreberne LIFO (last in first out eller på dansk først ind (altså øverst i tallerkenstabelen) first out eller først ud betyder, som ordet siger først ud - altså at man tager den øverste tallerken (det øverste eller sidst placerede element først. Modsvarende LIFO bruger man også FIFO i FORTH. Forkortelsen FIFO står for first in first out eller på dansk først ind først ud. Overført til tallerkenstablen betyder det, at den nederste tallerken i stakken, er den tjeneren benytter først. Jeg har vagt at forklare princippet ud fra FORTH, fordi det derved også bliver lettere at forstå, hvordan man i Python kan lave noget tilsvarende. I Python kalder man en stak for en

queue ligesom normalt må nøjes med at udskrive selve FIFO og LIFO elementerne. Det kan gøres således:

```
>>> queue = ["Arnager","Dueodde","Nexø","Svaneke"]
>>> queue.pop(0)
'Arnager'
>>> queue.pop(2)
'Svaneke'
>>> queue.append("Gudhjem")
>>> queue.pop(2)
'Gudhjem'
>>> queue.append("Tejn")
>>> queue.pop(2)
'Tejn'
>>>
```

2.16.1. Funktionelle programmeringsværktøjer

I Python er der tre fordefinerede funktioner, der er meget nyttige i forbindelse med lister:

`filter(), map(), and reduce().`

Eksempel: Syntaks: filter(funktion,sekvens) returnerer om muligt en sekvens af samme type som den i filters parameterliste. Den returnerede sekvens består af de værdier, der gør, at funktion(argument) er sand (true).

Eksempel:

```
Beregn Primtal:
Syntaks: filter(funktion, blok)
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

Syntaks: map(funktion, blok)

kalder funktion(parameter) for hver enkelt af sekvensens elementer og returnerer en liste indeholdende returværdierne.

Eksempel: Beregn kubiktal: >>> def kubik(x): return x*x*x ... >>> map(kubik, range(1, 11)) [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000] >>>

map kan også tage flere sekvenser. >>> def kvadrat(x): return x*x ... >>> map(None, sekvens, map(kvadrat, sekvens)) [(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)] >>>

Eksempel: Syntaks: reduce(funktion, sekvens) >>> 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 55 >>>

>>> def sumTotal(x,y): return x + y ... >>> reduce(sumTotal, range(1, 11)) 55 >>>

>>> reduce(sumTotal, range(11) ...) 55 >>>

Nyhed i Python version 2.3: >>> navne = ['London','Paris','New Yorkbyer','Gudhjem'] >>> [byer.strip() for byer in navne] ['London', 'Paris', 'New Yorkbyer', 'Gudhjem'] >>>

>>> drenge = ["Ole","Per","Sofus","Nikolai"] >>> [navne.strip() for navne in drenge] ['Ole', 'Per', 'Sofus', 'Nikolai'] >>>

```
>>> lige =  [2, 4, 6]
>>> [3*x for x in lige]
[6, 12, 18]
>>>
>>> [3*x for x in lige if x > 3]
[12, 18]
>>>
>>> [3*x for x in lige if x < 2]
[]
>>>
>>> [[x,x**2] for x in lige]
[[2, 4], [4, 16], [6, 36]]
>>>
>>> lige = [2, 4, 6]
>>> blandede = [4, 3, -9]
>>> [x*y for x in lige for y in blandede]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>>
>>> [x+y for x in lige for y in blandede]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>>
>>> [lige[i] * blandede[i] for i in range(len(lige))]
[8, 12, -54]
>>>
>>> [str(round(355/113.0, i)) for i in range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
>>>
>>> [x**3 for x in range(5)]
```

```
[0, 1, 8, 27, 64]
>>>
```

Tuples består af et anatal værdier adskille med kommaer. Man siger, at tuples kendes på deres kommaer, mens listen kendes på dens firkantede parenteser. Eks:

```
>>> t = 1,2,3,4,5
>>> t
(1, 2, 3, 4, 5)
>>>
At de adskillende kommaer virkelig betyder noget ses her:
>>> varsel = "Fare forude!"
>>> len(varsel)
12
>>> varsel = "Fare forude!",
>>> len(varsel)
1
>>>
```

I det første lilfælde er "Fare forude!" en tekststreng, mens det i det sidste tilfælde er en tuple. Forskellen er ene og alene kommaet.

```
Tuplen og dens anvendelse oprettet ud fra brugerdata:
t = int( raw_input( "Skriv aktuelt timetal: " ) )
m = int( raw_input( "Skriv aktuelt minuttal: " ) )
s = int( raw_input( "Skriv aktuelt sekundtal: " ) )
```

ligeNu = t, m, s # opret tuple

print "Tuplens indhold er:", ligeNu

print "Antal sekunder siden midnat", \ (ligeNu[0] * 3600 + ligeNu[1] * 60 + ligeNu[2])

Kørselsresultat: ajbo@linux:~> python anvendt_tuple.py Skriv aktuelt timetal: 11 Skriv aktuelt minuttal: 17 Skriv aktuelt sekundtal: 13 Tuplens indhold er: (11, 17, 13) Antal sekunder siden midnat 40633

Strengs, listes og tuples anvendelse: strengen = "abcdefghij" tuplen = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) listen = ["I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX", "X"]

print "strengen: ", strengen print "tuplen: ", tuplen print "listen: ", listen

start = int(raw_input("Vælg startpunkt: ")) sidste = int(raw_input("Vælg slutpunkt: "))

print "\nstrengen[", start, ":", sidste, "] = ", \ strengen[start:sidste]

print "tuplen[", start, ":", sidste, "] = ", \ tuplen[start:sidste]

print "listen[", start, ":", sidste, "] = ", \ listen[start:sidste]

Kørselsresultat: strengen: abcdefghij tuplen: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) listen: ['I', 'II', 'III', 'IV', 'V', 'VI', 'VII', 'VIII', 'IX', 'X'] Vælg startpunkt: 0 Vælg slutpunkt: 6

strengen[0 : 6] = abcdef tuplen[0 : 6] = (1, 2, 3, 4, 5, 6) listen[0 : 6] = ['I', 'II', 'III', 'IV', 'V', 'VI'] ajbo@linux:~>

Tuples kan flettes (be nested): >>> t = 12345, 54321, 'hej!' >>> t[0] 12345 >>> t (12345, 54321, 'hej!')

>>> # Tuples kan flettes: >>> u = t, (1, 2, 3, 4, 5) >>> u ((12345, 54321, 'hej!'), (1, 2, 3, 4, 5))

Tuples kan sammenlignes: >>> (1, 2, 3) < (1, 2, 4) True >>> >>> (1, 2, 3, 4) < (1, 2, 4) True >>> >>> (2,3,4) <> (2.0,3.0,4.0) False >>> >>> (1, 2) < (1, 2, -1) True >>> >>> (1, 2, 3) == (1.0, 2.0, 3.0) True >>> >>> (1, 2, 3,4) <>(1, 2, 3,"p") True >>> >>> (1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4) True >>>

```
Sammenligning af Lister og tuples
>>> [1, 2, 3] < (1, 2, 4)
True
>>>
>>> [1, 2, 3, 4] < (1, 2, 4)
True
>>>
>>> [2,3,4] <> (2.0,3.0,4.0)
True
>>>
>>> [1, 2] < (1, 2, -1)
>>> True
>>>
[1, 2, 3] == (1.0, 2.0, 3.0)
True
>>>
>>> [1, 2, 3,4] <> (1, 2, 3,"p")
True
>>>
>>> [1, 2, ['aa', 'ab']] < (1, 2, ('abc', 'a'), 4)
True
>>>
```

```
ordliste = {} # opretter tom dictionary
print "Ordlisten indhold:", ordliste
```

postnumre = { "Rønne": 3700, "Allinge": 3770, "Gudhjem": 3780, "Nexø": 3730} print "\nAlle indsatte postnumre:", postnumre

access og ret eksistenende ordliste print "\nRønne postnummer:", postnumre["Rønne"] postnumre["Gudhjem"] = 3760 print "Gudhjems rigtige postnummer:", postnumre["Gudhjem"]

tilføj postnummer postnumre["Aakirkeby"] = 3720 print "\nOrdlistens postnumre efter rettelsen:" print postnumre

slet indgang fra ordlisten del postnumre["Allinge"] print "\nOrdlistens nuværende indhold:" print postnumre

Ordlisten indhold: {}

Alle indsatte postnumre: {'Nexø': 3730, 'Gudhjem': 3780, 'Rønne': 3700, 'Allinge': 3770}

Rønne postnummer: 3700 Gudhjems rigtige postnummer: 3760

Ordlistens postnumre efter rettelsen: {'Nexø': 3730, 'Aakirkeby': 3720, 'Gudhjem': 3760, 'Rønne': 3700, 'Allinge': 3770}

Ordlistens nuværende indhold: {'Nexø': 3730, 'Aakirkeby': 3720, 'Gudhjem': 3760, 'Rønne': 3700}

Her er et eksempel på anvendelse af en ordliste: >>> postnr = {'Nyker': 3700, 'Hasle': 3790} >>> postnr['Gudhjem'] = 3760 >>> postnr {'Hasle': 3790, 'Nyker': 3700, 'Gudhjem': 3760} >>> >>> del postnr["Hasle"] >>> postnr {'Nyker': 3700, 'Gudhjem': 3760} >>> >>> postnr {'Muleby': 3700, 'Nyker': 3700, 'Gudhjem': 3760} >>> >>> postnr.keys() ['Muleby', 'Nyker', 'Gudhjem'] >>> >>> postnr.has_key("Hasle") 0 >>> >>> postnr.has_key("Gudhjem") 1 >>>

Konstruktøren dict() opretter ordlister direkte fra en liste hvis elementer er tuples: >>> liste = [('Muleby', 3700), ('Nyker', 3700), ('Gudhjem', 3760)] >>> dict(liste) {'Nyker': 3700, 'Muleby': 3700,

’Gudhjem’: 3760} >>> eller: >>> dict([(’Muleby’, 3700), (’Nyker’, 3700), (’Gudhjem’, 3760)]) {’Nyker’: 3700, ’Muleby’: 3700, ’Gudhjem’: 3760} >>>

>>> liste = [] >>> for i in range(6): ... liste.append((str(i), i* i)) ... >>>

>>> liste = [(’0’, 0), (’1’, 1), (’2’, 4), (’3’, 9), (’4’, 16), (’5’, 25)] >>> dict(liste) {’1’: 1, ’0’: 0, ’3’: 9, ’2’: 4, ’5’: 25, ’4’: 16} >>>

Løkke teknikker: >>> t = "Ugleenge","Sæne","Bækkely","Stampen" >>> t (’Ugleenge’, ’S\xe6ne’, ’B\xe6kkely’, ’Stampen’) >>>

ordliste = {"Ugleenge" : 1,"Sæne" : 2,"Bækkely" :3,"Stampen" : 4} >>> for i in ordliste.items(): print i ... (’S\xe6ne’, 2) (’B\xe6kkely’, 3) (’Ugleenge’, 1) (’Stampen’, 4) >>>

kalender = { 1 : "Januar", 2 : "Februar", 3 : "Marts", 4 : "April", 5 : "Maj", 6 : "Juni", 7 : "Juli", 8 : "August", 9 : "September", 10 : "Oktober", 11 : "November", 12 : "December" }

print "Ordlistens indhold:" print kalender.items()

print "\nOrdlistens indgange er:" print kalender.keys()

print "\nOrdlistens elementer er:" print kalender.values()

print "\nFor løkke henter ordliste elementer:" for indgang in kalender.keys(): print "kalender[", indgang, "] =", kalender[indgang]

Kørselsresultat: ajbo@linux:~> python kalender.py Ordlistens indhold: [(1, ’Januar’), (2, ’Februar’), (3, ’Marts’), (4, ’April’), (5, ’Maj’), (6, ’Juni’), (7, ’Juli’), (8, ’August’), (9, ’September’), (10, ’Oktober’), (11, ’November’), (12, ’December’)]

Ordlistens indeks (indgange) er: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

Ordlistens elementer er: [’Januar’, ’Februar’, ’Marts’, ’April’, ’Maj’, ’Juni’, ’Juli’, ’August’, ’September’, ’Oktober’, ’November’, ’December’]

For løkke henter ordliste elementer: kalender[1] = Januar kalender[2] = Februar kalender[3] = Marts kalender[4] = April kalender[5] = Maj kalender[6] = Juni kalender[7] = Juli kalender[8] = August kalender[9] = September kalender[10] = Oktober kalender[11] = November kalender[12] = December ajbo@linux:~>

2.17. Escape sekvenser

```
\n  Ny linje
\t  Vandret tabulator. Flytter markøren til næste tabulatorstop
\r  Vognretur. Flytter markøren til linjens begyndelse
\b  Backspace. Flytter markøren 1 plads tilbage
\a  Systemklokke
\\  Backspace. Indsætter \
\"  Indsætter anførselstegn
\'  Indsætter enkelt anførselstegn
```

2.18. Talsystemer

I computersproget Forth, kan man benytte omkring 70 forskellige talsystemer afhængig af sprogversionen. Det talsystem vi bruger i vore dage et 10-talsystemet eller det decimale talsystem. Sådan har det ikke været altid verden over. For omkring 4000 år siden brugtes 60-talsystemet i områ det lige nordvest for Persiske Hav. Lad os lige se på tallet 4320. Det består af 4 tusinder, 3 hundreder, 2 tiere og 0 enere, hvilket de fleste sikkert tager som en selvfølgelighed, men er det nu også så selvfølgelig? Nej, det allerede nævnte har sikkert allerede røbet, at det slet ikke er så enkelt 10-talsystemet er nonsens for enhver computer. Den kan alene arbejde i 2-talsystemet. Det skyldes, at computeren, som det elektriske apparat den er, reelt kun kan reagere på, om der går en strøm i en ledning eller ikke. Hvis der går en strøm, har man vedtaget at registrere tilstanden med tallet 1. Hvis der ikke går nogen strøm, registreres det med tallet 0.

2 er grundtal i totalsystemet, som 10 er det i 10-talsystemet og n i n-talsystemet. Cifrene i et talsystem udgør altid det antal, talsystemets grundtal angiver. Grundtallet er normalt det, der giver talsystemet dets navn. I totalsystemet er der to cifre (nul og et). I titalsystemet er der ti cifre (fra og med 0 til og med 9) osv.

Fordi alle talsystemer har 0 som laveste ciffer, bliver højeste ciffer en mindre end talsystemets grundtal. I 2-talsystemet bliver det størst ciffer således 2 - 1 = 1 og i 10-talsystemet 10 - 1 = 9 og sådan fremdeles. Det bageste ciffer (det længst til højre) vil repræsentere ciffer gange grundtal i første. Et ciffer i et tal vil altid udgøre ciffer gange grundtal i n - 1. Men lad os lade Python gøre det lidt klarere:

2.18.1. Hexadeximale og oktale tal

I Python specificeres et hexadecimalt tal ved et foranstillet nul efterfulgt af et x og afsluttet med det hexadecimale tal som f.eks. 0x49 svarende til 9 + 4 * 16 = 73 i vores normale 10-talsystem. Jeg kommer ikke nærmere ind på det oktale talsystem (det med grundtallet 8), men skal nøjes med kort at nævne, at i det er 7 største ciffer x udelades. Det betyder, at 0x49 svarer til 0111 (1 ener 1 8-er og 1 64-er eller 8 **

2) altså 1 + 8 + 64 = 73 i det decimale talsystem. Det kan være svært for en begynder at forstå, så lad os se på et par eksempler:

Først et eksempel fra 10-talsystemet (decimaltal systemet)

```
>>> for i in range(0,9):
...     print "10 i",i,". er:", 10 ** i
...
10 i 0 . er: 1
10 i 1 . er: 10
10 i 2 . er: 100
10 i 3 . er: 1000
10 i 4 . er: 10000
10 i 5 . er: 100000
10 i 6 . er: 1000000
10 i 7 . er: 10000000
10 i 8 . er: 100000000
>>>

Det binære talsystem:
>>> for i in range(0,9):
...     print "2 i ",i,". er:", 2 ** i
...
2 i  0 . er: 1
2 i  1 . er: 2
2 i  2 . er: 4
2 i  3 . er: 8
2 i  4 . er: 16
2 i  5 . er: 32
2 i  6 . er: 64
2 i  7 . er: 128
2 i  8 . er: 256
>>>

Det oktale talsystem:
>>> for i in range(0,9):
...     print "8 i",i,". er:",8 ** i
...
8 i 0 . er: 1
8 i 1 . er: 8
8 i 2 . er: 64
8 i 3 . er: 512
8 i 4 . er: 4096
8 i 5 . er: 32768
8 i 6 . er: 262144
8 i 7 . er: 2097152
8 i 8 . er: 16777216
>>>

Det hexadecimale talsystem:
>>> for i in range(0,9):
...     print "16 i",i,". er:",16 ** i
```

```
...
16 i 0 . er: 1
16 i 1 . er: 16
16 i 2 . er: 256
16 i 3 . er: 4096
16 i 4 . er: 65536
16 i 5 . er: 1048576
16 i 6 . er: 16777216
16 i 7 . er: 268435456
16 i 8 . er: 4294967296
>>>
```

Udskriv hexadecimale tal:

```
>>> for i in range(0,17):
...     print hex(i)
...
0x0
0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8
0x9
0xa
0xb
0xc
0xd
0xe
0xf
0x10
```

Udskriv oktale tal

```
>>> for i in range(0,17):
...     print oct(i)
...
0
01
02
03
04
05
06
07
010
011
012
013
014
015
016
```

```
017
020
```

2.18.2. Fra oktale og hexadecimale tal til decimale

```
>>> a = 0x49
>>> a
73
>>> a = 0111
>>> a
73

>>> # om der benyttes "store" eller "små"
>>> # bogstaver er uden betydning
>>> a = 0xa5
>>> a
165
>>> b = 0XB2
>>> b
178
```

2.18.3. Fra decimale til oktale og hexadecimale tal

```
>>> oct(73)
'0111'
>>> hex(73)
'0x49'
```

2.18.4. Hexadecimale tal i tekststrenge

I tekststrenge erstattes nul af backslach.

```
>>> "\x41", chr(65) # 1 + 4 * 16 = 65
('A', 'A')
>>> "\x61", chr(97) # 1 + 6 * 16 = 97
('a', 'a')
```

Det lettest sagte er at funktion chr er gammeldags, vil fra Python 2.4 være helt out, så det er ene og alene unichr funktionen, der bør bruges. Men går vi i dybden, så er det ikke helt så let. Forklaringen er, at chr funktionen i dens oprindelige udformning ene og alene var beregnet til karakterer (tal, bogstaver m.m.), hvis numeriske værdi var under 129. Til og med Python 2.2 returneredes der en fejlmelding, hvis karakterens numeriske værdi var større end 128. Fra og med karakter nummer 129 og tegntabellen ud skulle funktionen unichr bruges. Kort sagt er det en større tegntabel, hvor stor ligger sådan lidt hen i det uvisse, for Pythons vedkommende bliver den fra version 2.4 enorm stor, da man går over til at bruge 64 bits (et ettal efterfulgt af 64 nuller) kode. Der må have været en del overvejelser i Python samfundet, om hvordan chr funktionen skal fungere fremover, for selv i den samme version af sproget er der og har der været afvigelser. I den version jeg bruger lige nu (en Python 2.3) returnerer chr og unichr funktionerne tegnnummer 156 således:

```
>>> chr(156)
'\x9c'
>>> unichr(156)
u'\x9c'
```

I begge tilfælde er returneringen helt i orden. At der står et u foran anførselstegnet i unichr returneringen fortæller, at strengen er en unicode streng, hvad chr returneringen naturligvis ikke er. FORDI returneringen her var i orden, kan der være god grund til at antage, at Python har "snydt" lidt, så der i den version jeg bruger lige nu, benyttes samme kode i chr som i unichr funktionerne, men som antydet er det ikke altid tilfældet, hvad du vil kunne konstere ved at bruge en Python version lavere end 2.3

```
l = []  # her oprettes tom liste
for i in range(65,91):
if i / 78 == 0: print "\n"
l.append(chr(i)) # chr konverteret tal til karakter (character)
...
>>> l
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
>>> l = []
>>> for i in range(65 + 32,91 + 32):
...     l.append(chr(i))
...
>>> l
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
>>>
```

Tallet 32 er lig med afstanden mellem et "store" bogstavs numeriske værdi og det tilsvarende "lille" bogstavs numeriske værdi.

```
>>> def  uni():
...     tal = input("Skriv et helt tal mellem -1 og 256: ")
...     print tal, unichr(tal)
```

```
...
>>> uni()
Skriv et helt tal mellem -1 og 256: 255
255 ÿ
```

2.19. Komplekse tal

I Python er der naturligvis også støtte for komplekse tal (komplekse tal er sammensat af reelle tal (normale tal) og irrationale tal som f.eks. kvadrat- og kubikrod, Pii og uendelige uperiodiske decimalbrøker). De immaginære tal skrives med et efterstillet j eller J.

Komplekse tal med et reelt tal, de ikke er nul, skrives som (reelle tal + immaginære tal med efterstillet j) eller kan dannes ved anvendelse af complex(reelt tal, immaginært tal) funktionen.

```
>>> 5 * 4j
20j
>>> 5.0 * 5j
25j
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Komplekse tal repræsenteres ved en reel og en immaginær del. De to dele kan adskilles ved brug af kompleksTal.real og kompleksTal.imag:

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Konverterings funktionerne for flydende tal og heltal (float(), int() and long()) fungerer ikke på komplekse tal. Her skal du anvende abs(immaginære del af tal) og kompleksеTal.real:

```
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

2.20. Klasser

Objekter som flyvemaskiner, rugbrød og knapper i et Python GUI program har egenskaber som eksempelvis, der kan være fælles. Eksempler på egenskaber er farve og størrelse. Jo flere egenskaber, de har fælles, jo nærmere er de beslægtede. De kan have så mange fælles egenskaber, at det bliver naturligt at samle dem i fælles grupper eller klasser. I OOP (objektorienteret programmering - se først i bogen) går ud på at udnytte (genbruge) fælles egenskaber.

Klasser repræsenterer grupper af objekter med fælles egenskaber. At egenskaberne er fælles kan i høj grad bruges til noget, når det gælder Python og andre objektorienterede computersprog. Programmøren skal ikke genopfinde de fælles egenskaber, men kan lade sine klasser arve de egenskaber andre klasser allerede er i besiddelse af. Han eller hun fortæller Python, at der skal oprettes en klasse ved at indsætte nøgleordet class først på en programlinje eks:

```
class Klasse:
 pass
```

Det viste eksempel arver ikke direkte nogen som helst egenskaber, men der skal selvfølgelig en masse kode til for at danne de nødvendige forbindelser (interface mellem eksemplet og fortolkeren). Det kan følgende antyde:

```
>>> dir(Klasse)
['__doc__', '__module__']
>>>
```

```
Her ser du endnu et eksempel, på hvor beskeden en fuldgyldig Python klasse kan være:
```

```
>>> class Klasse:
 """ Dette er en fuldgyldig Python klasse."""
```

```
>>> class Klasse:
 pass
```

Defineringen skal indledes med class, der skal efterfølges af et lovligt navn og et kolon, hvor sidstnævnte er indledningen til klassens krop. Denne indledning er samtidig indledningen af en blok. I denne blok skal der være mindst en fejlfri programlinje. De er der i begge de viste eksempler, hvorfor de er helt OK. Det vil jeg godt se et bevis på lige nu og her:

```
Det kan vi få en kontrol på lige her og nu:

>>> class Klasse:
 """ Dette er en fuldgyldig Python klasse."""

>>> Klasse
<class __main__.Klasse at 0x8291e24>
>>>


>>> class Klasse:
 pass

>>> Klasse
<class __main__.Klasse at 0x827038c>
>>>
```

Klasse opfylder alle betingelser for at være en fuldgyldig Python klasse. Den begynder med nøgleordet class, har et navn, der er fuldt lovligt og dermed brugbart, og den omfatter en krop, der begynder med kolon og den i Python obligatoriske indrykning. Nøgleordet pass gør ingen ting, og det er netop formålet med pass.

Som vist kan det konstateres, at Klasse virkelig er en fuld færdig og lovlig Python klasse ved fra interaktiv mode at skrive klassenavnet (Klasse) og trykke på Enter-tasten.

```
x = Klasse()

Test:
>>> x
<__main__.Klasse instance at 0x826ca94>
>>>
```

I Python er der en række muligheder for at få uddybende informationer om en klasse. Dem vil vi se på her i indledningen til afsnittet om klasser:

```
>>> dir
<built-in function dir>

Den fordefinerede eller i Python indbyggede (built-in) funktion dir kan med
sin parameterliste fortælle, om Klasse nu også er en klasse.

>>> dir()
['Klasse', 'PyShell', '__builtins__', '__doc__', '__name__']
>>>
```

Der oprettes en liste, hvis første element er klassenavnet selv. Ved konstruktionen af Klasse er PyShell anvendt, hvad PyShell er, kan vi også få oplyst:

```
>>> PyShell
<module 'PyShell' from '/usr/lib/python2.2/site-packages/idle/PyShell.py'>
>>>
```

Det kan vel næppe komme som en overraskelse, at der er anvendt et modul til konstruktionen af vores Klasse, men prøv lige selv at lave følgende test, og du vil overraskes over, hvor mange stumper, der er anvendt til konstruktionen:

```
>>> import PyShell
>>> dir(PyShell)
['ACTIVE', 'ALL', 'ANCHOR', 'ARC', 'At', 'AtEnd', 'AtInsert', ...

Nederst i den lange udskrift, hvorfra jeg kun har medtaget starten, genfinder du

'__builtins__', '__doc__' og '__name__',
```

Helt tilsvarende kan man få oplysning, om hvilke programstumper (OOP dele), der indgår i Klasse:

```
>>> dir(Klasse)
['__doc__', '__module__']
>>>
```

Dem vil vi se nærmere på i forbindelse med Klasse, men Klasse er defineret tom, så vi behøver en udvidelse af erklæringen:

```
>>> class Klasse:
 def _init_(self):
  pass
```

"def _init_(Klasse)" er på sin vis 3 ting: 1: definering af funktionen _init_ og 2: initialisering (grundlæggende værditildeling til Klasse) og 3: Klasses konstruktør. Argumentet self fortæller, at Klassen bruger sine egne egenskaber, hvad det vil sige, skal vi snart se, men lad os først lave endnu en kontrol:

>>> dir(Klasse) ['__doc__', '__module__', '_init_']

Vi ser, at _init_ er indsat som element i listen.

>>> Klasse._init_ <unbound method Klasse._init_>

Vi har endnu ikke importeret et modul. Det vil vi gøre:

>>> import math >>> class Klasse(math): print math.pi

3.14159265359 >>>

Test: >>> import math >>> math <module 'math' from '/usr/lib/python2.2/lib-dynload/math.so'> >>>

Når Python-fortolkeren indleder kørslen af et program, er værdien af _name_ lig med _main_ (læg godt mærke til det, for programlinjen indgår i en lang række af bogens eksempler, hvor det er svært at

kommentere og samtidig bevare læsbarheden, og den derfor udelades programlinjen er if _name_ == "_main_": eks. if __name__ == '__main__': Dialog().mainloop() det skal læses som: Hvis det er klassen Dialog, der startes op, så skal løkken mainloop startes op - den kontrollerer, om der indtræder nye hændelser (events) fra program og/eller bruger.)

Namespace og scope er to navne for det samme - at angive et navns virkningsområde eller sagt på en anden måde det område, hvori navnet kan bruges. Det kan følgende eksempel belyse:

```
>>> x = 256
>>> class Klasse:
 print x

256
>>>
```

Klasse har ene og alene mulighed for at kende navnet x og dermed for at kunne returnere variablens værdi, fordi v er defineret som global variabel (globalt navn). Det fører videre til:

```
x = 1 # global variabel
```

ændrer den lokale variabel x (shadows (skygger for) den globale variabel def a(): x = 25

print "\nVærdien i den lokale x er", x, "efter defineringen og kaldet af a" x += 1 print "Den lokale x (den i a) er",x, #før a forlades"

ænder den globale variabel x def b(): global x print "\nDen globale x er nu",x,"Ved kald fra b" x *= 10 print "Globale x er",x,"når b er forladt"

print "Globale x er",x x = 7 print "Globale x er", x

a() b() a() b()

print "\nGlobale x er" ,x

Kørselsresultat:

Globale x er 1

Globale x er 7

Værdien i den lokale x er 25 efter defineringen og kaldet af a

Den lokale x (den i a) er 26 Den globale x er nu 7 Ved kald fra b

Globale x er 70 når b er forladt

Værdien i den lokale x er 25 efter defineringen og kaldet af a

Den lokale x (den i a) er 26 Den globale x er nu 70 Ved kald fra b

Globale x er 700 når b er forladt

Globale x er 700

```
>>> class Klasse:
...     i = 123
...     def f(self):
...             return "Python klasser er i en klasse for sig."
...
```

Klasse.i og Klasse.f er begge lovlige attribut referencer, der henholdsvis returnerer et heltal og brugt på rette måde resultatet af en metodeafvikling. Læg mærke til, at for at få nævnte resultat returneret, så skal både klasse og metode have parameterliste.

```
>>> Klasse.i
123
>>> print Klasse().f()
Python klasser er i en klasse for sig.
```

Har du gemt nogle af dine definitioner i py filer, har du for så vidt allerede lavet Python moduler, der principielt fuldstændig svarer til de fordefinerede moduler i Python. Men skriv de to følgende funktioner i en editor og gem dem med navnet fibonacci.py

```
def fib(n):
 a, b = 0, 1
```

```
 while b < n:
  print b,
  a, b = b, a + b
def fib2(n):
 resultat = []
 a, b = 0, 1
 while b < n:
  resultat.append(b)
  a,b = b, a + b
 return resultat
```

fibonacci.py er et ganske normalt python modul og skal derfor også importeres på den helt normale måde. Start Python op i interaktiv mode og skriv følgende kode:

>>> import fibonacci >>> fibonacci.fib(100) >>> fibonacci.fib2(100)

Gør du det vil programafviklingen se således ud: >>> import fibonacci >>> fibonacci.fib(100) 1 1 2 3 5 8 13 21 34 55 89 >>> fibonacci.fib2(100) [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89] >>>

Hvis du har til hensigt at bruge en funktion ofte, kan du tildele den et lokalt navn: >>> import fibonacci >>> f = fibonacci.fib

Et kørselsresultat: >>> f(500) 1 1 2 3 5 8 13 21 34 55 89 144 233 377 >>>

```
Naturligvis kan du også bruge  type og dir  funktionerne på fibonacci.py:
>>> type(fibonacci)
<type 'module'>
fibonacci.py er altså et modul, hvad vi  vel nok kunne ane.
>>> dir(fibonacci)
['__builtins__', '__doc__', '__file__', '__name__', 'fib', 'fib2']
Her returneres liste indeholdende alle navne, variabler, moduler,
funktioner m.v., der er defineret i øjeblikket.
>>> dir()
['__builtins__', '__doc__', '__name__', 'fibonacci']
>>>
```

Uden argumenter returnerer \dir() de navne, du har defineret i øjeblikket. Det drejer sig alene om fibonacci modulet. dir() lister ikke den indbyggede (fordefinerede) funktioners og variablers navne, hvis du ønsker at se dem, så kan det ske ved at udskrive en liste af standardmodulet __bultin__ 's indhold:

```
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'DeprecationWarning', ....
>>> import sys
>>> sys.path = ".:/usr/local/lib/python"
```

Kørselsresultat: >>> sys.path ['', '/usr/local/lib/python23.zip', '/usr/local/lib/python2.3', '/usr/local/lib/ python2.3/plat-linux2','/usr/local/lib/python2.3/lib-tk', '/usr/local/lib/ python2.3/lib-dynload', '/usr/local/lib/python2.3/site-packages'] >>>

>>> PYTHONPATH = sys.path # PYTHONPATH skal defineres for at kunne bruges

```
Et kørselsresultat:
>>> PYTHONPATH
['', '/usr/local/lib/python23.zip', '/usr/local/lib/python2.3', '/usr/local/lib/
python2.3/plat-linux2', '/usr/local/lib/python2.3/lib-tk', '/usr/local/lib/
python2.3/lib-dynload', '/usr/local/lib/python2.3/site-packages']
>>>
```

Som du ser, returnerer PYTHONPATH en liste. I den er de aktuelle stier, de stier hvori fortolkeren vil søge for at finde moduler m.v. Det kan du udnytte ved at tilføje det eller de biblioteker, hvori du gemmer dine selvkomponerede klasser, som element eller elementer. I det store og hele er det underordnet, hvor du placerer dine moduler. Jeg oprettede biblioteket mineModuler i mit brugerområdes rod. Nu kan jeg flytte mine moduler dertil og indsætte biblioteket som element i listen:

```
PYTHONPATH.append("/home/ajbo/mineModuler")
```

Kørselsresultat: >>> PYTHONPATH ['', '/usr/local/lib/python23.zip', '/usr/local/lib/python2.3', '/usr/local/lib/python2.3/plat-linux2', '/usr/local/lib/python2.3/lib-tk', '/usr/local/lib/python2.3/lib-dynload', '/usr/local/lib/python2.3/site-packages', '/home/ajbo/mineModuler'] >>>

```
Det er også muligt at adskille sys.path og PYTHONPATH,
så de hver især kan bruges efter behov:
>>> PYTHONPATH = "/home/ajbo/mineModuler"
>>> PYTHONPATH
'/home/ajbo/mineModuler'
>>> sys.path
['', '/usr/local/lib/python23.zip', '/usr/local/lib/python2.3',
```

```
'/usr/local/lib/python2.3/plat-linux2',
'/usr/local/lib/python2.3/lib-tk', '/usr/local/lib/python2.3/lib-dynload',
'/usr/local/lib/python2.3/site-packages']
>>>
```

Del sætningen kan bruges på 2 forskellige måder. Der er en måde, hvorpå et element kan fjernes fra en liste ud fra elementets indeks-nummer i stedet for selve elementet. Den kan bruges til at fjerne slices fra en liste. Og som eks. 2 viser kan del også fjerne selve listen.

```
Eksempel 1:
>>> liste = [-1, 1, 66.6, 333, 333, 1234.5]
>>> del liste[0]
>>> liste
[1, 66.6, 333, 333, 1234.5]
```

Eksempel 2: >>> del liste[2:4] >>> liste [1, 66.6, 1234.5]

```
Eksempel 3:
>>> del liste
```

2.21. Sortering

2.21.1. Sortering af basale typer

I mange computersprog bruges nogenlunde følgende rutine, når værdierne i to variabler skal bytte plads f.eks. i en sorteringsrutine:

```
>>> a = 50; b = 25
>>> temp = a
>>> a = b
>>> b = temp
>>> print a,b
25 50
>>>
```

```
I Python er samme proces kortere og mere logisk:
>>> a = 50; b = 25
>>> a, b = b, a
```

```
>>> print a,b
25 50
>>>
```

```
Sortering af elementerne i en liste foretages let med sort() funktionen:
```

```
>>> l = [2,34,5,4,78,6,12]
>>> l.sort()
>>> print l
[2, 4, 5, 6, 12, 34, 78]
>>>
```

Ud fra det, jeg har været inde på et eller flere andre steder i bogen, kan du passende forsøge at forklare, hvorfor også den følgende sortering er korrekt, for det er den.

```
>>> l = ["A","d","B","g","s","a","w","I","k","a","p","P"]
>>> l.sort()
>>> print l
['A', 'B', 'I', 'P', 'a', 'a', 'd', 'g', 'k', 'p', 's', 'w']
>>>
```

Sort funktionen kan anvendes uden parameter eller med en funktion som parameter. Anvendelsen uden parameter, som vist ovenfor, svarer helt til sort(cmp), hvor cmp() er en fordefineret funktion der sammenligner to værdier, x og y, og returnerer -1,0 og 1 afhængig af om\ x < y, x == y eller \ x > y.

```
>>> l = [2,34,5,4,78,6,12]
>>> l.sort()
>>> print l
[2, 4, 5, 6, 12, 34, 78]

Eller:
>>> l.sort(cmp)
>>> print l
[2, 4, 5, 6, 12, 34, 78]
>>>

Hvis du ønsker det, kan du let lave din egen sammenligningsfunktion:

>>> def Sam(a,b):
...     return a - b
...
>>> l = [2,34,5,4,78,6,1]
>>> l.sort(Sam)
>>> print l
```

```
[1, 2, 4, 5, 6, 34, 78]
>>>
```

Lambda kan være svær at forstå virkningen af. Som jeg er inde på andetsteds, så opererer lambda i baggrunden. Lad os se på Sam() ovenfor igen. Sam er en funktion med sit eget navn. Sam() trækker b fra a. Ved brug af lambda kan denne operation flyttes op i sort's parameterliste:

```
>>> l = [2,34,5,4,78,6,1]
>>> l.sort(lambda a, b : a- b)
>>> print l
[1, 2, 4, 5, 6, 34, 78]
>>>
```

Nu kunne jeg imidlertid godt tænke mig at anvende samme lambda på en sortering i faldende orden. Det eneste, jeg da skal gøre er at bytte om, så

```
>>> l = [2,34,5,4,78,6,1]
>>> l.sort(lambda a, b : b - a)
>>> print l
[78, 34, 6, 5, 4, 2, 1]
>>>
```

```
Tilsvarende kunne Sam() ganske let ændres til at fortage sortering med
faldende orden:
>>> def Sam(a,b):
...     return b - a
...
>>> l = [2,34,5,4,78,6,1]
>>> l.sort(Sam)
>>> print l
[78, 34, 6, 5, 4, 2, 1]
>>>
```

I sorteringrutiner er hastighed ofte et vigtigt parameter, så i sorteringer i stigende orden er det bedst at bruge sort uden parametre. Ønsker du sortering i faldende orden bliver ekspeditionstiden kortest ved at udføre operationen i to omgange. Først foretages en sortering i stigende orden med sort() og derefter anvendes funktionen reverse():

```
>>> l = [2,34,5,4,78,6,1]
>>> l.sort()
>>> l.reverse()
>>> print l
[78, 34, 6, 5, 4, 2, 1]
>>>
```

Med strengfunktionen split(), er det enkelt at konvertere en tekststreng til en liste. Når det er gjort, kan strengen sorteres helt tilsvarende den numeriske sortering, vi har set på ovenfor:

```
>>> import string
>>> streng = "Lad os dele opgaven op i enkeltelementer."
>>> s = streng.split()
>>> s
['Lad', 'os', 'dele', 'opgaven', 'op', 'i', 'enkeltelementer.']
>>> s.sort()
>>> s
['Lad', 'dele', 'enkeltelementer.', 'i', 'op', 'opgaven', 'os']
>>> s.reverse()
>>> s
['os', 'opgaven', 'op', 'i', 'enkeltelementer.', 'dele', 'Lad']
>>>

Her de samme rutiner med brug af Lambda funktionen:
>>> import string
>>> l = "Lad os dele opgaven op i enkeltelementer.".split()
>>> l.sort(lambda a,b:cmp(a,b))
>>> print l
['Lad', 'dele', 'enkeltelementer.', 'i', 'op', 'opgaven', 'os']
>>>

>>> import string
>>> l = "Lad os dele opgaven op i enkeltelementer.".split()
>>> l.sort(lambda a,b: - cmp(a,b))
>>>
>>> print l
['os', 'opgaven', 'op', 'i', 'enkeltelementer.', 'dele', 'Lad']
>>>

Her er en kasusafhængig strengsortering med stigende orden,
der anvender Lambda funktionen:
>>> import string
>>> l = "Lad os dele opgaven op i enkeltelementer.".split()
>>> l.sort(lambda a,b: cmp(string.lower(a), string.lower(b)))
>>> print l
['dele', 'enkeltelementer.', 'i', 'Lad', 'op', 'opgaven', 'os']

Og her den samme sortering i faldende orden:
>>> l.sort(lambda a,b: cmp(string.lower(b), string.lower(a)))
>>> print l
['os', 'opgaven', 'op', 'Lad', 'i', 'enkeltelementer.', 'dele']
>>>

Sorter strengen, indsæt elementerne i en tuple og sæt den
ind i en liste for senerer sortering:

>>> import string
>>> l = "Lad os dele opgaven op i enkeltelementer.".split()
>>> listen = []
>>> for i in range(len(l)):
```

```
...     listen.append((string.lower(l[i]),i))
...
>>> listen.sort()
>>> print listen
[('dele', 2), ('enkeltelementer.', 6), ('i', 5), ('lad', 0), ('op', 4),
('opgaven', 3), ('os', 1)]
>>>

Nu kan listen renses for tuplens indvirkning ved
(vi har ingen brug for indeksværdierne (tallene)) :

>>> nyListe = []
>>> for springOver, i in listen:
...     nyListe.append(l[i])
...
>>> print nyListe
['dele', 'enkeltelementer.', 'i', 'Lad', 'op', 'opgaven', 'os']
>>>

En anden vej til målet er at lagre de oprindelige data (elementer) som
listens andet led:

>>> import string
>>> l = string.split("Lad os dele opgaven op i enkeltelementer.")
>>> listen = []
>>> for element in l:
...     listen.append((string.lower(element), element))
...
>>>
>>> print listen
[('lad', 'Lad'), ('os', 'os'), ('dele', 'dele'), ('opgaven', 'opgaven'), ('op', 'op'),
 ('i', 'i'), ('enkeltelementer.', 'enkeltelementer.')]
```

2.22. Exceptions (undtagelser)

```
eks. 1
>>> x
Traceback (most recent call last):
  File "<pyshell\# 0>", line 1, in ?
    x
NameError: name 'x' is not defined
```

Ved at fortolke koden bagfra finder Python en fejl i linje 1. Det viser sig at være en navnefejl - variablen x er ikke defineret.

Der findes ofte rester af forskellig slags i arbejdslageret, er den krævede definering en meget stor fordel, for den betyder, at en nyerklæret variabel samtidig tildeles en værdi.En variabel i Python altid pege på den værdi, som du eller en anden programmør har sat den til ved seneste anvendelse. Den kan ganske enkelt ikke pege på en af de nævnte datarester.

eks. 2 Vi kunne undgå fejlmeldingen ovenfor ved at tildele variablen en værdi. Så er alt for så vidt i orden, i hvert fald i det aktuelle tilfælde:

```
>>> # Her er variablen defineret
>>> x = 1
>>> try: print x
except Navnefejl: x = 0
1
```

Fordi variablen er erklæret og tildelt værdi, så kan try løkken gennemføres tilfredsstillende, så afviklingen når aldrig frem til except løkken.

```
>>> # Tilsvarende uden defineret variabel
>>> try: print y
except Navnefejl: y = 0
Traceback (most recent call last):
  File "<pyshell\#51>", line 3, in -toplevel-
    except Navnefejl: y = 0
NameError: name 'Navnefejl' is not defined
```

Fordi variablen ikke er erklæret og tildelt værdi, så kan try løkken ikke gennemføres tilfredsstillende, så afviklingen når frem til except løkken. Her gives fejlmelding, da Navnefejl er ukendt, det samme gælder y, men afviklingen afbrydes inden y behandles.

2.23. Unicode

Python 2.0 kunne anvende grundtypen Unicode strenge. Unicode bruger 16-bit tal til at repræsentere karakterer i stedet for de 8-bits tal, der bruges af ASCII. Det betyder, at der i stedet for 256 tegn i en tegntabel nu er plads til 65.536 forskellige tegn.

I Python kildekode, det du skriver, når du programmerer i Python, skrives Unicode strenge som u"tekststreng". Unicode karakterer ved hjælp af escape sekvensen \\uHHHH, hvor HHHH er et 4-cifret hexadecimalt tal fra 0000 til FFFF. Den eksisterende \\xHHHH escape sekvens kan også anvendes. Det

samme gælder oktale escape sekvenser for karakterer op til U+01FF, der repræsenteres af \\777. Jeg har ikke fundet det nødvendigt at tage eksempler med her, men skal gøre opmærksom på at Python 2.4 også skulle kunne håndtere 32 og 64 bits unicoder.

En konvertering til Unicode streng returnerer en 8-bit streng i det ønskede kodeformat, der eksempelvis kan være 'ascii', 'utf-8' eller den vi vel nok oftest benytter i Danmark 'iso-8859-1'.

Sammenføjning af 8-bit og Unicode strenge vil altid returnere en Unicode streng.

eks: >>> 'a' + u'bc' u'abc'

2 metoder til målet:

>>> "Nexø" 'Nex\xf8' >>> print "Nexø" Nexø

2 andre metoder til 2 ANDRE mål: >>> print "Rønne" Rønne

>>> print str("Rønne") Rønne

Sådan skulle ord funktionen anvendes i Python med lavere versionsnummer end 2.3:

>>> ord(u'\xc6') 198

Fordi: >>> ord("Æ")

UnicodeError: ASCII encoding error: ordinal not in range(128)

Og sådan nu (med Python version 2.3): >>> ord("Æ") 198

Talværdien 198 er bogstavets korrekte nummer i Unicode tabellen.

>>> u"Computersproget\u0020Python\u0020er\u0020fra\u00201991" u'Computersproget Python er fra 1991' >>>

Konverter unicode streng til gammeldags 8-bit streng >>> u"æøåÆØÅ".encode('utf-8') '\xc3\xa6\xc3\xb8\xc3\xa5\xc3\x86\xc3\x98\xc3\x85' >>>

2.24. Prøv ... Ellers

BEMÆRK Pythons indrykninger SKAL bevares, ellers KAN eksemplet ikke afvikles:

```
>>> while 1:
 try:
  indkomst = int(raw_input("Skriv et helt tal: "))
  break
 except ValueError:
  print "Tallet var ikke et heltal (an integer). Prøv igen..."
```

Skriv et helt tal: 23.6 Tallet var ikke et heltal (an integer). Prøv igen... Skriv et helt tal: 24 >>>

Af hensyn til overskueligheden og til den absolut nødvendige bevarelse af indrykningerne, kommenterer jeg først eksemplet her: while 1: læses som lige så længe påstanden er sand, så skal blokken indkomst = int(raw_input("Skriv et helt tal: ")) og break udføres.

```
raw_input("Skriv et helt tal: ")
```

modtager en værdi, der lagres et eller andet sted i computerens arbejdslager, men inden det sker, konverteres den til et heltal (an integer) v.h.a. funktionen int().Kan det ske, sættes en variabel med navnet indkomst til at pege på stedet i lageret, hvor det hele tal findes i form af et antal "ledninger" med og uden strøm på (svarende helt til 1-taller og 0-er) i vores binære talsystem. Såfremt og kun såfremt den indtastede værdi kan tildeles indkomst, så fortsættes til næste programlinje break - afbryd while løkken. Er den indtastede værdi et heltal, kommer blokken except ValueError: aldrig i brug, så det næste Python skal gøre er at melde klar til mere aktiv kommunikation med brugeren, hvilket som bekendt meldes med \>>>

Men en gang mere PAS PÅ INDRYKNINGERNE. De SKAL ubetinget være korrekte, ellers kan Python-fortolkeren jo ikke afgøre, hvor en blok begynder og slutter.

2.25. Ordbog (dictionary)

Lad os starte gennemgangen af ordbøger (dictionaries) et helt andet sted, ved et tilbageblik til lister og tuples.

```
Liste:
>>> ordliste = ['Arnager', 1, 'Hasle', 2, 'Vang', 3]
>>> ordliste
```

```
('Arnager', 1, 'Hasle', 2, 'Vang', 3)
>>> ordliste[0]
'Arnager'
```

tuple: >>> ordliste = 'Arnager', 1, 'Hasle', 2, 'Vang', 3 >>> ordliste ('Arnager', 1, 'Hasle', 2, 'Vang', 3) >>> ordliste[0] 'Arnager'

```
Ordbogen:
>>> ordliste = {'Arnager': 1, 'Hasle': 2, 'Vang' : 3}
>>> ordliste
{'Arnager': 1, 'Hasle': 2, 'Vang': 3}
ordliste[0]
KeyError: 0
```

Sammenlign nu de ovenstående bevidst ukommenterede eksempler, så er vi ligesom klar til at gå videre til det egentlige. En ordbog/ordliste adskiller sig fra listen på den måde at listen er omgivet af firkaltede parenteser, mens ordbogen er omgivet af krøllede parenteser. tuples kan, men ikke nødvendigvis, være omgivet af almindelige runde parenteser. Mens liste og tuple kan returnere element v.h.a. navn og firkantede parenteser indeholdende elementnummer, kræver returnering fra en ordbog en lidt mere kompliceret operation. Den vil vi tage lidt i etaper:

```
>>> if "Hasle" in ordliste:
  print ordliste['Hasle']
else:
  print 'ikke fundet'
2
```

Hvis vi nu i stedet for at tænke på den direkte oversættelse af ordet dictionary (ordbog) og i stedet for tænker på den som en ordliste - en indholdsfortegnelse i en bog, så kan det returberede 2-tal ovenfor fint henvise til den side i bogen, hvor Hasle er omtalt og sådan fremdeles.

Hvis vi i stedet for at tænke på the dictionary som indholdslisten tænker på den som en lænke (a link) på en hjemmeside, så vil det ofte være således, at en hotlink i form af en virtuel knap med en eller anden påskrift ved et klik med musen fører til ankerpladsen - en beskrivelse eller omtale af påskriften, så kunne en ordliste komme til at se ud som denne:

```
ordliste = {"Arnager": "Lille fiskerby syd for Bornholms flyveplads." ,
"Hasle": "Lille vestbornholmsk by.", 'Vang' : 'Lille by, havde tidligere
```

```
en del stenindustri.' }
>>> if "Vang" in ordliste:
  print ordliste['Vang']
else:
  print 'ikke fundet'
Lille by, der tidligere havde en del stenindustri.
```

Nu er vi vist allerede kommet dertil, hvor Pythons ordbøger kan bruges til noget konstruktivt. Men der er meget mere endnu, vi skal se på i den forbindelse.

```
>>> indeks = {}
>>> def indskriv(indgangsord, sidenummer):
        if indeks.has_key(indgangsord):
            indeks[indgangsord].append(sidenummer)
        else:
            indeks[indgangsord] = [sidenummer]
>>> indskriv("Gudhjem",0)
>>> indeks
{'Gudhjem': [0]}
indeks = {}
def nyIndgang(noegle, sidenummer):
    if indeks.has_key(noegle):
        indeks[noegle].append(sidenummer)
    else:
        indeks[noegle] = [sidenummer]
>>> nyIndgang("Arnager",0)
>>> nyIndgang("Bodilsker",1)
>>> nyIndgang("Pedersker",2)
>>>
>>> indeks
{'Arnager': [0], 'Bodilsker': [1], 'Pedersker': [2]}
# Pas på med sidenumrene ellers:
>>> nyIndgang("Arnager",0)
>>> nyIndgang("Bolsker",0)
>>> indeks
{'Arnager': [0], 'Bolsker': [0]}
```

Mange har anskaffet et eller flere af de efterhånden prisbillige digitale kameraer. De lagrer oftest billederne i jpg eller tif format. Førstnævnte er det mest velegnede til hjemmesider m.v. Samme filtype kan desværre ikke uden videre anvendes af Python - der skal et ekstra modul til. Derfor kan det være en fordel at konvertere billeder til gif formater, der har en række fordele frem for jpg formatet, med generelt fylder en smule mere. Tif formatet fylder temmelig mere end de to andre formater. Det skyldes primært at tif kan gemme flere farver - farver der ofte er usynlige for det menneskelige øje. Her vil jeg ene og alene vise, hvordan gif og jpg formaterne hentes inde i Python programmer.

```
from Tkinter import *
root = Tk()
foto = PhotoImage(file = "AKTIV/h.gif")
Button(root, image = foto).pack()
root.mainloop()
```

eller f.eks.

```
from Tkinter import *
root = Tk()
fotobibliotek = "AKTIV"
foto = PhotoImage(file = "fotobibliotek + "h.gif")
Button(root, image = foto).pack()
root.mainloop()
```

Det følgende eksempel kræver PIL installeret - PIL håndterer over 30 formater PIL kan hentes fra www.pythonware.com til såvel Windows som Linux. Pas på at få den rette (nyeste) version af det gratis eller evt. det kommercielle modul.

```
from Tkinter import *
import ImageTk, Image
root = Tk()
fotobibliotek = "AKTIV"
foto = ImageTk.PhotoImage(file = fotobibliotek + "h.jpg")
Button(image = foto).pack()
root.mainloop()
```

Figur 2-1. Skyer

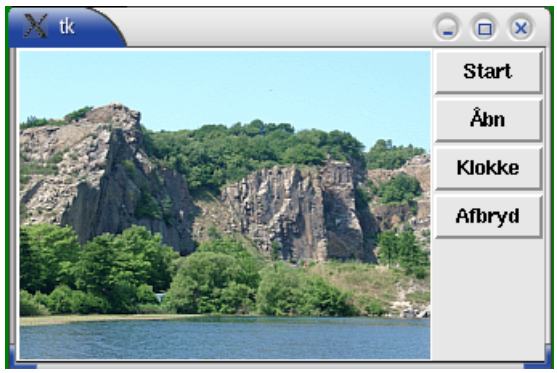

```
from Tkinter import *
from glob import glob
from tkMessageBox import askyesno
from tkFileDialog import askopenfilename
import random
def opretArvinger(root):
    global arbejdsplads, tilslutAfbryd
    arbejdsplads = Canvas(root, bg='white')
    arbejdsplads.pack(side=LEFT, expand=YES, fill=BOTH)
    tilslutAfbryd = Button(root, text='Start', command=onStart)
    tilslutAfbryd.pack(fill=BOTH)
    Button(root, text='Åbn',  command=onOpen).pack(fill=BOTH)
    Button(root, text='Klokke',  command=onBeep).pack(fill=BOTH)
    Button(root, text='Afbryd',  command=onQuit).pack(fill=BOTH)
def onStart():
    global rundgang
    rundgang = 1
    tilslutAfbryd.config(text='Stop', command=onStop)
    onTimer()
def onStop():
    global rundgang
    rundgang = 0
    tilslutAfbryd.config(text='Start', command=onStart)
def onOpen():
    global foto
    onStop()
    pick = askopenfilename(initialdir=bibliotek)
```

```
    if pick:
        foto = PhotoImage(file=pick)
        arbejdsplads.config(height=foto.height(), width=foto.width())
        arbejdsplads.create_image(2, 2, image=foto, anchor=NW)
def onQuit():
    if askyesno('Verify', 'Vil du slutte?'):
        root.quit()
def onBeep():
    global hyl
    hyl = hyl ^ 1
def onTimer():
    global foto
    if rundgang:
        pick = random.choice(fotos)
        foto = PhotoImage(file=pick)
        arbejdsplads.create_image(2, 2, image=foto, anchor=NW)
        if hyl: root.bell()
        root.after(milSekunder, onTimer)
import sys
if len(sys.argv) == 2:
    bibliotek = sys.argv[1]
else:
    bibliotek = '../gifs' # Forvalgt bibliotek
fotos = glob(bibliotek + '/*.gif')
root = Tk()
opretArvinger(root)
milSekunder = 2000
hyl  = 1
root.mainloop()
```

```
from Tkinter import *
from glob import glob
from tkMessageBox import askyesno
from tkFileDialog import askopenfilename
import random
def opretArvinger(root):
    global arbejdsplads, tilslutAfbryd
    arbejdsplads = Canvas(root, bg='white')
    arbejdsplads.pack(side=LEFT, expand=YES, fill=BOTH)
    tilslutAfbryd = Button(root, text='Start', command=onStart)
    tilslutAfbryd.pack(fill=BOTH)
    Button(root, text='Åbn',  command=onOpen).pack(fill=BOTH)
    Button(root, text='Klokke',  command=onBeep).pack(fill=BOTH)
    Button(root, text='Afbryd',  command=onQuit).pack(fill=BOTH)
def onStart():
    global rundgang
    rundgang = 1
    tilslutAfbryd.config(text='Stop', command=onStop)
    onTimer()
```

```
def onStop():
    global rundgang
    rundgang = 0
    tilslutAfbryd.config(text='Start', command=onStart)
def onOpen():
    global foto
    onStop()
    pick = askopenfilename(initialdir=bibliotek)
    if pick:
        foto = PhotoImage(file=pick)
        arbejdsplads.config(height=foto.height(), width=foto.width())
        arbejdsplads.create_image(2, 2, image=foto, anchor=NW)
def onQuit():
    if askyesno('Verify', 'Vil du slutte?'):
        root.quit()
def onBeep():
    global hyl
    hyl = hyl ^ 1
def onTimer():
    global foto
    if rundgang:
        pick = random.choice(fotos)
        foto = PhotoImage(file=pick)
        arbejdsplads.create_image(2, 2, image=foto, anchor=NW)
        if hyl: root.bell()
        root.after(milSekunder, onTimer)
import sys
if len(sys.argv) == 2:
    bibliotek = sys.argv[1]
else:
    bibliotek = '../gifs' # Forvalgt bibliotek
fotos = glob(bibliotek + '/*.gif')
root = Tk()
opretArvinger(root)
milSekunder = 2000
hyl  = 1
root.mainloop()
```

Figur 2-2. Fotoarkiv



Kapitel 3. Biblioteks reference

3.1. Repr funktionen

```
Eksempler på anvendelse af repr funktionen:
>>> print "Rønne"
Rønne
>>> print str("Rønne")
Rønne
>>> print repr("Rønne")
'R\xf8nne'

>>> str(0.1)
'0.1'
>>> repr(0.1)
'0.10000000000000001'

>>> x = 10 * 3.25
y = 200 * 200
s = 'Værdien af x er ' + repr(x) + ', af y er den ' + repr(y) + '...'
print s>>> >>> >>>
Værdien af x er 32.5, af y er den 40000...

>>> x = 20; y = x ** 3
>>> repr((x, y, ('tomater', 'ostemad')))
"(20, 8000, ('tomater', 'ostemad'))"
>>>
```

3.2. Tekststrenge

En tekststreng er alt mellem dobbelte eller enkelte anførselstegn

```
Tekststeng mellem dobbelte anførselstegn:
>>> print "Velkommen til Python!"
Velkommen til Python!
```

Tekststeng mellem enkelte anførselstegn: >>> print 'Velkommen til Python!' Velkommen til Python!

print "Tekststreng med \"dobbelte anførselstegn.\"" print 'En streng med "dobbelte anførselstegn."' print 'Tekststreng med \'enkelte anførselstegn.\'' print "En streng med 'enkelte anførselstegn.'" print

"""Strengen her har "dobbelte anførselstegn"og 'enkelte anførselstegn'. Du kan tilmed udskrive over flere linjer."""" print "'Denne streng har også "dobbelte"og 'enkelte' anførselstegn."'

Anførselstegnene skal være samme slags: >>> 'Velkommen til Python! " SyntaxError: EOL while scanning single-quoted string

>>> "Velkommen til Python! ' SyntaxError: EOL while scanning single-quoted string

print "Velkommen\ntil\n\nPython!"

Strenge kan lægges sammen: >>> print "Velkommen " + 'til Python' Velkommen til Python

Strenge kan gentages: >>> print "Velkommen til Python! " * 3 Velkommen til Python! Velkommen til Python! Velkommen til Python!

>>> 'Velkommen til Python! ' * 3 'Velkommen til Python! Velkommen til Python! Velkommen til Python! '

Strenge kan sammenlignes: >>> "æøå" < "ÆØÅ" False >>> "ÆØÅ" < "æøå" True >>> >>> 'Python' < 'Pascal'<'ABC' < 'C' False >>> >>> 'ABC' > 'C' < 'Pascal' < 'Python' True >>> 'ABC' < 'C' < 'Pascal' < 'Python' False >>> 'ABC' == 'C' == 'Pascal' == 'Python' False >>> >>> 4 == "fire" False >>>

Strenge kan indskrives med raw_input: integer1 = raw_input("Skriv et helt tal: ") # read en streng integer1 = int(integer1) # Konverter strengen til heltal

heltal2 = raw_input("Enter endnu et heltal: ") # indskriv streng integer2 = int(heltal2) # et heltal og an integer er det samme

Kopier indholdet i integer1 og heltal2 til sum sum = integer1 + heltal2

print "Summen af de to heltal er ", sum

Linjeskifte indsættes med \n >>> print "Velkommen\ntil\n\nPython!" Velkommen til

Python!

Tabulatorskifte indsættes med \t >>> print "Velkommen\ttil\t\tPython!" Velkommen til Python! Bemærk her at tekstfunktionerne oftes knyttes til tekststrengen som f.eks. streng.funktionsnavn()

Skriv første streng og konverter til heltal tal1 = raw_input("Indskriv første integer: ") tal1 = int(tal1)

Skriv anden streng og konverter til heltal tal2 = raw_input("Indskriv anden integer: ") tal2 = int(tal2)

if tal1 == tal2: print "%d er lig med %d" % (tal1, tal2)

if tal1!= 2: print "%d er forskellig fra %d" % (tal1, tal2)

if tal1< 2: print "%d er mindre end %d" % (tal1, tal2)

if tal1> 2: print "%d er større end %d" % (tal1, tal2)

if tal1<= 2: print "%d er mindre end eller lig med %d" % (tal1, tal2)

if tal1>= 2: print "%d er større end eller lig med %d" % (tal1, tal2)

>>> heltal = 414 >>> heltal 414 >>> print heltal 414 >>> print "Decimaltal %d" % heltal Decimaltal 414 >>> print "Hexadecimal tal %x" % heltal Hexadecimal tal 19e

>>> for i in range(1,10): ... print "Oktale tal: %o"%i ... Oktale tal: 1 Oktale tal: 2 Oktale tal: 3 Oktale tal: 4 Oktale tal: 5 Oktale tal: 6 Oktale tal: 7 Oktale tal: 10 Oktale tal: 11

>>> kommatal = 12.4 >>> print "Kommatal (float) %f" % kommatal Kommatal (float) 12.400000 >>> print "Forvalgt eksponent %e" % kommatal Forvalgt eksponentnotation 1.240000e+01

>>> # Udskriver med fordefineret tabulatorbredde 8 tegn >>> heltal = 414 >>> print "Højre justeret heltal (%8d)" % heltal Højre justeret heltal (414) >>> print "Venstre justeret heltal (%-8d)" % heltal Venstre justeret heltal (414)

streng = "Streng formatering" >>> heltal = 25 >>> print "Gennemtving 8 cifre i heltal %.8d" % heltal Gennemtving 8 cifre i heltal 00000025

>>> kommatal = 12.1 >>> print "Gennemtving 6 cifre efter komme i float %.6f " % kommatal Gennemtving 6 cifre efter komme i float 12.100000

>>> streng = "I alle de riger og lande..." >>> print "(%.15s) (%.5s)" % (streng, streng) (I alle de riger) (I all)

En streng er en liste. Her udskrives streng som liste: >>> s = "Dette er en streng." >>> l = len(s) # len finder strenglængde >>> for i in range(0,l): ... print s[i], ... D e t t e e r e n s t r e n g .

eks. 1 capitalize() funktionen konverterer strengens første bogstav til "stort" bogstav.

>>> import string >>> "dette er en mulig anvendelse".capitalize() 'Dette er en mulig anvendelse'

eks. 2 Eksempel 1 på en anden måde

```
>>> import string
>>> streng = "rødgrød med fløde er godt."
>>> streng.capitalize()
'R\xf8dgr\xf8d med fl\xf8de er godt.'
ELLER:
>>> print streng.capitalize()
Rødgrød med fløde er godt.
```

De fleste af de mange andre tekstfunktioner anvendes svarende til eksempel 1 og 2 ovenfor. BEMÆRK hvordan de danske specialtegn erstattes med hexadecimale værdier i den første udskrift. Det skyldes, at Python benytter 2 forskellige funktioner i immediate mode ved returneringen str() og repr(), hvad følgende kan vise:

```
>>> print str("rødgrød med fløde er godt.")
rødgrød med fløde er godt.
>>> print repr("rødgrød med fløde er godt.")
'r\xf8dgr\xf8d med fl\xf8de er godt.'
```

eks. 3 center(bredde)funktionen placerer en delstreng i en streng, hvis længde angives af det tal, der indsættes i parameterlisten her markeret med variablen bredde.

```
>>> import string
>>> streng = "Rønne er hovedstaden på Bornholm."
>>> s = streng.center(60)
>>> s
'             R\xf8nne er hovedstaden p\xe5 Bornholm.             '
>>> print s
             Rønne er hovedstaden på Bornholm.
```

```
Samme eksempel mere "naturlig":
>>> import string
>>> print "Rønne er hovedstaden på Bornholm.".center(60)
             Rønne er hovedstaden på Bornholm.
```

eks. 4// count(delstreng,begyndMed,slutMed) tæller antal forekomster af delstreng i streng.

```
>>> import string
>>> streng = "Rønne og Åkirkeby er to bornholmske byer."
>>> # Da der ikke er angivet hverken start- eller slutpunkt
>>> # tælles der op i hele strengen.
>>> streng.count("e")
5
>>> streng.count("e",6,15)
1
```

```
Eller:
>>> import string
>>> "Rønne og Åkirkeby er to bornholmske byer.".count("e")
5
```

eks. 5 endswith(delstreng, startplads, slutplads) funktionen fortæller, det den siger - om det er sandt/falskt at strengen slutter delstrengen. Som vi har set andre steder, så er 1 lig med sand, mens 0 betyder falsk.

```
>>> import string
>>> streng = "Hammershus er Nordeuropas største borgruin."
>>> streng.endswith(".")
1
>>> streng.endswith("n",10, len(streng) - 1)
1
>>> streng.endswith("n",10,12)
0
```

```
Eller:
```

```
>>> import string
>>> s1 = "Hammershus er Nordeuropas største borgruin.".endswith(".")
>>> strenglengde = len("Hammershus er Noreuropas største borgruin")
>>> s2 = "Hammershus er Nordeuropas største borgruin.".endswith("n",10,strenglengde)
>>> s3 = "Hammershus er Nordeuropas største borgruin.".endswith("n",10,12)
>>> print s1, s2, s3
True True False
>>> s1, s2, s3
(True, True, False)
```

Det første af de to eksempler er udført i Python 2.2, det sidste i Python 2.3 derfor forskellen i returneringen af sandhedsværdierne.

eks. 6// expandtabs([antalTomme]) returnerer en ny streng, hvori alle tabulatorstop er erstattet af mellemrum. Det valgfrie argument antalTomme specificerer antal mellemrum der erstatter en tabulatorindrykning. Det forvalgte antal er 8 tegn mellem hvert tabulatorstop. Bemærk \t for tabulator.

>>> import string

```
>>> t = "En streng\t streng"
>>> t
'En streng\t streng'
>>> t.expandtabs(30)
'En streng                       streng'
```

eks. 7 find(delstreng,startplads, slutplads) viser en delstrengs position i en streng.

```
>>> import string
>>> streng = "Børnene leger i skolegården."
>>> streng.find("ø")
1
>>> streng.find("å")
22
```

Hvis du tæller efter, synes det som om Python er gal på den - kan Python ikke regne? Jo, men det er vist et faktum i alle normale computersprog, at en streng er en form for tabel, og så er den god nok, for det er vist også fast, at en tabels første værdi er placeret som tabelelement nummer nul. Det betyder, at ø reelt

er element nummer 1 og å element nummer 22 som vist ovenfor. Du kan opnå den "rigtige" position således:

```
>>> streng.find("ø") + 1
2
```

```
Funktionen returnerer -1, hvis ikke delstrengen findes.
>>> streng.find("ø",2,)
-1
```

eks. 8 index(delstreng, startplads, slutplads) viser en delstrengs position i en streng. Udfører samme funktion som find(delstreng, startplads, slutplads), men i stedet for at returnere -1 returneres en ValueError exception, hvis ikke delstrengen findes i strengen.

```
>>> import string
>>> streng = "Børnene leger i skolegården."
>>> streng.index("ø")
1
>>> streng.index("å")
22
>>> streng.index("q")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: substring not found in string.index
```

Python melder to ting 1: fejlen findes i linje 1 (tabelcelle 2) og 2: delstrengen "ø" blev ikke fundet.

eks. 9 isalnum() returnerer 1 (sand), hvis strengen ene og alene indeholder alfanummeriske tegn (bogstaver og tal) ellers returneres nul.

```
>>> import string
>>> streng = "Bornholm2003"
>>> streng.isalnum()
1
>>> streng = "Bornholm 2003"
>>> streng.isalnum()
0
```

Der er en ASCII 32 (mellemrum) i strengen derfor 0.

```
eller:
>>> import string
>>> print "Bornholm2003".isalnum(),"Bornholm 2003".isalnum()
True False
```

eks. 10 isalpha() returnerer 1 hvis strengen ene og alene indeholder alfabetiske karakterer (bogstaver) ellers returneres 0

```
>>> import string
>>> streng = "Børnenelegeriskolegården"
>>> streng.isalpha()
1
>>> streng = "Børnene leger i skolegården."
>>> streng.isalpha()
0
```

eks. 11 isdigit() returnerer 1, hvis strengen ene og alene indeholder tal, ellers returneres 0.

```
>>> import string
>>> streng ="0123456789"
>>> streng.isdigit()
1
>>> streng ="0123456789 "
>>> streng.isdigit()
0
>>> streng ="0.123456789"
>>> streng.isdigit()
0
```

eks. 12 islower() funktionen returnerer 1, hvis alle bogstaver i strengen er "små".

```
>>> import string
>>> streng = "det er efterår nu"
>>> streng.islower()
1
```

```
>>> streng = "Det er efterår nu"
>>> streng.islower()
0
>>> streng = "vi er i efteråret 2003"
>>> streng.islower()
1
```

```
eller:
>>> import string
>>> "vi er i efteråret 2003.".islower()
True
```

eks. 13 isspace() returnerer 1, hvis strengen ene og alene indeholder tomme (ASCII 32) ellers returneres 0.

```
>>> streng = ""
>>> streng.isspace()
0
>>> streng = " "
>>> streng.isspace()
1
>>> streng = "    "
>>> streng.isspace()
1
>>> streng = " 1 "
>>> streng.isspace()
0
>>> streng = " K "
>>> streng.isspace()
0
```

eks. 14 istitle() I en engelsk bogtittel begynder det enkelte ord (som regel) med "store" bogstaver, deraf funktionsnavnet. Selve funktionen returnerer derfor 1, hvis der ene og alene forekommer "store" bogstaver som de enkelte ords første bogstav ellers returneres 0. Det kan føre til, at selv velkendte stavemåder kan returnere nul og omvendt, hvad dette eksempel vil vise:

```
>>> # Først den forkerte stavemåde:
>>> streng = "Suse Linux"
>>> streng.istitle()
1
```

```
>>> # Så den rigtige stavemåde:
>>> import string
>>> streng = "SuSE Linux"
>>> streng.istitle()
0
```

eks. 15// isupper() funktionen returnerer 1, hvis alle bogstaver i strengen er "store".

```
>>> import string
>>> streng = "SuSE Linux" # normale skrivemåde
>>> streng.isupper()
0
>>> streng = "SUSE LINUX" # unormal skrivemåde
>>> streng.isupper()
1
>>> streng = "SUSE LINUX 8.2"
>>> streng.isupper()
1
```

eks. 16 join(sekvens) knytter en strengliste (eller en delstreng) sammen med streng som "ordstyrer" og danner en lang streng.

```
>>> import string
>>> strengliste  = ["1","2","3","4","5","6"]
>>> streng
’123456’
```

eller: >>> streng = "".join(strengliste) + str(89) >>> streng ’12345689’

```
eller:
>>> ". listeelement   ".join(strengliste) + ". listeelement."
’1. listeelement   2. listeelement   3. listeelement
4. listeelement   5. listeelement   6. listeelement.’
>>>
```

eks. 17 ljust(bredde) ligner center(bredde) funktionen, hvis anvendelse tidligere er vist. Her placeres delstrengen bare venstrejusteret i den tomme streng, hvis længde angives med breddeargumentet.

```
>>> import string
>>> streng = "Venstrejusteret".ljust(20)
>>> streng
'Venstrejusteret     '
```

eks. 18// lower() lower() konverterer alle bogstaver til "små".

```
>>> import string
>>> "DETTE ER KUN EN TESTSTRENG".lower()
'dette er kun en teststreng'
```

```
"Der skal 12 til et dusin.".lower()
'der skal 12 til et dusin.'
```

eks. 19// lstrip() fjerner tomme (ASCII 32) i strengens begyndelse - venstre side.

```
>>> import string
"    Dette er en teststreng.".lstrip()
'Dette er en teststreng.'
```

eks. 20 replace(gamle,nye,maksimale) erstatter en "gammel" tekstsreng med en ny. Hvis du ikke ønsker at udskifte samtlig eksistende "gamle" med nye, kan et maksimalt antal udskiftninger angives.

```
>>> import string
>>> "En plads i haven".replace("hav","sol")
'En plads i solen'
```

```
>>> 20 udskiftninger er ikke mulig, da der kun er 1
>>> "En plads i haven".replace("hav","sol",20)
'En plads i solen'
```

eks. 21 rfind(delstreng , startplads, slutplads) finder positionen af den sidste delstreng i en streng. Hvis delstrengen ikke findes returneres -1.

```
>>> import string
>>> streng = "Der er en klippefast grund til at besøge Bornholm."
>>> streng.rfind("e")
39
```

```
eller:
>>> import string
>>> "Der er en klippefast grund til at besøge Bornholm.".rfind("e")
39
```

```
eller:
>>> import string
>>> l = len("Der er en klippefast grund til at besøge Bornholm.")
>>> "Der er en klippefast grund til at besøge Bornholm.".rfind("e",len(streng) - 10,l)
-1
```

eks. 22 rindex(delstreng,startpunkt,slutpunkt) virker fuldstændig som rfind funktionen, men i stedet for at returnere -1, hvis delstrengen ikke findes returneres en ValueError.

```
>>> import string
>>> streng = "Tidligere anvendtes Dueoddes sand i timeglas."
>>> streng.rindex("i")
37
>>> streng.rindex("ven")
12
>>> streng.rindex("dk")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: substring not found in string.rindex
```

Fejlmeldingen siger, at delstrengen "dk" ikke findes i streng.

eks. 23// rjust(bredde) svarer til center(bredde) og ljust(bredde) bortset fra at delstrengen her placeres højrejusteret i strengen.

```
>>> import string
>>> "h".rjust(30)
'                             h' # Et enkelt bogstav er også en streng.
```

eks. 24// rstrip() fjerner tomme (ASCII 32) i HØJRE side af en streng.

```
>>> # Python 2.2 afvikling:
>>> import string
>>> "    En unødvendig lang streng        ".rstrip()
'    En un\xf8dvendig lang streng'
>>> # Python 2.3 afvikling:
>>> import string
>>> print "    En unødvendig lang streng        ".rstrip()
    En unødvendig lang streng
```

eks. 25 split(seperator) Opdeler tekststreng i delstrenge, der indsættes i liste. Hvis ikke der indsættes seperator deles ved ASCII 32 (mellemrum).

```
>>> import string
>>> "Dette er kun en teststreng.".split()
['Dette', 'er', 'kun', 'en', 'teststreng.']
```

```
>>> "Dette\n er\n kun\n en teststreng.".split("\n")
['Dette', ' er', ' kun', ' en teststreng.']
```

eks. 26 split(seperator) Retuarnerer en liste af delstrenge dannet ved at dele den oprindelige streng ved hver seperator. Hvis seperatoren ikke anvendes eller er sat til None, deles strengen ved hver mellemrum (ASCII 32)

```
>>> import string
>>> "Tejn\nog\nGudjem\n er\nvelkendte turistbyer.".split()
['Tejn', 'og', 'Gudjem', 'er', 'velkendte', 'turistbyer.']
>>> "Tejn\nog\nGudjem\n er\nvelkendte turistbyer.".split("\n")
['Tejn', 'og', 'Gudjem', ' er', 'velkendte turistbyer.']
```

eks. 27 splitlines([1])returnerer en liste af delstrenge fremkommet ved at dele den oprindelige streng ved hver ny linje karakter. Hvis parameteret 1 indsættes bevares ny linje skiftet i delstrengene se nederste del af eksemplet.

```
>>> import string
>>> "Dette er\nen meget \nlang streng, \n der deles\ni delstrenge.".splitlines(1)
['Dette er\n', 'en meget \n', 'lang streng, \n', ' der deles\n', 'i delstrenge.']
>>> "Dette er en meget lang streng, der deles i delstrenge.".splitlines()
['Dette er en meget lang streng, der deles i delstrenge.']
>>> "Dette er\nen meget \nlang streng, \n der deles\ni delstrenge.".splitlines()
['Dette er', 'en meget ', 'lang streng, ', ' der deles', 'i delstrenge.']
```

eks. 28 startswith(delstreng,startpunkt,slutpunkt) returnerer 1, hvis strengen begynder med delstrengen ellers nul.

```
>>> import string
>>> "Denne streng er en violinstreng.".startswith("Den")
1
>>> "Denne streng er en violinstreng.".startswith("en")
0
```

eks. 29// strip() fjerner mellemrum i strengens begyndelse og slutning

```
>>> # Python 2.2 afvikling:
>>> import string
>>> "      Har du været på Christiansø?     ".strip()
'Har du v\xe6ret p\xe5 Christians\xf8?'
>>> import string
>>> "      Har du været på Christiansø?     ".strip()
'Har du v\xe6ret p\xe5 Christians\xf8?'
```

```
>>> print "      Har du været på Christiansø?     ".strip()
Har du været på Christiansø?
```

eks. 30// swapcase() konverterer alle "store" bogstaver til små og omvendt.

```
>>> import string
>>> "dANMARK ER VEL TRODS ALT ET GODT LAND.".swapcase()
'Danmark er vel trods alt et godt land.'
>>> "eR DANSKERNE ALT?".swapcase()
'Er danskerne alt?'
```

eks. 31 title() Returnerer en streng, hvori første bogstav af strengens enkeltord og kun dem er med stort begyndelsesbogstav.

```
>>> import string
>>> "er uldjyder lådne?".title()
'Er Uldjyder L\xe5dne?'
```

>>> "eR brAndMænd meget iltre?".title() 'Er Brandm\xe6nd Meget Iltre?'

```
eller:
>>> import string
>>> "er uldjyder lådne?".title()
'Er Uldjyder L\xe5dne?'
>>> s = "er uldjyder lådne?".title()
>>> print s
Er Uldjyder Lådne?
```

eks. 32// \upper() konverterer alle bogstaver til "store" bogstaver

```
>>> import string
>>> "var forfatteren nexø fra nexø?".upper()
'VAR FORFATTEREN NEX\xd8 FRA NEX\xd8?'
```

eller: >>> import string >>> "var forfatteren nexø fra nexø?".upper() 'VAR FORFATTEREN NEX\xd8 FRA NEX\xd8?' >>> print "var forfatteren nexø fra nexø?".upper() VAR FORFATTEREN NEXØ FRA NEXØ?

Brødmaskinen (slicing): >>> streng = "naturlig" >>> streng[4] 'r' >>> streng[0:2] 'na' >>> streng[2:4] 'tu' >>> streng[:2] 'na' >>> streng[2:] 'turlig' >>> 'u' + streng[1:] 'uaturlig' >>> streng[:2] + streng[2:] 'naturlig' >>> streng[:3] + streng[3:] 'naturlig' >>>

```
zfill funktionen gør det, den siger fylder op med zeros (nuller)
opfyldningen sker altid til venstre:
import string
string.zfill('12', 5)
'00012'
string.zfill('-3.14', 7)
'-003.14'
string.zfill('3.14159265359', 5)
'3.14159265359'
```

3.3. Range

```
syntaks: range(begynd, slut - 1, step) - slut er nødvendig.
>>> range(11)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9.10]
>>> range(0,21,2)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>>> range(0,40,3)
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39]
>>> range(1,-11,-1)
[1, 0, -1, -2, -3, -4, -5, -6, -7, -8, -9, -10]
>>> range(40,0,-5)
[40, 35, 30, 25, 20, 15, 10, 5]
>>> for i in range(1,11):
...     print i,
...
1 2 3 4 5 6 7 8 9 10
Range fungerer også ved nedtælling:
>>> for i in range(10,0, -1):
...     print i,
...
10 9 8 7 6 5 4 3 2 1
>>> for i in range(100,0, -5):
...     print i,
...
100 95 90 85 80 75 70 65 60 55 50 45 40 35 30 25 20 15 10 5
>>>
```

```
2 metoder til udskrift af kvadrat- og kubiktal:
>>> import string
>>> for x in range(1, 11):
...     print string.rjust(repr(x), 2), string.rjust(repr(x*x), 3),
...     print string.rjust(repr(x*x*x), 4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
>>> for x in range(1,11):
...     print '%2d %3d %4d' % (x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

3.4. Læs og skriv filer

```
heleTeksten= open('filnavn.txt').read()    # hent hele indholdet i en tekstfil
alleData = open('fuldeFilsti', 'rb').read() # hent hele indholdet i en binær fil

filObjekt  = open('filnavn.txt')
heleTeksten = filObjekt.read()
filObjekt.close()
listeAfAlleLinjer = filObjekt.readlinjes()
listeAfAlleLinjer = filObjekt.read().splitlinjes(1)
listeAfAlleLinjer = filObjekt.read().splitlinjes()
listeAfAlleLinjer = filObjekt.read().split('\n')
listeAfAlleLinjer = list(filObjekt)

filObjekt = open('fuldeFilsti', 'rb')
while 1:
    blok = filObjekt.read(100)
```

```
    if not blok: break
    anvend blokken (f.eks. til udskrift)
filObjekt.close()

for linje in open('filnavn.txt'):
    print linje

filObjekt = open('filnavn.txt')
while 1:
    linje = filObjekt.readlinje()
    if not linje: break
    print linje
filObjekt.close()
```

```
open('filnavn.txt', 'w').write(heleTeksten)  # skriv tekst til en tekstfil
open('fuldeFilsti', 'wb').write(alleData)    # skriv binære data til en binær fil

filObjekt = open('filnavn.txt', 'w')
filObjekt.write(heleTeksten)
filObjekt.close()

filObjekt.writelines(listeAfAlleTekststrenge)
open('fuldeFilsti', 'wb').writelines(listeAfAlleDataStrenge)
```

3.4.1. Hent fil linje for linje

```
import linecache

fuldeFilsti = "autoexec.bat"
valgteLinjenummer = 0

linjen = linecache.getline(fuldeFilsti, valgteLinjenummer)

# getline er fordefineret i Python
def getline(fuldeFilsti, valgteLinjenummer):
    if valgteLinjenummer < 1: return ''
    aktuelleLinjeNummer = 0
    for linje in open(fuldeFilsti):
        aktuelleLinjeNummer += 1
        if aktuelleLinjeNummer == valgteLinjenummer: return linje
    return ''

for linje in open(fuldeFilsti).xreadlines():
 print linje
```

3.4.2. Find antal linjer i tekstfil

```
import linecache
fuldeFilsti = "autoexec.bat"

antalLinjer = len(open(fuldeFilsti).readlines())
print antalLinjer
```

3.4.3. Skriv til udfil

```
import sys
try:
   udfil = open( "filnavn.txt", "w" )
except IOError, melding:
    print >> sys.stderr, "Fejlmelding:", melding
    sys.exit( 1 )

udtekst = ""
tekst = " "

while len(tekst) > 0:
 tekst = raw_input( "Skriv tekstlinje (tom lukker fil): " )
 udtekst += tekst + "\n"

print >> udfil, udtekst        # skriv tekst til udfil
udfil.close()
print "Udfilen er lukket."
```

Kørselsresultat:

```
python udfilNy.py
Skriv tekstlinje (tom lukker fil): Ny tekst i ny udfil."
Skriv tekstlinje (tom lukker fil):
Udfilen er lukket.
```

Indhold i den nyoprettede filnavn.txt Ny fil i ny udfil

3.4.4. Hent fil i Python (Linux) bibliotek

```
import sys
try:
   indfil = open( "/usr/lib/python2.2/calendar.py", "r" )
```

```
except IOError:
   print >> sys.stderr, "Filen blev ikke hentet ind."
   sys.exit( 1 )

indtekst = indfil.read()
print str(indtekst)
indfil.close()
print
print "Filen er lukket."

import sys
try:
   udfil = open( "filnavn.txt", "wb" )
except IOError, melding:
    print >> sys.stderr, "Fejlmelding:", melding
    sys.exit( 1 )

udtekst = ""
tekst = " "

while len(tekst) > 0:
 tekst = raw_input( "Skriv tekstlinje (tom lukker fil): " )
 udtekst += tekst + "\n"

print >> udfil, str(udtekst )      # skriv tekst til udfil
udfil.close()
print "Udfilen er lukket."
```

3.4.5. Gem liste i fil

```
import sys
try:
   udfil = open( "filnavn.txt", "w" )
except IOError, melding:
    print >> sys.stderr, "Fejlmelding:", melding
    sys.exit( 1 )
```

liste = [1,2,3,4,5,6,7,8,9]

print >> udfil, str(liste) # skriv liste til udfil udfil.close() print "Udfilen er lukket."

filnavn.txt kontrol viser indholdet: [1, 2, 3, 4, 5, 6, 7, 8, 9]

3.4.6. Udvid fil (append)

```
import sys
try:
   udfil = open( "filnavn.txt", "a" )
except IOError, melding:
    print >> sys.stderr, "Fejlmelding:", melding
    sys.exit( 1 )
udtekst = ""
tekst = " "
while len(tekst) > 0:
 tekst = raw_input( "Skriv tekstlinje (tom lukker fil): " )
 udtekst += tekst + "\n"
print >> udfil, str(udtekst )      # skriv tekst til udfil
udfil.close()
print "Udfilen er lukket."
```

3.4.7. Læs fra fil

```
import sys
try:
   udfil = open( "filnavn.txt", "w" )
except IOError, melding:
    print >> sys.stderr, "Fejlmelding:", melding
    sys.exit( 1 )

udtekst = ""
tekst = " "

while len(tekst) > 0:
 tekst = raw_input( "Skriv tekstlinje (tom lukker fil): " )
 udtekst += tekst + "\n"

print >> udfil, str(udtekst )      # skriv tekst til udfil
udfil.close()
print "Udfilen er lukket."
```

BEMÆRK: str funktionen kræves a.h.t. dansk/norske specialtegn.

Kørselsresultat: ajbo@linux:~> python indfil.py ['Microsoft får en ordentlig bredside i en ny rapport fra den amerikanske \n', 'organisation Computer & Communications Industry Association (CCIA). Manglen \n', 'på ordentlig sikkerhed i Microsofts programmer og det faktum, at Windows er \n', 'det mest udbredte styresystem i verden, betyder tilsammen, at verden generelt \n', 'er blevet mere sårbar overfor trusler mod sikkerheden, konkluderer rapporten.\n']

Filen er lukket. ajbo@linux:~>

Teksten er uddrag fra en Netavisen Infopaq mail udsendt/modtaget 24. sept 03

3.4.8. Skriv bytes til udfil

```
import sys
try:
   udfil = open( "filnavn.txt", "wb" )
except IOError, melding:
    print >> sys.stderr, "Fejlmelding:", melding
    sys.exit( 1 )

udtekst = ""
tekst = " "

while len(tekst) > 0:
 tekst = raw_input( "Skriv tekstlinje (tom lukker fil): " )
 udtekst += tekst + "\n"

print >> udfil, str(udtekst )      # skriv tekst til udfil
udfil.close()
print "Udfilen er lukket."
```

Fra kørselsresultat: filnavn.txt indeholder: Nu går vi over til at udsende teksten i antal bytes. Flere bytes skrives til filen.

3.4.9. Hent bytes fra fil

```
import sys

try:
   indfil = open( "filnavn.txt", "rb" )
except IOError:
   print >> sys.stderr, "Filen blev ikke hentet ind."
   sys.exit( 1 )

indtekst = indfil.readlines()
print str(indtekst)
indfil.close()
print
print "Filen er lukket."
```

Kørselsresultat:

```
ajbo@linux:~> python indfilBytes.py
['Nu går vi over til at udsende teksten i antal bytes.\n',
 'Flere bytes skrives til filen.\n', '\n', '\n']
Filen er lukket.
ajbo@linux:~>
```

3.4.10. Hent bytes fra fil eksempel 2

Til dette eksempel har jeg benyttet command.com fra Microsoft Windows Me

```
import sys

try:
 indfil = open( "/windows/C/command.com", "rb" )
except IOError:
    print >> sys.stderr, "Filen blev ikke hentet ind."
    sys.exit( 1 )

indtekst = indfil.read()
print str(indtekst)
indfil.close()
print
print "Filen er lukket."
```

Kørselsresultat (uddrag af den store udskrift): enhed Den terminalenhed, som du vil bruge, f.eks. COM1. 1Viser eller indstiller datoen.

DATE [dato]

Skriv DATE uden parametre for at f vist den aktuelle datoindstilling og for at kunne angive en ny. Tryk p ENTER for at beholde den nuvrende dato. aSletter en eller flere filer.

3.4.11. Hent bytes fra fil eksempel 3

I dette eksempel læses filen vinduer.jpg, der også benyttes et andet sted i bogen.

```
import sys
try:
   indfil = open( "vinduer.jpg", "rb" )
```

```
except IOError:
   print >> sys.stderr, "Filen blev ikke hentet ind."
   sys.exit( 1 )

indtekst = indfil.read()
print str(indtekst)
indfil.close()
print
print "Filen er lukket."
```

Kørselsresultat (ganske kort uddrag af returgodset)

```
python indfilBytes3.py
ÿØÿàJFIFHHÿþLEAD Technologies Inc. V1.01ÿUC
```

3.4.12. Søg og erstat tekst i fil

```
#!/usr/bin/env python
import os, sys

antalArgumenter = len(sys.argv)

if not 3 <= antalArgumenter <= 5:
    print "Anvend: %s tekstAtFinde erstatningsTekst [indfil [udfil]]" % \
        os.path.basename(sys.argv[0])
else:
    findeTekst = sys.argv[1]
    erstatteTekst = sys.argv[2]
    input = sys.stdin
    output = sys.stdout
    if antalArgumenter > 3:
        input = open(sys.argv[3])
    if antalArgumenter > 4:
        output = open(sys.argv[4], 'w')
    for s in input.xreadlines():
        output.write(s.replace(findeTekst, erstatteTekst))
    output.close()
    input.close()
```

```
 Et kørselsresultat:
python erstatTekst.py
Anvend: erstatTekst.py tekstAtFinde erstatningsTekst [indfil [udfil]]
```

Når filen søges afviklet uden de nødvendige parametre udskrives der som vist en anvisning på, hvordan kaldet skal foregå. sys.argv[0] er selve programmets navn (erstatTekst.py). I Python bøger er det normalt at angive valgfrie parametre i firkantede parenteser. Det er altså i herværende tilfælde ikke absolut

nødvendigt at angive en indfils navn, det samme er tilfældet med en udfils navn. Udeladelse af de to navne er imidlertid upraktisk i det aktuelle tilfælde, så et kald kunne være:

```
python erstatTekst.py "Microsoft" "Linux" "/windows/C/autoexec.bat" "autoexec.bat"
```

Der er bare det ved det, at ordet "Microsoft" næppe forekommer i autoexec.bat filen. Resultatet vil altså alene blive at filen bliver kopieret fra Microsoft Windows biblioteket til rodbiblioteket i Linux, men det kan jo også være fint nok.

```
import os, sys, string
indfil = open( "/windows/C/autoexec.bat", "r" )
udfil = open( "autoexec.bat", "w" )
findeTekst = "WINDOWS"; erstatteTekst = "Linux"
for s in indfil.readlines():
 udfil.write(string.replace(s, findeTekst, erstatteTekst))
indfil.close
udfil.close
```

Før programafvikling: SET windir=C:\WINDOWS SET winbootdir=C:\WINDOWS SET COMSPEC=C:\WINDOWS\COMMAND.COM SET PATH=C:\WINDOWS;C:\WINDOWS\COMMAND SET PROMPT=pg SET TEMP=C:\WINDOWS\TEMP SET TMP=C:\WINDOWS\TEMP

Efter programafvikling: SET windir=C:\Linux SET winbootdir=C:\Linux SET COMSPEC=C:\Linux\COMMAND.COM SET PATH=C:\Linux;C:\Linux\COMMAND SET PROMPT=pg SET TEMP=C:\Linux\TEMP SET TMP=C:\Linux\TEMP

De to linjer: for s in indfil.readlines(): udfil.write(string.replace(s, findeTekst, erstatteTekst))

Kan også skrives: for s in indfil: udfil.write(s.replace(findeTekst, erstatteTekst))

Lad Python fortælle, hvad et modul indeholder:

```
>>> sys
<module 'sys' (built-in)>
```

returstrengen fortæller, at sys er et i Python indbygget (fordefineret) modul. sys inkluderes i dette eksempel ene og alene for at kunne lukke programmet ned, når der klikkes på knappen. Herved udføres kommandoen root.exit

I interaktiv mode kan det undertiden være vanskeligt at få kørslen stoppet og komme ud af programmet. I Windows kan du komme ud ved at trykke på Ctrl og Z tasten samtidig. I Linux skulle du kunne komme ud ved at trykke samtidig på Ctrl og D. I SuSE 8.2 skal du imidlertid trykke Ctrl C for at komme ud. Men det korte og det lange er, at sys.exit og nævnte tastetryk har samme virkning.

Vi kan også få at se, hvad modulet indeholder:

```
>>> import sys
>>> dir (sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',
'__stdin__', '__stdout__', '_getframe', 'argv', 'builtin_module_names',
'byteorder', 'copyright', 'displayhook', 'dllhandle', 'exc_info',
'exc_type', 'excepthook', 'exec_prefix', 'executable', 'exit',
'getdefaultencoding', 'getrecursionlimit', 'getrefcount',
'hexversion', 'last_traceback', 'last_type', 'last_value',
'maxint', 'maxunicode', 'modules', 'path', 'platform', 'prefix', 'ps1',
'setcheckinterval', 'setprofile', 'setrecursionlimit', 'settrace', 'stderr',
'stdin', 'stdout', 'version', 'version_info', 'warnoptions', 'winver']
```

3.5. Tidsfunktioner

Tidsfunktioner

3.5.1. Timemodulet

```
>>> import time
>>> type(time) # unødvendigt bevis for dig
<type 'module'>

>>> time.time()
1064465521.312474
>>> time.gmtime()
(2003, 9, 25, 4, 53, 17, 3, 268, 0)
>>> time.localtime()
(2003, 9, 25, 6, 53, 46, 3, 268, 1)
>>> time.asctime()
'Thu Sep 25 06:54:19 2003'
>>> t = time.localtime()
>>> aar = t[0]
>>> aar
2003
>>> maaned = t[1]
>>> maaned
```

```
9
>>> dag = t[2]
>>> dag
25

>>> # pausefunktion
>>> # kontrol :   >>>  fremkommer efter pause på 10 sekunder.
>>> time.sleep(10)
>>>
```

3.5.2. Datetime modulet

```
>>> import datetime
>>> type(datetime)  # unødvendigt bevis for dig
<type 'module'>

>>> nu = datetime.datetime.now()
>>> nu.isoformat()
'2003-09-25T05:41:32.611578'
>>> nu.ctime()
'Thu Sep 25 05:41:32 2003'
>>> nu.strftime("%D %d %b")
'09/25/03 25 Sep')
>>> nu.strftime("%D %d")
'09/25/03 25'

>>> Dag = nu.strftime("%d")
>>> Dag
'25'

>>> Timetal24 = nu.strftime("%H")
>>> Timetal24
'07'

>>> Timetal12 = nu.strftime("%I")
>>> Timetal12
'07'

>>> AmPmLokal = nu.strftime("%p")
>>> AmPmLokal
'AM'
>>> EngelskEfterskrift = nu.strftime("%H") + nu.strftime("%p")
>>> EngelskEfterskrift
'07AM'
>>> # eller:
>>> EngelskEfterskrift = nu.strftime("%H") + nu.strftime("%p")
>>> EngelskEfterskrift = nu.strftime("%H") + " " + nu.strftime("%p")
>>> EngelskEfterskrift
'07 AM'

>>> # Funktionen beregner ugenummeret 1 for lavt
```

```
>>> # derfor nedenstående omregning
>>> Ugenummer = int(Ugenummer) + 1
>>> Ugenummer
39
>>> # konverteret til streng:
>>> Ugenummer = str(Ugenummer)
>>> Ugenummer
'39'

>>> # Søndag er ugens første dag (element 0)
>>> UgedagDecimal = nu.strftime("%w")
>>> UgedagDecimal
'4'

>>> AArstalKort = nu.strftime("%y")
>>> AArstalKort
'03'

>>> AArstalLang = nu.strftime("%Y")
>>> AArstalLang
'2003'
```

Kapitel 4. Med Python på internettet

I en tid, hvor det ikke er ualmindeligt at modtage mails fra en meget stor del af den globale verden, kan det være rart at se, hvilken server, det måtte være, hvorfra der sendes til en. I Python kan det ske særdeles enklet. De tre forholdsvis ens eksempler, jeg nu vil vise, er så selvforklarende, at jeg kun vil tilføje, at Date: viser, hvornår jeg hentede de viste info ind. Dog skal tillægges 2 timer grundet forskellen i længdegrad fra Rønne til London og sommertiden. Endelig skal jeg gøre opmærksom, på at Content-Length: informationen viser det antal bytes, der er i indexfilen og ikke andet. Jeg finder det selv interessant at se, hvilken server den besøgte benytter. Jeg kunne ikke lade være med at tage såvel folketingets som Alt Om Datas udskrift med, fordi førstnævnte har vedtaget, at det offentlige skal bruge open source - og så bruger samme folketing endda en ældre version af Microsofts server (version 4) - der er ingen forbindelse mellem påstanden om at ville bruge eksempelvis Linux og så det at gøre det.

```
>>> from urllib import urlopen
>>> dokument = urlopen("http://www.python.org").read
>>> # Her er der en kort pause mens URL kontaktes
>>> dokument = urlopen("http://www.python.org")
>>> print dokument.info()
Date: Thu, 07 Aug 2003 19:13:13 GMT
Server: Apache/1.3.26 (Unix)
Last-Modified: Wed, 06 Aug 2003 23:54:30 GMT
ETag: "5a750c-3ac2-3f319536"
Accept-Ranges: bytes
Content-Length: 15042
Connection: close
Content-Type: text/html
```

```
>>> from urllib import urlopen
>>> dokument = urlopen("http://www.folketinget.dk").read
>>> dokument = urlopen("http://www.folketinget.dk")
>>> print dokument.info()
Server: Microsoft-IIS/4.0
Date: Fri, 08 Aug 2003 03:40:56 GMT
Content-Type: text/html
Set-Cookie: ASPSESSIONIDTBRRRBAB=DBJEHKFBCNBPDGMCPBIKLJJG; path=/
Cache-control: private
```

```
>>> from urllib import urlopen
>>> dokument = urlopen("http://www.aod.dk").read
>>> dokument = urlopen("http://www.aod.dk")
>>> print dokument.info()
Server: Microsoft-IIS/5.0
Date: Fri, 08 Aug 2003 03:21:45 GMT
X-Powered-By: ASP.NET
X-AspNet-Version: 1.1.4322
Set-Cookie: ASP.NET_SessionId=dmbwzz455wrgrsmlwyhtyq45; path=/
Cache-Control: private
Content-Type: text/html; charset=iso-8859-1
```

```
Content-Length: 23164

>>> import ftplib
>>> # host kan læses som stedet, hvor filen er placeret.
>>> ftp = ftplib.FTP("ftp.host.dk")
>>> ftp.login("brugernavn skal ind her","password skal ind her")
'230 User brugernavn logged in.'

Hent dokument fra url på Web
>>> from urllib import urlopen
>>> dokument = urlopen("http://www.sslug.dk").read()
>>> print dokument
```

Appendiks A. Revisionshistorie for bogen

Første udgave af denne bog er skrevet af Alfred Jensen, der har overdraget den til »Linux – friheden til ...«-projektet.

Vi frigiver ofte nye versioner, når der er kommer en del rettelser ind, eller nye afsnit er blevet skrevet. Kommentarer, ris og ros, og specielt fejl og mangler bedes sendt til linuxbog@sslug.dk (mailto:linuxbog@sslug.dk), men er du medlem af SSLUG så skriv til sslug-bog@sslug.dk (mailto:sslug-bog@sslug.dk). Her er en liste over, hvad der er ændret i bogen.

Arbejdet på bogen koordineres via SSLUG's postliste sslug-bog@sslug.dk (mailto:sslug-bog@sslug.dk). Du kan tilmelde dig postlisten på http://www.sslug.dk/tilmeld#bog (bemærk at du samtidig bliver medlem af SSLUG hvis du ikke allerede er det).

- Version 1.0.20040516 - 16. maj 2004: Jacob Sparre Andersen: Rettet sprog.
- Version 1.0 - 25. januar 2004: Jacob Sparre Andersen: Går i gang med at konvertere bogen fra LaTeX til Docbook/XML. Fanget mystisk fejl i omtale af Forth. Gitte Wange: Færdiggjort konverteringen af bogen fra LaTeX til Docbook/XML. Omstruktureret bogen.

Ordliste

fri standard

En fri standard overholder følgende krav:

1. Ingen restriktion på anvendelse.
2. Ingen restriktion på implementation.
3. Bevarelse af standardens integritet.
4. Fri tilgængelighed til standarden.

ad 1) En standard må ikke sætte begrænsninger på hvor standarden kan finde anvendelse, og heller ingen begrænsninger på hvad man må gøre ud over hvad standarden foreskriver. (cirka open source-definitionen punkt 5 og 6)

ad 2) Der må ikke være restriktioner på hvem der må implementere standarden, hvordan den implementeres, eller hvordan implementationen licensieres. Dette hindrer blandt andet at åbne standarder anvender patenterede algoritmer. (cirka open source-definitionen punkt 5 og 6, giver mulighed for at anvende OSD-licenser på implementationen)

ad 3) En standard må kræve at der tydeligt skal gøres opmærksom på hvor implementationen går ud over hvad standarden implementerer. (cirka open source-definitionens punkt 4)

ad 4) Det skal være tilladt af videredistribuere hele eller dele af standarden. (cirka open source-definitionens punkt 1)

Se i øvrigt Peter Makholms skriblerier om åbne og frie standarder (http://peter.makholm.net/skriblerier/openstandards) for en uddybning af denne definition.

frie programmer

Programmer der distribueres efter "Open Source"-reglerne (http://www.opensource.dk/docs/definition.html). Kort fortalt går det ud på at hvis du har fået et *frit program*, så:

- Må du frit sælge eller videregive programmet.
- Har du ret til at få kildeteksten til programmet.
- Har du ret til at videreudvikle programmet.

GNU Compiler Collection (GCC)

GNU-projektets oversættersystem. Det kan oversætte adskillige programmeringssprog til nok endnu flere forskellige platforme. Fidusen ved GCC er at oversættersystemet er delt i to dele; "forenden", hvor der findes en til hvert programmeringssprog, og "bagenden", hvor der findes en til hver platform. Og kommunikationen mellem for- og bagende sker på en form der er uafhængig af både programmeringssprog og platform.

åben standard

En åben standard overholder de første tre krav til en *fri standard*, men behøver ikke nødvendigvis at kunne distribueres frit.

Stikordsregister

Symboler

C

F

H

O

P

R