Innehåll

1 Python för den som kan programmera

1.1 Hur man blir pythonkramare

Många javaprogrammerare blir pythonbitna när dom ser hur mycket enklare koden blir i Python. Här är först javafilen `Candles.java` med körexempel.

```
import java.io.*;
class Candles{
    public static void main(String[] args){
        BufferedReader in = new BufferedReader(
                            new InputStreamReader(System.in));
        System.out.print("Födelseår:");
        try{
            String born = in.readLine();
            int year=Integer.parseInt(born)+1;
            System.out.println("Minns du ljusen på dina födelsedagstårtor?");
            String candles="|";
            while(year<2004){
                System.out.println(year+" "+candles);
                year++;
                candles+="|";
            }
            System.out.println("och i år...");
            System.out.println(year+" "+candles);
        }
        catch(Exception e){
            System.out.println(e);
        }
    }
}
```

```
Födelseår: 1998
Minns du ljusen på dina födelsedagstårtor?
1999 |
2000 ||
2001 |||
2002 ||||
2003 |||||
och i år...
2004 ||||||
```

Javafilens längd är sexhundra tecken och mycket kortare än så blir inte javaprogram. Men samma program i Python blir bara tvåhundra tecken långt, nämligen så här.

```
born = input("Födelseår:")
print "Minns du ljusen på dina födelsedagstårtor?"
year = born + 1
candles="|"
while year < 2004 :
    print year, candles
    year += 1
    candles += "|"
print "och i år..."
print year, candles
```

Pythonprogrammet är det som finns kvar när man befriat javaprogrammet från skräp som semikolon, måsvingar, deklarationer, `class`, `main`, `public`, `static`, `void` och så vidare. Resultatet blir onekligen mer lättläst.

Om nu Python är enklare än Java och ändå lika kraftfullt, varför är det då fortfarande mest javaprogrammerare som efterfrågas på arbetsmarknaden? Till stor del är det säkert tröghet men det finns också rationella anledningar. I ett stort projekt med många programmerare är det större risk för oupptäckta buggar i pythonkod än i javakod. Pythonkramarna menar dock att den tid man sparar på programmeringen kan läggas på testkörningar och att slutprodukten därför blir buggfriare!

Resten av detta kapitel är en snabbkurs i pythonspråket för den som redan kan programmera. Förklaringar kommer i senare kapitel. Den fullständiga beskrivningen av pythonspråket och pythonbiblioteken finns på webben `<http://www.python.org>`.

1.2 Inläsning och utskrift

Printsatser skriver ut talvärden och text. Kommatecken ger ett mellanslag. Om printsatsen slutar med ett komma blir man kvar på samma rad, annars blir det ny rad.

```
a=4
print a*a+1, "uttalas sjutton"   =>  17 uttalas sjutton
print "Hipp hipp",
print "hurra!"                   =>  Hipp, hipp hurra!
```

Talvärden läses in med `input()` och text med `raw_input()`. Inom parenteserna kan man skriva en uppmaning. Returtryckning avslutar inmatningen men följer inte med in i texten.

```
namn=raw_input("Namn: ")         =>  Namn: Henrik
x=input("Längd(cm): ")           =>  Längd(cm): 188
print x, "cm", namn, "väger:",
v=input()                        =>  188 cm Henrik väger: 78.5
print "BMI:", v*10000/(x*x)      =>  BMI: 22.2102761431
```

För att skriva ut tal med önskat antal siffror används procentformat som i C. Om det inne i en text står `%7d` betyder det att sju platser avsätts för ett heltalsvärde som anges senare. Om det står `%5.1f` avsätts fem platser för ett decimaltal som ska skrivas ut med en decimal. Satsen ovan blir då
`print "BMI:%5.1f" %(v*10000/(x*x)) => BMI: 22.2`

Formatet `%9.2e` avsätter nio platser för ett decimaltal med två decimaler följt av tioexponent, och formatet `%8s` åtta platser för en textsträng:
`print "Du %8s är %9.2e cm" %(namn,x) => Du  Henrik är 1.88e+002 cm`
Inläsning från fil och utskrift på fil görs på följande sätt.

```
prisfil=file("priser.dat")          =>  Öppnas för läsning
grisfil=file("grisar.ut","w")       =>  Öppnas för skrivning
rad=prisfil.readline()              =>  En filrad blir en textsträng
rader=prisfil.readlines()           =>  Hela filen blir en lista av strängar
print>>grisfil, "Grispris:", rad    =>  print fungerar som vanligt...
grisfil.writelines(rader)           =>  ...men writelines finns också
grisfil.close()                     =>  Stäng färdigskriven fil
```

1.3 Textsträngar och konvertering

Textsträngar skrivs inom citattecken. Radbyten anges som `\n`.

```
print "Hälsningar \n Vahid M"     =>     Hälsningar
                                          Vahid M
```

Småbitar ur textsträngar kan kopieras och klistras ihop enligt nedan. Observera att numreringen börjar från noll och att utklipp anges av två tal: det första index som tas med och det första index som inte ska tas med. Sista tecknet kan också numreras -1, näst sista -2 osv.

```
pryl = "Elvisp"
print "längd:", len(pryl)            =>  längd: 6
print pryl[1]+pryl[5]                =>  lp
print pryl[0:5]                      =>  Elvis
print pryl.upper()                   =>  ELVISP
print pryl.lower()                   =>  elvisp
print pryl.find("v")                 =>  2
print "sista:", pryl[-1]             =>  sista: p
klok = pryl[2:-1]
print "Den råd lyder är", klok       =>  Den råd lyder är vis
print "rade"*2                       =>  raderade
```

I avsnitt 1.5 tar vi upp `split`/`splitlines` som skapar en lista av ingående ord/rader i en lång text.

För att säga vad `x + y` eller `z*2` innebär måste man veta om variablernas värden är tal eller text. Typkonvertering är enkelt.

```
x = 1;  y = 7                       ...två talvärden
print x, "och", y, "är", x+y  =>  1 och 7 är 8
x = str(x); y = str(y)              ...två textsträngar
print x, "och", y, "är", x+y  =>  1 och 7 är 17
x = int(x)                          ...ett tal och en text
print x, "och", y, "är", x+y  =>  TypeError for +: 'int' and 'str'
```

1.4 `if`-satser och `while`-slingor

En `if`-sats anger ett villkor och sedan följer en eller flera satser som ska utföras om villkoret är uppfyllt.

```
x = input("Vad är klockan? ")
if x<10:     print "God morgon!"
if 10<=x<12: print "God förmiddag!"
if 12<=x<18: print "God eftermiddag!"
if 18<=x<22: print "God kväll!"
if 22<=x:    print "God natt!"
```

När högst ett av villkoren kan uppfyllas är `elif`-satser bra. En `else`-sats på slutet fångar upp det som inte fastnat i något av alternativen.

```
if x<10:     print "God morgon!"
elif x<12:   print "God förmiddag!"
elif x<18:   print "God eftermiddag!"
elif x<22:   print "God kväll!"
else:        print "God natt!"
```

Det vanligaste felet i `if`-satser visar vi här:
`if x=12: print "God middag!" => SyntaxError: invalid syntax`
Det skulle ha stått dubbla likhetstecken: `if x==12`, det enkla likhetstecknet betyder ju tilldelning. För skildfrånvillkor används antingen `x<>12` eller `x!=12`.

Oftast styr villkoret flera följande satser, och då anger man med indragen hur långt villkoret sträcker sej.

```
if x<17:
    print "Goddag!"
    print "Vi stänger om", 17-x, "timmar"
else:
    print "Adjö!"
    print "Vi stängde för", x-17, "timmar sedan"
```

En `while`-slinga fungerar likadant som en `if`-sats men upprepas tills villkoret inte är uppfyllt.

```
rätt = 17
gissat = input("Gissa ett heltal:")
while gissat<>rätt:
    if gissat<rätt:
        print "För liten gissning!"
    else:
        print "För stor gissning!"
    gissat = input("Gissa igen:")
print "Rätt, men tyvärr en gissning för mycket för att få pris."
```

Villkoret kollas före varje varv i slingan men ibland vill man bryta sej ur när man är mitt inne i slingan. Det gör man med satsen `break`. Det finns också en `continue`-sats som skippar resten av det pågående varvet och går direkt till villkoret igen.

Ett villkor som alltid är sant, till exempel `while 17>0`, skulle ge en oändlig slinga om inte `break` fanns, men det är en vanlig programmerings stil.

```
while 17>0:
    gissat = input("Gissa ett heltal:")
    if gissat==rätt: break
    if gissat<rätt:  print "För liten gissning!"
    else:  print "För stor gissning!"
```

Komplicerade villkor kan byggas upp med orden `and`, `or` och `not`.

```
if x>0 and y>0 : - - -
if x==17 or x==666 : - - -
while not 0<=x<=1 : - - -
```

1.5 Listor och `for`-slingor

En textsträng är en *lista* av tecken. I andra språk säger man vektor eller array, men pythonbegreppet är lista. Man kan göra listor av tal, listor av textsträngar och listor av listor och alla indexeras, klipps och klistras på samma sätt som textsträngar gör.

```
v = [17, 666]
v = v + [4711]
print v, len(v)              =>  [17, 666, 4711] 3
```

```
print v[0], v[-1]            =>  17 4711
print v[3]                   =>  IndexError: list index out of range
opera = ["Flex","och","Plex"]
print opera[2][0]            =>  P
knas = [v, opera]
print knas[1][2][1:3]        =>  le
```

Ska man gå igenom alla element i en lista är `for`-slingor oftast bäst.

```
for y in v: print y+1,        =>  18 667 4712
for z in opera: print z[0],   =>  F o P
for i in [0,1,2]: v[i]=v[i]-1
print v                       =>  [16, 665, 4710]
sum = 0
for i in [0,1,2]: sum=sum+v[i]
print sum                     =>  5391
for j in [0,1,2]:
    for i in [0,1]:
        print knas[i][j],     => 17 Flex 666 och 4711 Plex
```

Som synes behöver man ofta listor av typen `[0,1,2]`. Hur skapar man enklast en sådan lista med talen 0 till 99? Här är ett förslag som utgår från en tom lista.

```
v = []
i = 0
while i<100:
    v = v + [i]
    i = i + 1
print v            =>  [0, 1, 2, 3, 4, 5, 6, ..., 99]
```

Listigt, ja, men helt överflödigt eftersom en sådan lista skapas av `range(100)`. Vill man börja på något annat tal än noll går det bra att skriva till exempel `range(17,100)`.

Ordet `in` är reserverat och får inte användas som variabelnamn. Utom i `for`-slingor kan det användas när man vill kolla om ett element finns med i en lista. Både `if 17 in v:` och `if 17 not in v:` fungerar. Om man också vill veta var i listan elementet finns ger `v.index(17)` index för första förekomsten.

Listor av typen `v = [17, 666, 4711]` skiljer sej från textsträngar i ett viktigt avseende – dom kan ändra sitt innehåll. Ett exempel:

```
namn = "Gun"
namn[2] = "d"       =>  TypeError: object doesn't support item assignment
v = [17, 666, 4711]
v[2] = 42
print v             =>  [17, 666, 42]
v.reverse()
print v             =>  [42, 666, 17]
v.sort()
print v             =>  [17, 42, 666]
```

För textsträngar finns inte vare sej `reverse` eller `sort`, men `ord = texten.split()` skapar en lista av textens ord och `rader = texten.splitlines()` en lista av textens rader.

Den som av någon anledning föredrar listor som inte kan ändra innehåll skriver runda parenteser i stället för kantiga. Pythonterminologin kallar detta för en *tupel* och det är allt man behöver veta om tupler.

Vanliga listor indexeras med $0, 1, 2, \ldots$ och det är inte alltid vad man vill göra. Ett `dictionary` (uppslagsbok) kan indexeras med vad som helst, men oftast med textsträngar. Man utgår från ett tomt dictionary och lägger till uppslagsord och deras värden.

```
eng = {}
eng["vår"] = "spring"
eng["sommar"] = "summer"
eng["höst"] = "autumn"
eng["vinter"] = "winter"
for x in eng:
    print "Vad heter", x, "på engelska?"
    svar = raw_input()
    if svar==eng[x]: print "Rätt"
    else: print "Fel,", eng[x]
```

Man tar bort något ur databasen med till exempel `del eng["vinter"]`.

1.6 Funktioner och moduler

Python har fyra taltyper:

int Heltal, högst tiosiffriga.
long Heltal, hur många siffror som helst.
float Reella tal, sexton siffrors precision.
complex Komplexa tal, sexton siffrors precision i real- och imaginärdel.

Typnamnen kan användas för konvertering av textsträng till tal. Komplexa tal skrivs som `2+3j` där `j` är imaginära enheten.
Förutom plus, minus, gånger och delat med finns operationerna

`abs(x)`	absolutbeloppet av x
`x**y`	x upphöjt till y
`x // y`	heltalsdelen av x/y
`x % y`	resten vid divisionen.

Dom båda sista operationerna bör bara användas för positiva tal.

```
print 2e9                       =>  2000000000.0  (obs decimalpunkt!)
print 2*10**9                   =>  2000000000
print 2*10**10                  =>  20000000000L  (L betyder long)
a="3+4j"
z=complex(a)
print abs(z)                    =>  5.0
365 // 7                        =>  52  (veckor på ett år)
365 % 7                         =>  1   (och en dag över)
```

Nuvarande pythonversion tolkar `x/y` som `x//y` om `x` och `y` är heltal, alltså blir `7/2` inte `3.5` utan `3`. För att få vanlig division kan man skriva `float(x)/y` och `7.0/2`. Slutligen kan alla tilldelningssatser av typen `x=x+17` skrivas kortare som `x+=17` och det finns också `*=` osv.

Det här är vad som ingår i själva pythonspråket. För övriga matematiska funktioner måste modulen `math` importeras.

```
from math import *              (importerar allting, lite onödigt)
print sin(pi/2), log(e)   =>  1.0 1.0
print sqrt(2)             =>  1.4142135623730951
print 4*arctan(1)         =>  NameError: name 'arctan' is not defined
print 4*atan(1)           =>  3.1415926535897931
```

För att inte råka importera namn som man redan använder bör man i stället för stjärnan räkna upp vad man vill importera. Tydligast är att bara skriva `import math` och sedan skriva till exempel `math.sin(math.pi/2)`.

Man kan definiera egna funktioner och man kan importera egna moduler. Så här definieras en funktion.

```
def g(x):           En funktion av en variabel definieras...
    y=x*x+1         ...en lokal hjälpvariabel y används...
    return y*y      ...funktionsvärdet returneras...
                    ...och längre ner i programmet görs anropet.
print g(0),g(1)     => 1 4
```

Funktionen kan som följande exempel visar returnera flera värden eller inget värde alls.

```
from math import pi, sin, cos, sqrt
def xycoordinates(r, phi):
    x = r*cos(phi)
    y = r*sin(phi)
    return x,y                          Två värden returneras...
a,b = xycoordinates(sqrt(2), pi/4)     ...och tas emot.

def header(title):
    n = (60 - len(title))//2
    stars = "*"*n
    print stars, title, stars           Ingenting returneras.
header("Pythonkramaren")
header("Kapitel 1")
```

Globala variabler som används i funktionen bör deklareras med en sats av typen `global x`. Utan globaldeklaration är fortfarande `x`-värdet åtkomligt, men försök till tilldelning `x=...` gör `x` lokal. Ett lurigt exempel:

```
def zeroall():
    x = 0                         Gör x till lokal variabel
    v[0] = 0
    v[1] = 0
    print "Efter:", x, v

x = 17
v = [666, 4711]
print "Före:", x, v           =>  Före: 17 [666, 4711]
zeroall()                     =>  Efter: 0 [0, 0]
print "Till slut:", x, v      =>  Till slut: 17 [0, 0]
```

För att nollställa det globala `x` måste man infoga satsen `global x` i funktionen. För att inte nollställa det globala `v` hade satsen `v = [0,0]` dugt!

Det finns hundratals moduler för grafik, numerik, internetkoppling, systemanrop osv. Och varje hemgjord pythonfil kan också importeras. Tänk bara på att exekverbara satser i filen utförs vid importen!

1.7 Klasser och objekt

Objekt och metoder har vi redan sett exempel på. Klassen `str` har bland annat metoden `upper` och klassen `list` har metoden `reverse`. Egna klasser skapar man så här:

```
class Monty:
    name = "Python"              Startvärden kan anges

john = Monty()                   Ett objekt av klassen skapas
print john.name             =>   Python
john.name = "Cleese"
john.born = 1939                 Nya attribut kan läggas till
print john.name, john.born  =>   Cleese 1939
```

Anta att vi vill definiera en metod `write` i klassen Monty som kan anropas `john.write()`. Om inte punktnotationen fanns skulle det anropet ha sett ut som `write(john)`, och det är så metoden ska programmeras!

```
class Monty:
    name = "Python"
    def write(self):
        print self.name, self.born
```

Javaprogrammerare kanske föredrar `this` men i pythonkretsar är `self` det vanligaste namnet på den osynliga första parametern i metodanrop.

```
class Monty:
    name = "Python"
    def setname(self,name):
        self.name=name
    def write(self):
        print self.name, self.born
```

Konstruktorn för `john = Monty("Cleese",1939)` måste heta `__init__` och kommer att se ut så här:

```
class Monty:
    name = "Python"
    def __init__(self, name, born):
        self.name = name
        self.born = born
```

```
    def setname(self, name):
        self.name = name
    def write(self):
        print self.name, self.born
```

Om man vill ha den parameterlösa konstruktorn kvar kan man ge standardvärden, alltså `def __init__(self, name="Python", born=1969):`

En underklass till Monty skapas så här enkelt:

```
class PythonPart(Monty):
    part = "unspecified"
    def setpart(self, part):
        self.part = part
    def write(self):
        print self.name, self.born, self.part
john = PythonPart("Cleese", 1939)
john.write()                          => Cleese 1939 unspecified
john.setpart("parrot owner")
```

1.8 Körning och avlusning

För att datorn ska hitta pythontolken kan man behöva ange sökvägen dit. Ett av följande kommandon bör läggas in i någon uppstartsfil.

```
module add python           => Unix
PATH = %PATH%;C:\Python23   => Windows
```

Om datorn inte hittar pythonmoduler man försöker importera får man fixa till sökvägen `PYTHONPATH` på samma sätt.

Vare sej man använder den grafiska pythonmiljön IDLE eller skriver i Emacs går körning och avlusning till på samma sätt: Man skriver kod i ett fönster och kör den i ett annat fönster, där pythontolken är igång. Avlusade program kör man direkt med `python hej.py`. Vid den första lyckade körningen skapas en kompilerad fil `hej.pyc` som används vid framtida körningar. Kommande körningar går därför fortare än den första.

Ibland blir det fel, men i Python finns botemedel. Ett *särfall* kan tas om hand inne i programmet av `try - except`.

```
while 0<17:
    try:
        x = input("Gissa mitt tal:")
        break
```

```
    except:                             =>  Hopp hit vid inläsningsfel
        print "Inget tal, försök igen!"
```

Ett fel i try-avsnittet ger ett hopp till except-avsnittet. Man kan själv med `raise` skapa ett särfall för att hoppa till `except`:

```
while 0<17:
    try:
        x = input("Gissa mitt tal:")
        if x==4711: raise
    except:                             =>  Hopp hit vid rätt gissning...
        print "Rätt!"                   => ...men även vid inläsningsfel
        break
```

Exemplet visar att man kan behöva skilja mellan olika särfall, så här:

```
while 0<17:
    try:
        x = input("Gissa mitt tal:")
        if x==4711: raise Exception
    except StandardError: print "Inget tal, försök igen!"
    except Exception:
        print "Rätt!"
        break
```

StandardError är språkets inbyggda särfall, inga andra. Exception räknar in även dom hemgjorda särfallen.

Ett bra sätt att avlusa redan vid kodningen är att infoga *assert*-satser här och där.

```
x=input("Din ålder:")
assert 2<x<110          => Felavbrott om villkoret inte uppfylls
```

Vill man inte ha felavbrott kan man ha `try - except AssertionError`.

Pythontolken är bra vid avlusning; man kan ju kolla en variabels värde genom att skriva dess namn. I IDLE finns dessutom en debugmeny där man kan stega sej igenom programmet.

Vi avslutar kapitlet med en förteckning över dom lurigaste pythonfelen.

- **Kolon** brukar man glömma i `if, while, for` osv.
- **self** glömmer man alltid i objektmetoder.

- `a = b` betyder inte att `b`-värdet kopieras utan att `a` ska peka på det värde som `b` redan pekar på. Om det är en lista eller ett objekt kan effekten bli oväntad. Då kommer `b[1] = 17` att ändra även på `a[1]` och `b.tal = 17` att ändra även på `a.tal`.

- **Globala variabler** blir lokala om dom tilldelas ett värde i metoden utan att ha globaldeklarerats.

```
he ="lumberjack"
def sing():
    print he      => local variable 'he' referenced before assignment
    he = "OK"        ...men utan denna sats hade utskriften fungerat!
```

- **Klassvariabler** kan användas som startvärden för objektvariabler men om dom är listor är det riskabelt, som följande exempel visar.

```
class Brian:
    life = ["jolly","rotten"]        =>  Listan är klassvariabel...
    def whistle(self, bright, side):
        self.life[0] = bright        => ...som alla objekt ändrar i
        self.life[1] = side
```

Det hade blivit rätt med `self.life = [bright,side]`, för när variabeln `self.life` tilldelas ett värde uppstår en objektvariabel. Innan dess är det klassvariabeln som används.

2 Räkna, skriva, läsa – `print` och `input`

I det här kapitlet förutsätts ingen programmerarerfarenhet. Efter att ha läst det kommer du att kunna skriva pythonprogram av följande slag.

```
---PYTHONS NYBURGARE---
Önskat antal burgare: 4
Önskat antal pommesfrites: 3
Önskat antal cola: 5
Det blir 167 kronor, tack!
```

2.1 Räkna med python

Python är ett språk som är lätt att lära både människor och datorer. Det finns bara trettio pythonord (se omslagets insida) vartill kommer namn, som man själv hittar på. Det datorprogram som förstår pythonspråket kallas *pythontolken* och startas med kommandot `python` eller med klickning på lämplig ikon. Eftersom pythontolken förstår matematiska tecken kan man använda den som kalkylator.

```
Python 2.3.2
Type help, copyright, credits or license for more information.
>>> 1+2                         Skriv ett matematiskt uttryck...
3                               ...så räknar python ut det.
>>> 1+2*3                       Stjärna betyder multiplikation.
7
>>> r=50                        Egna variabelnamn får användas.
>>> 2*r                         Så beräknas diametern.
100
>>> pi=3.14                     Decimalkomma skrivs som en punkt.
>>> 2*pi*r                      Så beräknas omkretsen.
314.0
>>> T=37.5                      Lite feber i celsiusgrader.
>>> F=32+9*T/5                  Omräkning till fahrenheitgrader.
>>> F                           Vad blev det?
99.5                            Lite fahrenheitfeber.
>>> pi/6                        Division som inte går jämnt ut...
0.52333333333333332             ...ger sexton riktiga siffror.
>>> pi/pi                       Borde förstås bli 1, men svaret...
1.0                             ...skrivs med en decimal. Varför?
>>> 10**2                       Dubbelstjärna är upphöjt till.
```

```
100
>>> 2**10                       Denna konstant kallas 1K.
1024                            Mer om det senare!
```

Plus, minus, stjärna, snedstreck och dubbelstjärna är *räkneoperatorer*. Vi återkommer snart till några mera spännande operatorer.

2.2 Skriva program

Den som startar pythontolken med `python` kan sedan ge kommandon som utförs direkt. Vanligare är att man först skriver kommandona i en fil och sedan ger den till pythontolken. Det är det som kallas programmering och körning och det kan se ut så här.

```
> python heartbeat.py
Antal hjärtslag är cirka             |
1 per sekund                         |
60 per minut                         |
3600 per timme                    Körning
86400 per dygn                       |
31536000 per år                      |
2522880000 per liv                   |
```

Programfilen ser ut så här.

```
print "Antal hjärtslag är cirka"
s = 1
print s, "per sekund"
m = 60*s
print m, "per minut"
h = 60*m
print h, "per timme"
d = 24*h
print d, "per dygn"
y = 365*d
print y, "per år"
l = 80*y
print l, "per liv"
```

Filnamnet är `heartbeat.py` (alla pythonprogram bör ha efternamnet `.py`) och filen kan skrivas i valfritt skrivprogram. Det bästa heter Emacs, men till och med MS Word kan användas om man sparar filen som ren text.

Programrader kallas för *satser* och i hjärtslagsprogrammet finns bara två sorter. *Printsatser* kommenderar python att skriva något på skärmen, antingen tal eller text, och text måste skrivas mellan citattecken. *Tilldelningssatser* tilldelar ett variabelnamn ett värde som räknas ut. Man får göra uträkningar inne i printsatsen också, till exempel `print 60*60*24,"per dygn"` men tilldelningssatsen gör det tydligare.

2.3 Läsa input

Läsning kallas det när programmet får input från användaren, så här.

```
Hur många år är du?
23                                Användarens input läses...
725328000 hjärtslag hittills      ...och används av python.
```

Så här ser satserna ut.

```
print "Hur många år är du?"
age = input()
print age*y, "hjärtslag hittills"
```

När python stöter på inputsatsen blir det paus så att användaren ska få skriva in ett värde och trycka retur (eller enter). Först då läser python det tal som skrivits och tilldelar variabeln `age` detta värde. Det är viktigt att programmet skriver ut en fråga före inputsatsen. Här görs det med en printsats, men det går lika bra att stoppa in frågan mellan inputparenteserna:

```
age = input("Hur många år är du? ")
```

2.4 Mera matte

Likhetstecknet gör att en tilldelningssats ser ut som en ekvation, men det är den inte. En av dom vanligaste tilldelningssatserna är `x = x+1`; en avskyvärd lögn för en matematiker men för en datalog betyder det bara att x-värdet ökas med 1. Det finns ett förkortat skrivsätt för det, för att känsliga matematiker inte ska få hjärtslag: `x += 1` (utläses "x ökas med ett") och motsvarande för övriga operatorer. Vårt hjärtslagsprogram hade kunnat skrivas kortare (och obegripligare) så här.

```
print "Antal hjärtslag är cirka"
x = 1
print x, "per sekund"
x *= 60
```

```
print x, "per minut"
x *= 60
print x, "per timme"
x *= 24
print x, "per dygn"
x *= 365
print x, "per år"
x *= 80
print x, "per liv"
```

Python gör skillnad på heltal och decimaltal vid utskrift. Resultatet av en operation med decimaltal anses alltid vara ett decimaltal, även om det går jämnt ut. Värdet av `0.5+0.5` är alltså inte `1` utan `1.0` och värdet av `17/1.0` är inte `17` utan `17.0`. För att bli av med decimalerna kan man använda funktionen `int`. Varför heter den så?

```
>>> int(17.0)              Int ska de va decimaler, int...
17
>>> int(-3.14)
-3
```

Divisionssnedstreck bör inte användas vid heltalsräkning. Men det finns två andra mycket användbara operationer: *heltalsdivision* och *modulo*.

```
>>> 17 // 5                  Heltalsdivision ger heltalssvar.
3
>>> 17 % 5                   Modulo ger resten.
2
>>> 365 // 7                 Antal hela veckor på ett år.
52
>>> 365 % 7                  Antal överskjutande dagar.
1
>>> pengar = 100             Du har en hundring att köpa glass för.
>>> pinne = 7                Varje glasspinne kostar sju kronor.
>>> pengar // pinne          Hur många pinnar räcker pengarna till?
14
>>> pengar % pinne           Och hur många kronor får du över?
2
```

Varning för vanligt divisionsstreck mellan heltal! Nuvarande pythonversion tolkar det som dubbelstreck, alltså heltalsdivision. Som exemplen visar räcker det att sätta en decimal på något av talen.

```
>>> 1/8                          Snedstreck mellan heltal ger fel.
0
>>> 1.0/8                        Om det finns decimal blir det rätt.
0.125
>>> 1/9.0                        Oändliga bråk räknas ut med sexton siffror.
0.1111111111111111
>>> 100.0/10*10                  En division följd av en multiplikation.
100
>>> 100.0/(10*10)                En multiplikation följd av en division.
1
```

Uttryck som `17+3*x**2/y` tolkar python rätt, alltså som $17 + 3\,x^2/y$, och tvetydiga uttryck som `a/b/c` räknas ut från vänster till höger, alltså som $(a/b)/c$.

2.5 Hemma hos datorn: Binära tal

Talet 17 matar man in genom att trycka på tangenten 1 och tangenten 7. Om tangenterna finns på en telefon eller en räknedosa lagras då en etta och en sjua, men en dator lagrar i stället `10001`. Datorn räknar nämligen inte decimalt (med siffrorna noll till nio) utan *binärt* (tvåvärt), och datorns minne består av *bitar* som kan ha värdet noll eller ett. Det kan vara magnetpunkter på skivminnen, elektriska pulser i kretskort eller ljuspulser i optiska fibrer, det viktiga är att man kan representera nollor och ettor.

En följd av nollor och ettor kan tolkas som ett *binärt tal*. Om man skriver upp binära och decimala tal i var sin kolumn kan man översätta, *konvertera*, mellan talsystemen. Man ser att dom binära talen $1, 10, 100, 1000, 10000$ betyder tvåpotenserna $1, 2, 4, 8, 16$ och det gör det enkelt att till exempel konvertera det binära talet 10001. Den första ettan betyder 16 och den sista 1, alltså $16 + 1 = 17$. Det är bra att kunna tvåpotenserna utantill, åtminstone upp till 1024.

1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
10000	16

Dom flesta datorer har minnesceller som rymmer trettiotvå bitar. Hur stort binärt tal får då plats i en minnescell? Vet man att $2^{10} = 1024$ (datalogins viktigaste konstant) så förstår man att tio bitar räcker till talet tusen, tjugo bitar till talet en miljon och trettio bitar till talet en miljard. Eftersom en bit, teckenbiten, används som plus eller minus, (negativa tal ska ju också få finnas) kan en trettiotvåbitars minnescell räkna till cirka plusminus två

miljarder. När större tal dyker upp går Python automatiskt över till att använda två eller flera minnesceller. Antalet hjärtslag per liv kan Python därför ange till två och en halv miljard. Samma program i Java eller C skulle ha angett antalet hjärtslag till -1772087296. Kan du gissa varifrån minustecknet kommer?

Så lagras alltså heltal binärt, men hur lagras decimaler, som i 3.14 ? Om man får använda tiopotenser kan man ju bli av med decimalerna genom att skriva $314 \cdot 10^{-2}$ och lagra dom båda heltalen 314 och -2 i stället. Just så gör datorn, men med tvåpotenser i stället för tiopotenser. Sådana tal kallas *flyttal* eftersom decimalpunkten i det första talet flyter iväg dit det andra talet (exponenten) anger.

2.6 Varningar och råd

Decimalpunkt används i program, aldrig decimalkomma.

Variabelnamn skrivs oftast med små bokstäver men stora bokstäver, siffror och understreck tillåts också. Våra utmärkta svenska bokstäverna åääö är tills vidare utmobbade.

Tomrader i programmet gör ingen skada, inte mellanrum inne i satserna heller, men börja inte en rad med några blanksteg! Vad det betyder kommer i nästa kapitel.

Printsatser kan avslutas med kommatecken om man inte vill ha ny rad efter utskriften.

2.7 Övningar

21. Skriv nyburgarprogrammet som inleder kapitlet!

22. Skriv programmet `aha.py` som uppför sej enligt körexemplet.

```
Vilket år är du född? 1984
Aha, 20 år i år!
Bara 45 år till pensionen...
```

23. Skriv ett program som frågar efter två heltal och skriver ut summan, differensen, produkten och kvoten!

24. Skriv programmet `seconds.py` som frågar efter ett antal sekunder och skriver ut hur många dagar, timmar, minuter och sekunder det är!

25. Skriv programmet `siffersumma.py` som frågar efter ett tresiffrigt tal och skriver ut dess siffersumma. Om talet exempelvis är 516 blir siffersumman 12.

26. Skriv programmet `fahrenheit.py` som uppför sej enligt körexemplet. Algoritmen är $F = 32 + C * 1.8$.

```
Temperatur (grader Celsius): 37.5
Det är 99.5 grader Fahrenheit.
```

27. Skriv programmet `rabatt.py` enligt körexemplet. Rabatten är tio procent och räknas i hela kronor.

```
Pris (kr): 231
Avdrages 23 kr
Nettopris 208 kr
```

28. Skriv programmet `arvode.py` enligt exemplet. Timlönen är 97 kr och lönen utbetalas i hela kronor.

```
Började jobba: 08.30
Slutade jobba: 11.15
Arvodet blir 276 kr.
```

29. I programmet finns två tal, `x` och `y`. Vad gör satserna här intill? Pröva med några insatta värden!

```
x = x+y
y = x-y
x = x-y
```

3 Funktioner och procedurer – `def` och `return`.

Pythonspråket har bara trettio ord – `print`, `input` och några till – men man kan också definiera egna ord. En hammarbysupporter vill kanske definiera ordet `bajen` så att det skriver ut en hejaramsa. Så här går det till.

```
def bajen():
    print "Heja Bajen, friskt humör,"
    print "det är det som susen gör!"
```

Om den här definitionen står i början av programmet behöver supportern sedan bara skriva satsen `bajen()` när hon vill att ramsan ska komma ut på skärmen.

Det som supportern har definierat kan kallas ett underprogram eller en *procedur*. Ännu vanligare är att man vill definiera en egen *funktion*. Den som ofta behöver räknaa om temperaturer från Celsius till Fahrenheit vore tacksam om det funnes en funktion som kunde anropas `fahrenheit(37.5)` och då gav funktionsvärdet 99.5. Och en sådan kan man lätt definiera själv.

```
def fahrenheit(c):
    f = 32 + c*1.8
    return f
```

Om den här definitionen står i början av programmet kan man sedan programmera som om `fahrenheit` ingick i pythonspråket.

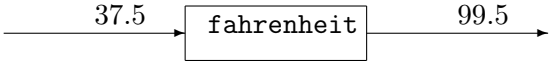
```
print "Temperatur (grader C): ",
x = input()
y = fahrenheit(x)
print "Alltså", y, "grader F"
```

3.1 Anrop

Ett anrop av en procedur eller funktion ska alltid ha parenteser. Mellan parenteserna kan det stå ett eller flera *parametervärden*, men det kan också vara tomt. Några exempel:

```
normal = fahrenheit(37.0)        <= Funktionsanrop med en parameter.
x = input()                      <= Funktionsanrop utan parameter.
ritarektangel(60,25)             <= Proceduranrop med två parametrar.
bajen()                          <= Proceduranrop utan parameter.
```

När python utför ett anrop måste funktionen redan vara definierad. Det första som händer är att funktionens parametrar får sina värden och sedan fortsätter körningen med satserna i definitionen. När ett funktionsvärde beräknats returneras det till anropet och den avbrutna programkörningen kan fortsätta. En bra bild av vad som händer är den här.



Anropet kan vara en liten del av ett komplicerat uttryck, till exempel
`print fahrenheit(37.5)-fahrenheit(37.0),"grader F över det normala"`

Några matematiska funktioner finns inbyggda.

```
abs(-17)              Absolutbelopp, alltså 17
int(3.14)             Heltalsdel, alltså 3
float(4711)           Samma tal som flyttal, alltså 4711.0
```

Andra matematiska funktioner måste man importera först. Deras definitioner finns i modulen `math`.

```
from math import *    Importerar allting i math-modulen
sqrt(2)               --> 1.4142135623730951
cos(0)                --> 1.0
sin(pi/2)             --> 1.0
exp(1)                --> 2.7182818284590455 (skrivs även e)
log(e)                --> 1.0
sin(pi/6)             --> 0.49999999999999994 (nästan rätt!)
```

Slumpfunktioner finns i modulen `random`. Dom två viktigaste ser ut så här:

```
from random import *
random()              --> ett slumptal mellan 0 och 1
randint(7,19)         --> ett heltal 7<=x<=19
```

Proceduranrop returnerar inget funktionsvärde, men kan ändå ha parametrar. Parametervärden är ofta tal men kan också vara text.

```
bajen()               => Heja Bajen, friskt humör,
                         det är det som susen gör!
writestars(17)        => *****************
heja("Djurgårn"}      => Heja Djurgårn, friskt humör,
                         det är det som susen gör!
```

3.2 Definitioner

En definition av en funktion måste göras före första anropet av den, men satserna inne i definitionen exekveras inte förrän vid anropet.

Efter pythonordet `def` ska man ange ett bra namn på sin funktion eller procedur följt av ett parentespar och ett kolon. Inom parentesen får det stå ett parameternamn eller rentav flera namnåtskilda av komma. Här kommer en mer användbar version av proceduren `bajen`.

```
def heja(laget):
    print "Heja", laget, ", friskt humör,"
    print "det är det som susen gör!"
```

Anropet `heja("Djurgårn")` skickar med texten `Djurgårn` till proceduren, som tilldelar texten variabelnamnet `laget`. Första printsatsen skriver först ut texten `Heja`, sedan det värde som variabeln `laget` har, alltså `Djurgårn`, och sist texten `, friskt humör,`.

Vilka satser som ingår i definitionen framgår av indraget, alltså att raderna börjar med ett antal blanktecken. Om man skriver i Emacs eller IDLE får man automatiskt indrag efter ett kolon. När man vill bli av med indraget suddar man sej åt vänster igen.

En funktion räknar ut ett värde och returnerar det. Man kan definiera en matematisk funktion som tar kvadraten på ett tal så här enkelt.

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

Och nu kan man använda funktionen `square` för att definiera nya funktioner. Så här räknar Pytagoras ut hypotenusan i en rätvinklig triangel med kateterna a och b.

```
from math import sqrt
def hypotenusa(a,b):
    c = sqrt(square(a)+square(b))
    return c
```

För att testa funktionen gör vi några anrop och skriver ut resultaten.

```
print hypotenusa(3,4)            =>  5.0
print hypotenusa(0,7)            =>  7.0
print hypotenusa(5,12)           => 13.0
```

3.3 Programutveckling enligt schampometoden

Programutveckling är en konst som kräver mycket mer än att man behärskar sitt programspråk. Vägen från problem till program är kantad av otaliga frågor. Hur ska man veta vilka procedurer och funktioner som man bör definiera? I vilken ordning ska man skriva programmet? Ska man tänka och rita först eller ska man skriva pythonkod direkt? Som grön programmerare önskar man att det funnes någon allmän metod att hålla sej till. Det gör det – den heter SCHAMPO!

Det är ett minnesord med fem ljud som ska påminna om dom fem stegen.

SCHAMPOMETODEN

1 Skärmen Rita hur den ser ut efter en lyckad körning. Dä ser man all in- och utmatning som gjorts.

2 Algoritmen Skriv hur man skulle lösa problemet utan dator. I enkla fall är algoritmen en formel.

3 Minnet Skriv upp variabelnamn och vilka värden dom tildelas i ditt körexempel.

4 Procedurer Varje deluppgift får en egen procedur eller funktion. Se dom som medarbetare, specialiserade på var sin uppgift.

5 Oppifrån Skriv nu programsatserna oppifrån och ner.

Vi tillämpar nu schampometoden på följande problem.
Skriv ett program som beräknar hur många flaskor ormschampo som behövs för att tvätta en pyton. Varje flaska räcker till exakt 1m^2 *ormskinn.*

1. Skärmens utseende skulle kunna vara så här. I körexemplet kan man hitta på vilka ormdimensioner som helst

```
Ormens längd (m): 5.19
Ormens diameter (m): 0.17
Det går åt 4 schampoflaskor.
```

2. Algoritmen är att först beräkna ormskinnets area och sedan höja värdet till ett heltal flaskor. Skinnarean är längden gånger bredden, $a = l \cdot b$, där bredden är π gånger diametern, $b = \pi \cdot d$.

3. Variabelnamnen tar vi ur algoritmen:

längd	l	=	5.19
diameter	d	=	0.17
bredd	b	=	0.58016
area	a	=	3.126
flaskantal	f	=	4

4. Om man hade en medarbetare skulle man överlåta den tråkiga areaberäkningen till honom. Fick man ännu en medarbetare skulle hon få sköta flaskräkningen. Det behövs alltså två procedurer av funktionstyp: `area(l,d)` och `flaskor(a)`.

5. Nu kan man skriva programmet oppifrån och ner, först funktionsdefinitionerna, sedan in- och utmatning enligt skärmens utseende och med funktionsanrop på lämpliga ställen.

   ```
   def area(l,d):
       b = 3.14*d     # noggrannare pi behövs inte
       a = l*b
       return a
   def flaskor(a):
       f = int(a)     # så många hela flaskor går åt
       f = f+1        # och en flaska går delvis åt
       return f
   l = input("Ormlängd (m):")
   d = input("Ormdiameter (m):")
   a = area(l,d)
   f = flaskor(a)
   print "Det behövs", f, "flaskor ormschampo"
   ```

   Lägg märke till kommentarerna. Allt på raden efter nummertecknet # struntar Python i och det används för kommentarer till den som vill förstå programmet. Om man börjar raden med # blir hela raden kommentar.

3.4 Hemma hos datorn: Namn

Python har en *namnlista* (eng. name space) över alla namn den känner till. Från början finns bara pythonord som `print` och `return` där, men efter hand utökas namnlistan med variabler som införs och funktioner som definieras. Låt oss se på ett exempel.

```
x = input("Din längd (cm): ")
y = x*0.01
def idealvikt(langd):
    vikt = 23*langd*langd
    return vikt
print "Idealvikt:", idealvikt(y)
```

Dom första två satserna utökar namnlistan med `x` och `y`. Den tredje satsen lägger dit funktionsnamnet `idealvikt`, men inte parameternamnet `langd`. Fjärde och femte satsen hoppas över och sjätte satsen innehåller inget nytt namn, men väl ett funktionsanrop. Medarbetaren `idealvikt` anropas och `y`-värdet skickas med. Nu tar chefen paus medan medarbetaren jobbar. Hon utökar nu den *globala* namnlistan med en *lokal* namnlista. Först på den hamnar parameternamnet `langd` som tilldelas det överskickade värdet. Sedan kommer `vikt` på den lokala listan. Så småningom returnerar hon ett värde till den anropande chefen och samtidigt kastar hon sin lokala lista i papperskorgen. Varje gång hon anropas sker samma sak: lokala variabler uppstår, lever ett intensivt liv och dör sedan.

Lokala variabler uppstår på två sätt: antingen står dom som parameternamn eller också till vänster i en tilldelningssats. Variablerna på den globala namnlistan (`x` och `y` i exemplet) kan också användas inne i funktionen. Satsen `print x, y, langd, vikt` skulle alltså fungera strax före return-satsen.

Namnkrockar kallar man det för när samma namn står på både den globala och den lokala namnlistan. Regeln är då att den lokala variabeln avses om man inte uttryckligen skriver att det är den globala som gäller. Om namnet `langd` i exemplet byts ut mot `x` fungerar därför allt lika bra som förut, men den som läser programmet blir nog mycket förvirrad! Likaså kan namnet `vikt` bytas mot `y` utan att det spelar någon roll.

Ibland vill man inne i en funktionsdefinition tilldela en global variabel ett nytt värde. Låt oss säga att vi i exemplet stoppar in satsen `x = 0` eftersom vi vill nollställa det globala `x`. Men det som då händer är att det skapas en lokal variabel `x` som nollställs. Namn till vänster i tilldelningssatser uppfattas ju som lokala. För att ändra på globala `x` måste man skriva så här.

```
x = input("Din längd (cm): ")   # En global variabel...
y = x*0.01
def idealvikt(langd):
    global x                    # ...som inne i funktionen...
    x = 0                       # ...tilldelas nytt värde.
    vikt = 23*langd*langd
    return vikt
print "Idealvikt:", idealvikt(y)
```

Alla namn på listan har tilldelats ett *värde*, som i Python är ett *objekt*. Värdet kan vara någon sorts tal, en text. ett paket med personuppgifter eller en funktionsdefinition. En `def`-sats är i själva verket en tilldelning och borde kanske egentligen skrivas `idealvikt = def (langd): - - -`

3.5 Varningar och råd

Kommentar överst i programmet med datum, titel och upphovsman är en god vana. Kommentarer mitt inne i koden är sällan nödvändigt.

Rita variablernas minnesceller som rutor, så kan du provköra programexemplet på papperet, sats för sats!

Buggar eller löss (eng. bugs) kallar man programfel. Att avlusa programmet tar ofta längre tid än att skriva det. Med SCHAMPO slipper du löss!

Import är att utöka namnlistan med ett namn från en annan modul, som i `from math import sqrt`, eller alla namn från en annan modul, som i `from math import *`.

Flyttalsräkning är inte exakt utan har små avrundningsfel. Man är van vid att $8 * 0.1 = 0.8$, men eftersom 0.1 inte kan lagras exakt binärt får datorn $8 * 0.1$ till 0.80000000000000004.

3.6 Övningar

31. Skriv funktionen `celsius(f)` som översätter grader Fahrenheit till grader Celsius.

32. Skriv ett idealviktsprogram som använder algoritmen `idealvikt = längd - 105`

```
Din längd (cm): 168
Din idealvikt är 63 kg.
```

33. Skriv en funktion `area(a,b,c)` som beräknar arean för en triangel med sidorna `a,b,c`. Algoritmen heter Herons [1] formel och lyder så här: $T = \sqrt{p(p-a)(p-b)(p-c)}$ där $p = (a+b+c)/2$.

34. Skriv en funktion `seconds(d,h,m,s)` som beräknar hur många sekunder ett givet antal dagar, timmar, minuter och sekunder är.

35. Variabeln `n` har värdet 17 före anropet `minska(n)`. Vilket värdet har `n` efter anropet?

```
def minska(n):
    n = n-1
```

[1] Heron, grekisk vetenskapsman i Alexandria, född år 23

4 Grenar och slingor – `if` och `while`.

Varje sats har utförts exakt en gång i dom exempel vi sett. Men ofta vill man att vissa satser ska utföras många gånger och att vissa satser ibland ska hoppas över.

I följande exempel hoppar python först över ormbettsdefinitionen, utför sedan tilldelningarna och print-satserna och stöter så på en while-sats. Vad händer?

```
def ormbett():
    print "Ormen hugger..."
    print "Du har", n, "liv kvar"

n = 9          # Antal liv
skatt = 17     # Hål där skatten finns
print "Här finns tjugo ormhål."
print "I ett av dom ligger skatten gömd."

while n>0:
    x = input("Hand ner i hål nr ")
    if x==skatt:
        print "Du fick skatten!"
        break
    else:
        n = n-1
        ormbett()
```

Villkoret `n>0` kollas och eftersom 9 är större än 0 ska satserna i whileslingan utföras. När användaren valt ett tal kollas if-villkoret och om gissningen var rätt bryts slingan. Annars förbrukas ett liv och proceduren `ormbett` anropas. Python slingrar iväg till proceduren ett tag men är snart tillbaka och då är det dags för nästa varv i slingan.

Lägg märke till det som är gemensamt för `def`, `if` och `while`:

- Satsen avslutas med kolon.
- Följande satser är indragna.
- Det kan bli dubbelt indrag om konstruktionerna kombineras.

4.1 Grenar

Den enklaste användningen av `if` är när man vill utföra en sats bara i vissa fall. Om mamma är orolig för att lille Vahid ska gå för tunnklädd skriver hon kanske följande program.

```
temp = input("Utetemperatur: ")
if temp<5:
    print "Ta på dej halsduken!"
```

Den indragna print-satsen körs bara om villkoret är uppfyllt. Om mamma vill att programmet ska grena sej i två olika utskrifter krävs två satser till.

```
else:
    print "Då slipper du halsduken!"
```

Typiska villkor är olikheter: `x<y`, `x>y`, `x<=y`, `x>=y`, `x!=y` (skilt från) och likheter: `x==y`. Ja, det ska vara dubbla likhetstecken; man vill nämligen undvika sammanblandning med tilldelning.

I en if-sats delar sej programmet i två grenar. Men ibland vill man ha tre grenar, och det går också.

```
if age<18:
    print "Barnbiljett"
else:
    if age>=65:
        print "Pensionärsbiljett"
    else:
        print "Vuxenbiljett"
```

Det blir lite mer lättläst med pythonordet `elif` eftersom man då genast ser att det är tre alternativa grenar.

```
if age<18:
    print "Barnbiljett"
elif age>=65:
    print "Pensionärsbiljett"
else:
    print "Vuxenbiljett"
```

4.2 Slingor

Programmet `automat.py` säljer burkar för 17 kr med hjälp av en while-slinga (eng. loop). Den fungerar som en if-sats, men körs om och om igen så länge villkoret är uppfyllt. Om villkoret inte är uppfyllt ens första gången körs slingan noll gånger.

```
pris = 17
ilagt = 0
while ilagt<pris:
    mynt = input("Mynt:")
    ilagt += mynt
    print ilagt, "kr ilagt"
print "Varsågod, burken är din!"
```

Om det alltid är sant får man en oändlig slinga – en av dom vanligaste buggarna. Se på det här exemplet:

```
print "Ett fyrfaldigt leve!"
antal = 0
while antal<4:
    print "Hurra!"
```

Det blir ett evigt hurrande eftersom programmeraren glömt att öka `antal` med ett i slingan. Användaren får försöka bryta programmet med **ctrl-C**. Ibland skriver man avsiktligt en oändlig slinga genom att ange ett villkor som alltid är sant.

```
while 1<2:
    age = input("Ålder (avsluta med nolla):")
    if age==0:
        break
    print 65-age, "år till pensionen."
print "Tack för den här gången!"
```

Det här programmet använder pythonordet `break`, som bryter slingan på ett snyggt sätt. Man hade kunnat skriva `else:` före print-satsen, men det är faktiskt onödigt. Kan du komma på varför?

En vanlig användning av slingor är för att se till att användaren skriver in rimliga värden.

```
while 1<2:
    cm = input("Din längd i cm:")
    if 130<cm<230:
        break
    print "Orimligt, försök igen!"
```

Programmet kommer att envisas tills det fått ett rimligt värde.

4.3 Värstingvillkor

Det går att skriva hur komplicerade villkor som helst med hjälp av `and`, `or` och `not`.

```
if 0<x+y<17 and z!=4711: - - -
if namn=="Henrik" or namn=="Vahid": - - -
if not 0<=siffra<=9: - - -
if year%4==0 and year%100!=0: - - -  Villkor för skottår.
```

Men man ska inte skriva onödigt komplicerade villkor. Försök att undvika onödiga negationer av typen `if not x>0:`. Det är lättare att förstå `if x<=0:` som ju betyder samma sak. Och skriv framför allt inte dubbla negationer av typen `while not x!=17:`. Hur det kan förenklas får du själv tänka ut.

```
if 0<x+y<17 and z!=4711: - - -
if namn=="Henrik" or namn=="Vahid": - - -
if not 0<=siffra<=9: - - -
if year%4==0 and year%100!=0: - - -  Villkor för skottår.
```

Det går lika bra att ange villkor för texter som för tal.

```
while namn<"Eriksson": - - -          ...kommer före i bokstavsordning.
while bokstav in "aeiouyåäö": - - -   ...är en liten vokal.
```

4.4 Hemma hos datorn: Logik

Att datorn kan förstå logiska uttryck verkar märkligare än att den kan räkna ut matematiska uttryck. Det är det inte; logik kan nämligen göras om till enkel aritmetik. Det upptäckte den engelska artonhundratalsmatematikern George Boole och efter honom kallas tricket för *boolesk algebra*. Så här går det till.

Sanna uttryck ges sanningsvärdet 1 och falska uttryck sanningsvärdet 0. Exempelvis får $1 < 2$ värdet 1 och $1 > 2$ värdet 0. Orden `and` och `or` ersätts av `*` och `+`. Uttrycket `1<2 and 2<3` blir alltså $1 * 1$, dvs 1 och det betyder sant. Uttrycket `1>2 or 2<3` blir `0+1`, alltså 1 och sant.

Det finns mycket mer i boolesk algebra, men redan det här visar att man kan göra en del oväntade saker med villkorssatser.

```
while 1: - - -       Ger en oändlig loop.
while True: - - -    True betyder talet 1.
if n%2: - - -        Om n är udda...
ok = False           Booleska värden kan tilldelas...
if ok: - - -         ...och testas i villkorssatser.
```

4.5 Varningar och råd

Dubbla likhetstecken i villkor glömmer man alltid.

Bryta slingan kan man bara göra med `break`. Python kollar inte om while-villkoret plötsligt blir osant mitt inne i slingan.

Booleska funktioner kallar man funktioner som returnerar ett sanningsvärde: `True/False` eller om man så vill `1/0`. Mycket användbara i villkor av typen `if primtal(n): - - -`.

4.6 Övningar

41. Skriv funktionen `maximum(x,y)` som returnerar maximum av två tal. Testa sedan anropet `maximum("hej","hå")`.

42. Skriv ett gissningsprogram som fungerar enligt vidstående körning. Ett slumpat heltal mellan 1 och 100 ska gissas.

```
Gissa talet: 50
För stort!
Gissa talet: 17
För litet!
Gissa talet:
```

43. Skriv ett program som beräknar medelvärde av inmatade betygssiffror. En nolla avslutar inmatningen.

```
Betyg: 3
Betyg: 5
 - - -
Betyg: 0
Medelbetyget är 3.86
```

5 Textsträngar och tecken – `str` och `chr`.

Som följande exempel visar kan datorn behandla text lika väl som tal. Sådana program är extra roligt att skriva och att köra, men egentligen är det mest frågan om att klippa och klistra text.

```
Och se, din son skall heta...
Ja, vad heter du själv? Henrik
Och se, din son skall heta Henrikson
```

Vad är text och vad är tal? Är `sjutton` text eller tal? Är `420119-0818` text eller tal? För python är svaret enkelt: Allt i programmet som står inom citattecken är text. Alltså är `17` ett tal och `"17"` en text. För input från tangenterna gäler att `input()` används för tal och `raw_input()` för text. Efter satserna

```
x = input("Ett tal:")
y = raw_input("Ett tal:")
```

kan alltså `x` vara `17` och `y` vara `"17"`. Är `y` råare än `x`, kan man undra. Ja, för `x` har gjorts om till det binära talet `10001`, men `y` är fortfarande bara siffrorna `1` och `7`.

Texter kallas också *strängar*, en felöversättning av *string*, som här egentligen betyder pärlband – ett pärlband av bokstavstecken. Med `y = str(x)` och `x = int(y)` kan man göra om heltal till strängar och tvärtom. Ytterligare ett användbart anrop är `len(y)` som ger strängens längd, alltså 2 om `y="17"`.

5.1 Klippa och klistra text

För programexemplet som skapade sonnamn är algoritmen enkel: Klistra ihop namn+son. Så här blir pythonkoden.

```
print "Och se, din son ska heta..."
namn = raw_input("Ja, vad heter du själv? ")
sonnamn = namn+"son"
print "Och se, din son ska heta", sonnamn
```

För att klippa ut bitar ur en sträng anger man inom hakparentes önskat avsnitt. Första bokstaven i `namn` kallas `namn[0]`, andra `namn[1]` och så vidare. Om `namn` består av sex bokstäver blir sista bokstaven `namn[5]`. Med `namn[1:3]` menas avsnittet `namn[1]+namn[2]` och med `namn[3:]` menas avsnittet `namn[3]+namn[4]+namn[5]`. Om `namn="Henrik"`, vad skrivs då ut av följande sats?

```
print namn[1:3], namn[3:], namn[1:3]
```

Det är inte helt rätt att säga att man klipper ut en del av ordet; man kopierar delen. Pythonsträngar är oföränderliga! Ett praktiskt exempel på klippa och klistra följer nu.

```
pnr = raw_input("Personnummer: ")
if len(pnr)==11:
    pnr = pnr[0:6]+pnr[7:]
```

Så lätt är det att få bort bindestrecket i `420119-0818`, men kanske borde man kollat att det är ett bindestreck man klipper bort med satsen

```
if pnr[6]=="-": - - -
```

Ur personnumret kan man försöka få fram födelseåret.

```
year = pnr[0:2]      # Textsträngen "42"...
year = int(year)     # ...blir heltalet 42...
year += 1900         # ...som blir talet 1942
```

Om man skriver ut många tal med `print` kan resultatet se slarvigt ut, Man vill oftast få en rak högerkant genom att få in lagom många blanktecken först på raden. Om `n=1` kanske man försöker lägga på sju blanka genom hopklistringen `+n`, men det går inte att plussa ihop en text med ett tal! Man får göra om talet till en sträng, klistra ihop, se hur långt det blev och klippa bort onödiga blanka i början.

```
1
10
100
1000
10000
100000
1000000
10000000
```

```
n = "       "+str(n)
l = len(n)           # Vi vill ha längden 8...
print n[l-8:l]       # ...så överskottet kapas.
```

Man får tänka en del innan man insett att det fungerar. Eftersom man ofta vill räkna från slutet av en sträng tillåter python att man skriver `-8` i stället för `l-8`. På så sätt behöver man inte ta reda på längden utan kan skriva så här enkelt.

```
n = "       "+str(n)
print n[-8:l]
```

5.2 Textmokeri

Datorn skapades som räknemaskin men blev med tiden allt mer textmokare. Skrivprogram som Emacs och MS Word innehåller massor av avancerade funktioner, men pythonprogrammeraren klarar sej rätt bra med det vi gått

igenom här. En oväntad svårighet vid allt textmokeri är dock *radbytena*, där olika datorer använder olika tecken. Ett pythonprogram behöver ofta inte bry sej om radbytestecknet, eftersom `input` och `raw_input` läser en rad och sedan glufsar i sej radbytestecknet utan att skicka det vidare. På samma sätt skriver varje `print` ett radbytestecken utan att man ber om det.

Men ibland behöver textmokare syssla med radbytestecknet och då betecknas det `\n` i python.

Exempel:

```
print "Poeter\nära\norden"
```

```
Poeter
ära
orden
```

5.3 Hemma hos datorn: Textkodning

För datorn är radbyten tecken, precis som bokstäver och siffror. När en textfil skrivs ut på skärmen eller på papper gör radbytestecknen att utskriften delas upp i rader, och så långt verkar allt enkelt. Men problemet är att datorvärlden inte kommit överens om vilka tecken som ska användas för radbyte. I teckenstandarden finns det två olika tecken att välja på, tecken nummer 10 (radframmatning, eng. linefeed) och tecken nummer 13 (retur, eng. return), och unixvärlden använder det första medan macvärlden i stället använder det andra. Krångligast är pcvärlden, som har båda tecknen i varje radbyte.

Text måste liksom tal representeras binärt i datorn och man använder helt enkelt bokstävernas ordningsnummer i ett alfabet. Det vanligaste och primitivaste alfabetet är en amerikansk standard som kallas ASCII (uttalas aski). ASCII innehåller 128 tecken och numren får alltså plats i sju bitar. Alfabetet börjar med osynliga styrtecken, till exempel radframmatning och retur, och fortsätter med skiljetecken, siffror och andra ickebokstäver. På plats 65 kommer äntligen `A` osv fram till `Z` och på plats 97 kommer `a` osv fram till `z`. Det finns två funktionsanrop som översätter mellan tecken och ordningsnummer.

```
ord("{}A")              --> 65
chr(65)               --> "A"
chr(65+32)            --> "a"
ord("{}å)               --> 229
```

Hallå där, var kommer 229 ifrån? Jo, för att få med saknade europeiska bokstäver har man utökat ASCII med 128 tecken, däribland åääö, och en sådan standard brukar kallas Latin 1, som alltså är en åttabitskod. För att få in ryska, arabiska, kinesiska och tusen andra språk har man också standarden

Unicode, en sextonbitars kod med cirka 65000 tecken. Python hanterar Unicode hur lätt som helst, men det får du ta reda på själv om du är intresserad.

5.4 Varningar och råd

`v[0]` är det första tecknet i strängen `v`.

`v[3:7]` klipper efter tre tecken och efter sju tecken i strängen.

`v[2]="q"` går inte eftersom pythonsträngar är oföränderliga.

5.5 Övningar

51. Det första programexemplet har `print namn+"son"`. Hade det gått lika bra med `print namn, "son"`?

52. Skriv en funktion `versal` som med hjälp av `ord` och `chr` översätter små bokstäver till stora: Anropet `versal("e")` ska alltså returnera `"E"`.

6 Listor och index – `v[3]` och `for`.

En textsträng är en lista av skrivtecken, där varje tecken har ett index. Givetvis finns det också listor av tal och alla andra objekt. Man skriver listans element åtskilda av komman, alltihop inom hakparentes.

```
v = [17,666,4711]
w = [2004,17]
u = v+w              => u=[17,666,4711,2004,17]
print u[2:4]         => [4711,2004]
```

Precis som för textsträngar alltså, men det finns en viktig skillnad. En sats som `v[2] = 42` är nu tillåten. Namnen `v[0],v[1],v[2]` är alltså nu helt självständiga variabler som kan tilldelas nya värden. Och det gäller förstås också `w[0],w[1]` och `u[0],u[1],u[2],u[3],u[4]`.

En textsträng är däremot ett enda objekt och i den kan man inte ändra på en viss bokstav, om man nu skulle vilja det.

För en matematiker är det lättast att sej att index sitter nertill, så att variablerna heter v_0, v_1, v_2 osv, och då kan dom ses som *vektorkomponenter* i vektorn `v`.

Nan vill ofta göra input av flera värden på samma rad:
`"Dina betyg: "` **3 5 3 4 4**
Då läser man först hela raden och splittrar den sedan till en lista av tal.

```
rad = raw_input("Dina betyg: ")
tal = rad.split()
print tal             => ["3","5","3","4","4"]
```

Raden styckas i mellanslagen av `rad.split()`. Det konstiga skrivsättet med punkt efter `rad` kallas metodanrop och det återkommer vi till sist i detta kapitel. Programmet är inte helt jättebra om man ska beräkna betygsmedel, för nu har man fått en lista av textsträngar och inte en lista av tal. Men det är ju bara att konvertera till heltal med `int(tal[3])`.

6.1 Listgenomgång med `for`

Om ett visst värde finns i en viss lista testas med `in`.

```
x = input("Gissa ett heltal: ")
if x in v:
    print "Ett av vinsttalen!"
```

```
else:
    print "Inget vinsttal."
```

Men den vanligaste användningen av `in` är i for-satser.

```
v = [17,666,4711]
for x in v:
    print x,"är ett vinsttal"
```

Python går igenom listan `v` och `x` tilldelas i tur och ordning värdena `v[0],v[1],v[2]`. Efter varje sådan tilldelning görs satserna efter kolon. Vi kunde lika gärna ha skrivit

```
for pnyxtr in v:
    print pnyxtr,"är ett vinsttal"
```

Funktionen `len(v)` ger längden av listan `v`, men annars kunde man ha gjort så här.

```
l = 0
for x in v:
    l+=1
```

En textsträng kan visserligen inte ändras, men med for-satsen gör man lätt en förändra kopia. Så här stammar man fram en mexikansk bergstopp.

```
stamning = ""                       # En tom sträng
for tkn in "Popocatepetl":
    stamning += tkn*2
print stamning                      # PPooppooccaatteeppeettll
```

Lägg märke till att gångertecknet kan användas för att skapa en lång vektor där alla komponenter är likadana. En vektor med sjutton komponenter som alla har värdet 3.14 skapas enklast så här: `v = [3.14]*17`.

Båda användningarna av `in` kombineras ofta, bland annat när man vill viska namnet på sin älskade.

```
viskning = ""
for tkn in "Petronella":
    if not tkn in "aeiouyåäö":
        viskning += tkn
print viskning                      # Ptrnll
```

Knepigare är det att säga namnet baklänges, men följande fungerar faktiskt.

```
bakfram = ""
for tkn in "Petronella":
    bakfram = tkn+bakfram
print bakfram                       # allenorteP
```

Varför kan man inte använda det kortare skrivsättet `bakfram += tkn`?

6.2 Räkning med index

I alla sammanhang då man vill räkna något är det for-satser som används. Här är ett typfall.

```
vokaler = "aeiouyåääöAEIOUYÅÄÖ"
siffror = "0123456789"
v=0
s=0
rad = raw_input("Skriv en rad:")
for tkn in rad:
    if tkn in vokaler:
        v+=1
    if tkn in siffror:
        s+=1
print "Meningen hade",v,"vokaler och",s,"siffror."
```

Slingan går lika många varv som strängens eller listans längd. Ibland vet man antalet varv men har ingen lista av den längden. Då kan man använda konstruktionen `range(17)` som skapar en lista med längden sjutton, nämligen `[0,1,2,3,...,16]`.

```
for i in range(17):
    print "*"               => Sjutton stjärnor på höjden.
summa = 0
fakultet = 1
for n in range(1,17):       => Vi vill inte börja med noll
    summa+=n
    fakultet*=n
print summa,fakultet        => 136 20922789888000
```

Observera att vi här räknat fram $1 + 2 + \cdots + 16$ och $1 \cdot 2 \cdot \cdots \cdot 16$. Vi får aldrig med övre gränsen i `range(a,b)`.

Om man uppfattar listor som matematiska vektorer tycker man att `v+w` skulle betyda vektoraddition (som det faktiskt gör i Matlab). Men det betyder som sagt hopklistring. Vektoraddition får man programmera i en forslinga (vi antar att vektorerna har tre komponenter).

```
u = [0]*3                       => Tre komponenter som är noll
for i in range(3):
    u[i] = v[i]+w[i]            => Komponentvis addition
```

I sådana här fall är det alltid bäst att först tillverka en nollvektor med rätt antal komponenter, i stället för att klistra ihop den komponent för komponent. En anledning är att varje klistring skapar en ny kopia av alla gamla komponenter.

Exempel: Horners algoritm för att räkna ut polynomvärden: Om man vill räkna ut $2x^2 + 3x - 5$ då $x = 17$ är det onödigt arbetsamt att beräkna term för term. Horners effektiva algoritm sätter först $y = 0$ och upprepar sedan satsen $y = y * x+$koefficient för alla polynomkoefficienter.

```
y=0
y=y*x+2
y=y*x+3
y=y*x-5
```

I filen `poly.py` programmerar vi detta så här

```
def polyval(p,x):
    y=0
    for c in p:
        y=x*y+c
    return y
```

Anropet `polyval([2,3,5],17)` löser nu vårt ursprungliga problem.

Exempel: Procedur för tabellering av polynomvärden. Om vi inte nöjer oss med ett enstaka polynomvärde utan vill se en hel tabell kan en tabelleringsprocedur kännas angelägen. Så här vill vi kunna anropa den.

```
pol=[2,3,-5]
polytab(pol, 0, 5)            #Tabell för x = 0, 1, 2, 3, 4, 5
polytab(pol, 0, 1, 0.1)       #Tabell för x = 0.0, 0.1,..., 1.0
```

Det första anropet har bara tre parametrar, eftersom den sista parametern ska underförstås som 1 om den inte anges. Så här ordnar man det.

```
def polytab(p,start,slut,steg=1):    # OK om steg inte anges så 1
    x=start
    while x<=slut:
        print x,polyval(p,x)
        x=x+steg                     # Kan också skrivas x+=steg
```

6.3 Tabeller och matriser

En lista av textsträngar skapas så här: `song = ["Ja",må","hon",leva"]` och nu är hon `song[2]`. Hur betecknas bokstaven h? Tydligen med `song[2][0]`. Dubbla index är alltså tillåtet.

Multiplikationstabellen har tio rader med tio tal i varje rad. För att skriva ut den krävs en dubbel for-sats.

```
for i in range(1,11):
    for j in range(1,11):
        m = "   "+str(i*j)
        print m[-4:]
    print
```

```
   1    2    3    4    5  ..
   2    4    6    8   10  ..
   3    6    9   12   15  ..
   4    8   12   16   20  ..
  ..   ..   ..   ..   ..
```

För en rektangulär tabell A kan en komponent anges som A_{42} och då menar man att den står i rad 4 och kolumn 2. Python ser en tabell som en lista av listor och då skriver man `a[4][2]`. Rad fyra, dvs lista nummer fyra, betecknas `a[4]` och dess komponent nummer två blir alltså `a[4][2]`.

En matris är detsamma som en tabell och python uppfattar den alltid som en lista av listor. Här är tre olika sätt att tillverka en matris med formatet 3×3.

```
enhetsmatris = [[1,0,0],[0,1,0],[0,0,1]]

nollmatris = [None]*3          # Tre tomrader...
for i in range(3):
    nollmatris = [0,0,0]       # ...som ändras till nollistor

inputmatris = []               # Tommatris att fylla på
for i in range(3):
    rad=[]                     # Tomrad att fylla på
    for j in range(3):
        x = input()
        rad+=[x]               # Raden fylls på med ett tal
    inputmatris+=[rad]         # Matrisen fylls på med en rad
```

6.4 Sökning och sortering

I databassammanhang är sökning och sortering dom viktigaste operationerna. Smarta algoritmer ska beskrivas i senare kapitel, men med for-satsen kan man i alla fall direkt skriva fungerande, om än ineffektiva, metoder.

Vi tänker oss en databas med namn och telefonnummer för `n` personer och vi låter den vara en lista av `n` poster av typen `["Henrik","7908163"]`. Att söka efter en persons telefonnummer är nu enkelt.

```
person = raw_input("Person: ")
for post in databas:
    if post[0]==person:
        print "Tel:",post[1]
```

Det här är *linjär sökning* på dummaste sätt. Det dumma består i att sökningen fortsätter även efter det man hittat personen! Man bör förstås bryta slingan då, och man bör också se till att det blir en utskrift när sökningen misslyckas. Ett vanligt sätt är att använda en boolesk variabel.

```
found = False
person = raw_input("Person: ")
for post in databas:
    if post[0]==person:
        print "Tel:",post[1]
        found = True
        break
if not found:
    print "Finns inte i databasen."
```

Att hitta en person i telefonkatalogen vore hopplöst om inte namnen stod i bokstavsordning. Med sortering menar man att man ordnar en lista av `n` texter eller tal i bokstavsordning eller storleksordning. Den enda metod vi tar upp nu är *urvalssortering*, som visserligen är långsam och osmart, men som är en bra tillämpning på sökning.

Principen är att man söker reda på var det minsta värdet i listan finns och sedan byter man plats på detta värdet och det som står först. Sedan har man `n-1` element kvar att ordna på samma sätt.

Vid sökningen efter det minsta värdet letar man efter ett värde som är mindre än det minsta man redan har sett. Efter att ha noterat det nya minstvärdet (och var det fanns) stegar man sej vidare genom listan. För att ha något att starta med sätter man `v[0]` som provisoriskt minstvärde.

```
imin = 0
vmin = v[0]
for i in range(n):
    if v[i]<vmin:
        imin = i
        vmin = v[i]
```

Nu finns det minsta värdet `vmin` på index `imin` och man kan låta det byta plats med `v[0]`.

```
v[imin] = v[0]
v[0] = vmin
```

Om man upprepar detta men höjer undre gränsen från 0 till $1, 2, \ldots$ har man gjort en urvalssortering. Så här blir det färdiga programmet

```
v = [3,5,1,4,2]
n = len(v)
for a in range(n):
    imin = a
    vmin = v[a]
    for i in range(a,n):
        if v[i]<vmin:
            imin = i
            vmin = v[i]
    v[imin] = v[a]
    v[a] = vmin
print v                     => [1,2,3,4,5]
```

Det fungerar lika bra för att sortera textsträngar. Första raden kan till exempel ersättas av

```
v = raw_input("Skriv några ord: ").split()
```

6.5 Hemma hos datorn: Objekt

Varje namn i pythons namnlista är kopplat till ett *objekt*. I enklaste fall är objektet bara ett heltal, men det kan också vara mycket mer innehållsrikt. Ett objekt är ett minnesutrymme med två fack, det första är *typen*, det andra är *värdet*. Den lilla satsen `n = 17` tillverkar ett objekt med typen `int` och värdet `10001` (binärt). Namnlistan utökas med `n`, om det är ett nytt namn, och efter namnet skrivs minnesadressen till det nyskapade objektet. Det är hela kopplingen.

Dom flesta programspråk låter typen höra till variabelnamnet. Java och C kräver att man *deklarerar* typen för alla namn man inför: `int n = 17;` Nackdelen är att `n` nu bara kan ha heltalsvärden. Fördelen är att man får felmeddelande om programmet av misstag gör `n = 3.14` eller `n = "pi"`.

Dom enklaste objekten, tal och textsträngar, är oföränderliga. Om man tilldelar `n` ett nytt värde skapas ett nytt objekt – det gamla objektet kan ju inte ändras. Men mer komplicerade objekt, i första hand då listor, kan ändras av satser som `v[2] = 4711`. Förklaringen till det är namnet `v` pekar på en lista av namn, `v[0], v[1],...` som i sin tur pekar på oföränderliga objekt. Blev det klarare av det?

Typen för ett objekt kan innehålla funktionsdefinitioner. Dom kallas metoder och anropas med punkt efter variabelnamnet: `rad.split()`. Typen `str` har ett tjugotal metoder, mer eller mindre användbara.

6.6 Varningar och råd

`for x in v` låter x peka på listans element ett efter ett. Om man gör en tilldelning `x = 17` påverkar det inte `v`.

`v[3:7]` kopierar ett avsnitt, `v[:]` kopierar hela listan.

`v[2,3]` går inte, `v[2][3]` ska det vara.

`a = [[0,0,0]]*3` gör inte nio nollor utan bara tre. Därför blir matrisen `a` spöklik – ändras ett element ändras två till av sej själv!

6.7 Övningar

61. Skriv en procedur `skrivstora(v)` som skriver ut alla tal i `v` som är större än en miljon.

62. Skriv en procedur `skrivkorta(txt)` som skriver ut alla ord i textsträngen `txt` som har färre än fem bokstäver.

63. Skriv en funktion `krock(u,v)` som kollar om något element finns i båda listorna och i så fall returnerar `True`.

64. Skriv en funktion `produkt(u,v)` som returnerar skalärprodukten av två trekomponentersvektorer.

65. Skriv en funktion `median(u)` som returnerar medianen (mellersta värdet) av en godtycklig talvektor.