Python från början

Jan Skansholm

Studentlitteratur

a

Python från början

JAN SKANSHOLM

b

Kopieringsförbud

Studentlitteratur har både digital och traditionell bokutgivning. Studentlitteraturs trycksaker är miljöanpassade, både när det gäller papper och tryckprocess.

Art.nr 40543

ISBN 978-91-44-13493-2

Upplaga 1:6

studentlitteratur.se

Studentlitteratur AB, Lund

Omslagslayout: Jens Martin/Signalera

Omslagsbild: Shutterstock

Kapitelbilder: Shutterstock

Printed by Latgales Druka, Latvia 2023

Innehåll

Förord

Välkommen till *Python från början!* Detta är boken för dig som vill lära dig att programmera i Python. Du behöver inte ha några tidigare kunskaper i programmering, men det är förstås bra om du är lite van vid datorer och kan hantera filer, mappar och webbsidor.

Att programmera är ett praktiskt arbete, även om det förstås också krävs mycket tankeverksamhet. Ibland liknas programmering vid ett intellektuellt hantverk. I den här boken får du lära dig grunderna i detta hantverk, och det verktyg du ska använda i ditt praktiska arbete är programspråket Python. Allt gås igenom i en lugn takt; du får börja med de allra enklaste programkonstruktionerna och sedan bygga på dem bit för bit. Efter hand blir det lite mer avancerat.

Jag har försök att göra boken både pedagogisk och lättläst. De olika momenten gås igenom i en naturlig ordning, där allt nytt bygger på sådant som redan gåtts igenom. Det finns mängder med förklarande exempel och uppgifter att öva på. För att göra boken så överskådlig som möjligt, har jag använt olika färger och lagt ner stor omsorg på typografi och layout.

Python är ett av de mest använda programspråken. Många anser att språkreglerna i Python är relativt enkla och att det därför, även för en nybörjare, inte är så svårt att komma igång. En annan styrka med

språket är att det kan användas för att utveckla flera olika slag av applikationer. Detta tack vare att det finns en uppsjö av tilläggsmoduler, som man kan komplettera grundversionen med. Det finns t.ex. moduler för avancerade matematiska beräkningar, för grafik och för webbprogrammering. Allt detta får naturligtvis inte plats i en bok om grundläggande programmering. *Python från början* går igenom själva språket Python och de viktigaste av de moduler som ingår i standarddistributionen.

Målsättning

När du har arbetat dig igenom boken ska du kunna skriva ganska avancerade program som kan hantera data av olika slag och som kan göra olika slag av beräkningar. Målsättningen är att ge dig en stabil grund att stå på, så att du sedan kan fortsätta och studera mer avancerad programmering, t.ex. utveckling av program för matematiska och vetenskapliga beräkningar eller program för webbapplikationer. Utöver att lära ut programspråket Python ger boken dig också allmänna kunskaper i programmering som du har nytta av även om du senare programmerar i andra språk. Du kommer t.ex. att känna till tekniker för att hantera listor och filer, och du kommer att ha insikter i det objektorienterade sättet att utveckla program.

Vem passar boken för?

Denna bok är tänkt att passa som kurslitteratur i grundläggande programmeringskurser, t.ex. i en första kurs på universitet eller högskolan, i kurser för högskoleingenjörer, eller i gymnasiekurserna *Programmering 1* och *Programmering 2.* Den kan också användas i de matematikkurser på gymnasiet som numera innehåller programmering som ett moment.

Boken kan också användas av var och en som på egen hand vill lära sig grunderna i programmering eller för den som redan kan programmera i något annat språk och vill lära sig Python.

Hur läser man boken?

Du ska helst läsa boken från början till slut. Eftersom de olika avsnitten bygger på varandra, refererar många exempel och uppgifter till sådant du gjort tidigare. *Undvik därför att hoppa i boken.* Ett undantag är de avsnitt som är markerade som "överkurs". Dessa kan du hoppa över första gången du läser boken och återkomma till senare om du vill. Men det bästa är att läsa också dessa avsnitt i ordning.

[fullständigt program]

All `programtext` är tryckt med blå färg, utom i vissa program där det som är mest intressant har markerats med `rött`. [Information till läsaren: färger utgår i e-textboken.] De exempel som är fullständiga program har markerats med den bild du ser i marginalen.

I texten finns dessutom faktarutor som sammanfattar viktiga moment som just gåtts igenom. Faktarutorna ger en kort repetition och gör det också möjligt för dig att snabbt slå upp något som du behöver.

Faktaruta

Visar viktiga moment. Fungerar som snabbreferens.

I slutet av varje kapitel finns en sammanfattning som markeras med symbolen här i marginalen. Innan du går vidare till nästa kapitel bör du läsa sammanfattningen och förvissa dig om att du kan det som står där.

Övningsuppgifter

Du kan inte lära dig att programmera genom att bara läsa en bok. Du måste prova själv och träna. Därför finns det många övningsuppgifter i boken. De är markerade med den bild som visas här i marginalen. För varje nytt moment finns det en eller flera uppgifter. För att få ut det mesta av boken *bör du inte hoppa över dessa uppgifter,* utan göra dem när de dyker upp.

Förutom alla de uppgifter som ligger insprängda i texten, finns det i slutet av varje kapitel ett antal övningar av varierande svårighetsgrad. I marginalen ser du hur dessa är markerade.

[överkurs]

Vissa avsnitt, uppgifter och övningar är lite svårare och kan betraktas som "överkurs". De är markerade på detta sätt i marginalen, med en trappa.

Sist i boken finns ett omfattande sakregister. Med hjälp av detta kan du snabbt hitta det du letar efter.

Adresser

Via webbsidan `www.studentlitteratur.se` kan du hitta lösningar till alla uppgifter och övningar som finns i boken.

1 Att komma i gång

I detta första kapitel får du en kort introduktion av datorprogram och programspråk. Dessutom diskuteras olika typer av programspråk, hur program översätts och hur man får datorn att köra programmen. Du får därefter i detalj lära dig hur man skaffar sig verktyg för att kunna utveckla och

köra program skrivna i programspråket Python. Du får se två alternativ: köra på den egna datorn eller köra online.

1.1 Vad är ett datorprogram?

Vi kan börja med ett exempel. Tänk på vad det är för skillnad mellan en vanlig gräsklippare och en robotgräsklippare. Den vanliga klipparen måste man gå eller åka runt med och man måste då bestämma var den ska klippa. Det enda den kan göra är att snurra med kniven och eventuellt driva hjulen framåt. En robotgräsklippare, däremot, kör runt och klipper av sig själv. Man behöver inte gå med den. Om man vill kan man säga att robotgräsklipparen är "intelligent" medan den vanliga gräsklipparen bara är en "dum" maskin.

I själva verket innehåller robotgräsklipparen en liten *dator* som skickar ut signaler till gräsklipparens olika motorer. Datorn styrs av *ett program.* Det är detta som gör att robotgräsklipparen verkar vara intelligent. Programmet består av en följd av *instruktioner,* som steg för steg talar om för gräsklipparen vad den ska göra. Programmet kan t.ex. tala om vad som ska göras när robotgräsklipparen stöter på ett träd eller när den kommer till kanten på gräsmattan. Programmet har skrivits av en programmerare som i förväg räknat ut vilka olika situationer som kan uppstå och vad som ska göras då.

När man talar om datorer tänker man annars i första hand på vanliga skrivbordsdatorer (pc) eller på bärbara datorer (laptop). Men smarta mobiler och surfplattor är också datorer. Dessutom används datorer som delar i mer eller mindre komplicerade tekniska apparater, t.ex. i robotgräsklippare. I dag omges vi av allt fler saker som styrs av program. Man kan säga att en *dator* är en "maskin" som har till uppgift att

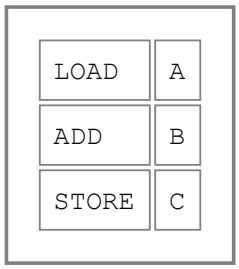
behandla och lagra information. (Datorer kallades ju från början "datamaskiner".) Grundläggande för alla datorer är att de styrs av progam.

Ett program består av en följd av instruktioner. Varje instruktion representeras i sin tur av ett antal bitar, alltså nollor eller ettor. En del av ett program kan t.ex. se ut på följande sätt:

```
0111000100001111 1001110110110001 1110000100111110
```

Om ett program är lagrat på detta sätt säger man att programmet finns i *maskinkod.* Ett program måste finnas i maskinkod för att datorns hårdvara ska kunna förstå programmet och köra det. Maskinkod kod är naturligtvis mycket besvärlig för människor att direkt skriva och förstå. Men i datorernas barndom var man tvungen att skriva program i denna form. I de riktigt tidiga datorerna, t.ex. ENIAC som blev klar 1946, var man till och med tvungen att programmera genom att ställa in ett antal knappar för hand.

Enklare blev det när man började använda s.k. *assembleringsspråk.* Då kunde man skriva programmet som vanlig text. Det kan t.ex. ser ut på följande sätt:



I assembleringsspråket motsvarar varje rad i programmet en instruktion i maskinkoden. Programavsnittet ovan består alltså av tre instruktioner. Eftersom ett program måste finnas som maskinkod när det ska köras, eller *exekveras* som man brukar säga, kan man inte direkt köra ett program som är skrivet i assembleringsspråk. Programmet måste först översättas till maskinkod. Detta görs av ett speciellt översättningsprogram, en s.k. *assembler.* Ett sådant översättningsprogram behöver inte vara så komplicerat, eftersom assembleringsspråket till sin struktur ligger så nära maskinkoden.

Exekvering

När ett program utförs (körs) i en dator säger man att programmet *exekveras.*

Assembleringsspråk är en enkel typ av *programspråk.* Även om det är mycket lättare att skriva program i ett assembleringsspråk än i maskinkod

är det fortfarande mycket komplicerat. Nästa steg i utvecklingen var att man konstruerade mer avancerade programspråk, s.k. *högnivåspråk.* Ett högnivåspråk är mer anpassat till det mänskliga sättet att uttrycka sig än till datorns maskinkod. I ett högnivåspråk skrivs programmen i en sorts halvengelska, och matematiska beräkningar skrivs på ett sätt som påminner om det man är van vid från matematiken. Programavsnittet som du såg tidigare skulle kunna se ut så här:

$$c = a + b$$

När man använder högnivåspråk kan man koncentrera sig på det problem som ska lösas och behöver inte bry sig om och känna till exakt hur datorns instruktioner ser ut.

Programspråk

För att ett program ska kunna exekveras i datorn måste det finnas som *maskinkod.* Maskinkoden är en följd av nollor och ettor.

I ett *assembleringsspråk* skrivs varje datorinstruktion i en enkel, symbolisk form. Det är krångligt att skriva assemblerprogram.

I ett *högnivåspråk* skriver man programmet i en form som är lättare att förstå för människor.

1.2 Olika typer av programspråk

Det finns två huvudprinciper för hur man köra program skrivna i högnivåspråk; man kan antingen översätta dem till maskinkod i förväg eller så kan man använda ett program som interpreterar, tolkar, dem.

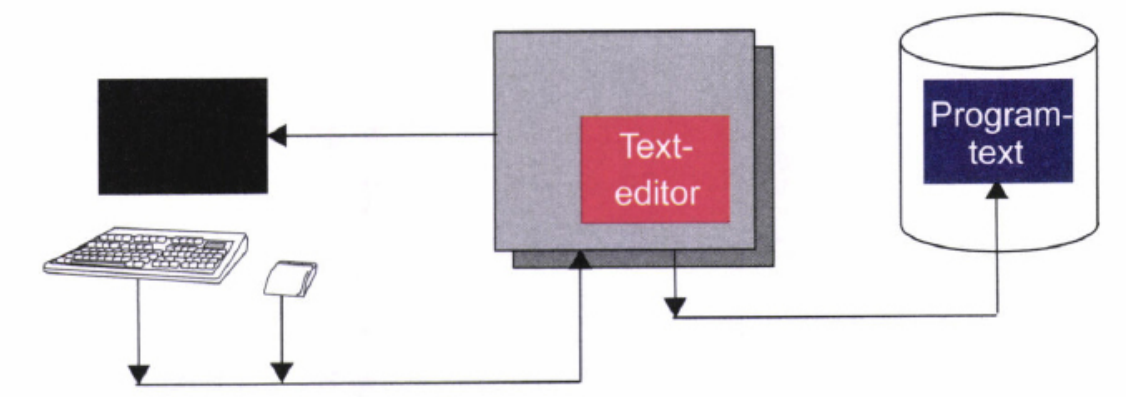
1.2.1 Kompilerade språk

Kompilerade språk är sådana språk där programmen i förväg översätts till datorns maskinkod innan man kör dem. Viktiga exempel på kompilerade språk är *C, C++* och *Objective-C.*

När man använder denna typ av språk är det första steget att skriva själva programmet. Detta gör man som en vanlig text. För att mata in den ursprungliga *programtexten,* eller *källkoden* som den ibland kallas, kan programmeraren använda ett enkelt textredigeringsprogram, en s.k. *texteditor.* (Exempel på en texteditor är *Anteckningar* i Windows.) I figur

1.1 visas hur det ser ut när en texteditor körs. Den text som man skriver in sparas i en fil.

Figur 1.1 Redigering av källkod.

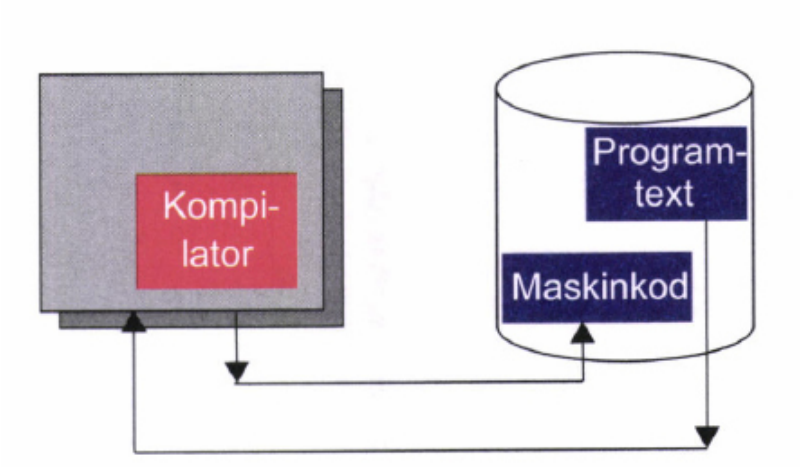


Bilden visar hur programtexten matas in ett textredigeringsprogram som visas på skärmen. Programtexten sparas sedan som en fil.

En programtext som innehåller ett program brukar sparas i en fil vars namn slutar på en bokstavskombination som anger vilket programspråk det gäller, t.ex. `.c` för C-program och `.cpp` för C++ -program. Den första delen av filnamnet kan man bestämma själv, men man bör välja ett namn som avspeglar vad filen innehåller.

När man har skrivit in programtexten är nästa steg att översätta programtexten från vanlig text till maskinkod. Detta görs som visas i figur 1.2 med hjälp av ett speciellt översättningsprogram, en s.k. *kompilator.* Varje kompilator är konstruerad för ett visst programspråk. För att kunna översätta ett C-program måste man alltså ha en C-kompilator och för att översätta ett C++-program behövs en C++-kompilator.

Figur 1.2 Kompilering.



Bilden visar hur kompilatorn översätter programtexten till maskinkod och kontrollerar att den stämmer.

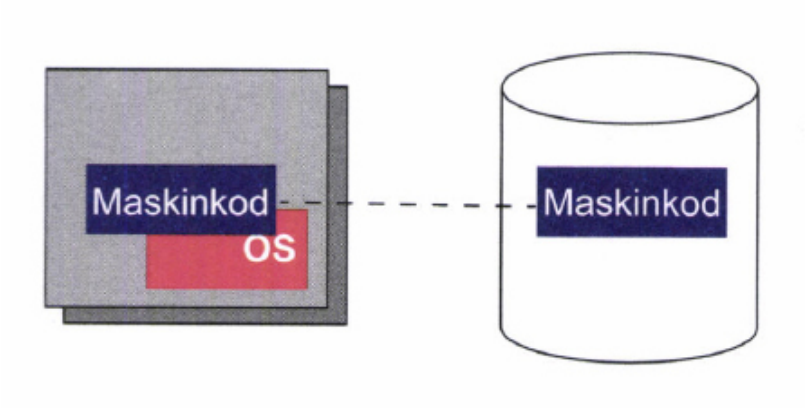
För varje programspråk finns speciella regler för hur olika programkonstruktioner får se ut. Man säger att varje språk har en viss *syntax.* Kompilatorn läser programmet från den textfil man tidigare skapat och kontrollerar först att programmet följer språkreglerna. Om kompilatorn

upptäcker några fel får man felutskrifter. Man får då gå tillbaka ett steg och rätta till felen med hjälp av texteditorn. Därefter kan man göra ett nytt försök att kompilera programmet.

Om inte kompilatorn upptäcker några fel, översätter den programmet från text till maskinkod. Kompilatorn skapar en fil i vilken den sparar den producerade maskinkoden.

Det sista steget är att få in det exekverbara programmet i datorns primärminne så att det kan köras. Studera figur 1.3.

Figur 1.3 Exekvering av program.



I de tidigare figurerna har bara ett program i taget visats i primärminnet, det program som för tillfället exekveras. I själva verket finns det alltid ytterligare ett program. Det är *operativsystemet,* t.ex. Windows, MacOS eller Linux. Användaren kan ge kommandon till operativsystemet, t.ex. begära att ett visst program ska exekveras. Ett sätt är att dubbelklicka på den exekverbara filen, ett annat att helt enkelt skriva programmets namn i ett kommandofönster. Operativsystemet kopierar då den exekverbara filen till primärminnet, såsom antyds i figuren. Därefter överförs kontrollen till det inladdade programmet som exekveras.

De olika stegen för kompilerade programspråk

Med hjälp av en texteditor skapas programtexten (källkoden).

Kompilatorn översätter programtexten till en fil med maskinkod.

Operativsystemet placerar den exekverbara filen i datorns primärminne, varefter programmet exekveras (körs).

Fördelen med kompilerade språk är att de program man utvecklar kan bli mycket snabba, eftersom de kör "riktig" maskinkod som tolkas direkt av datorns hårdvara. En annan fördel är att många fel upptäcks tidigt, vid kompileringen, innan man försöker köra programmen.

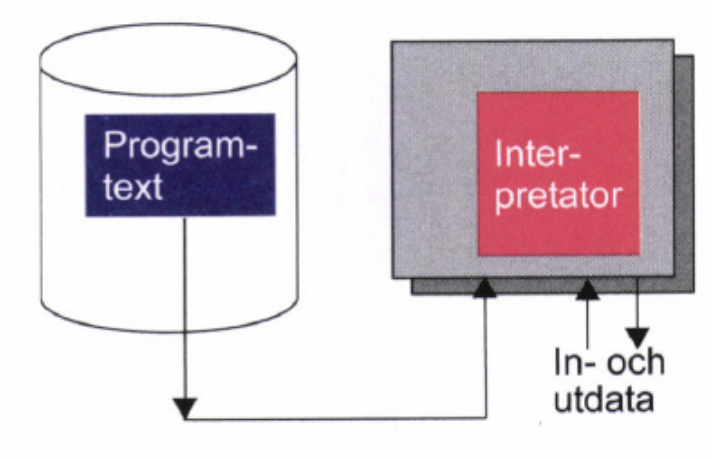
Nackdelen är att man måste utföra det extra steget med kompilering. En annan nackdel är att ett körbart program blir speciellt för den typ av dator det utvecklats för. Vill man köra samma program på en annan typ av dator måste man kompilera om det med en annan kompilator.

Kompilerade programspråk är viktiga eftersom de används när det ställs stora krav på effektivitet. De används därför för att programmera operativsystem, t.ex. Windows, MacOS och Linux, och andra systemprogram. Ett annat användningsområde är program i inbäddade system, dvs. tekniska komponenter som styrs av en inbyggd dator.

1.2.2 Interpreterade språk

Python är ett interprerat språk. Några andra exempel på vanliga interpreterade språk är *JavaScript, PHP, Ruby, BASIC* och *LISP.* I de interpreterade språken görs ingen översättning av programmen som ska köras. Men eftersom datorns hårdvara då inte kan förstå programmen måste de tolkas, interpreteras. Detta görs med hjälp av ett speciell program, en *interpretator,* eller *tolk,* som läser programtexten och ser till att de instruktioner som finns i denna utförs. Figur 1.4 visar hur det ser ut.

Figur 1.4 Exekvering med interpretator i script mode.



Bilden visar hur programtexten tolkas av en interpretator.

Utifrån ser det ut som om programmet "exekveras" i interpretatorn. Programmet kan läsa indata och producera utdata, precis som ett "riktigt" program. Men observera att det egentligen är interpretatorn som exekveras. Programtexten utgör indata till denna.

Ett viktigt område där interpreterade språk används är i webbaserade tillämpningar, s.k. client-server-tillämpningar, där användaren (klienten) använder en webbläsare, t.ex. Chrome, och där tillämpningen helt eller delvis körs på en avlägsen dator, en server. I sådana tillämpningar kan det finnas programdelar, "småprogram", som är utvecklade i ett interpreterat språk. Dessa programdelar kallas *skript (scripts* på engelska).

De programspråk som används kallas därför *skriptspråk (scripting languages).* Skript kan finnas både på klientsidan, de körs då i själva webbläsaren, och i servern. På klientsidan används nästan alltid JavaScript, men i servern finns flera alternativ. PHP är vanligast, men flera andra språk används också, t.ex. Python, Ruby, Java och JavaScript.

Ett annat område där skriptspråk används är i kommandotolkar för att tolka de kommandon som användaren skriver. Exempel är *UNIX shell scripts* och *batch scripts* i Windows kommandotolk.

Python har blivit populärt att använda när det gäller matematiska beräkningar i tekniska sammanhang. Det beror dels på att det, till skillnad från andra alternativ som *MATLAB,* är gratis och dels på att det finns flera tilläggspaket till Python som är användbara i denna typ av tillämpningar, t.ex. *NumPy* och *Matplotlib.*

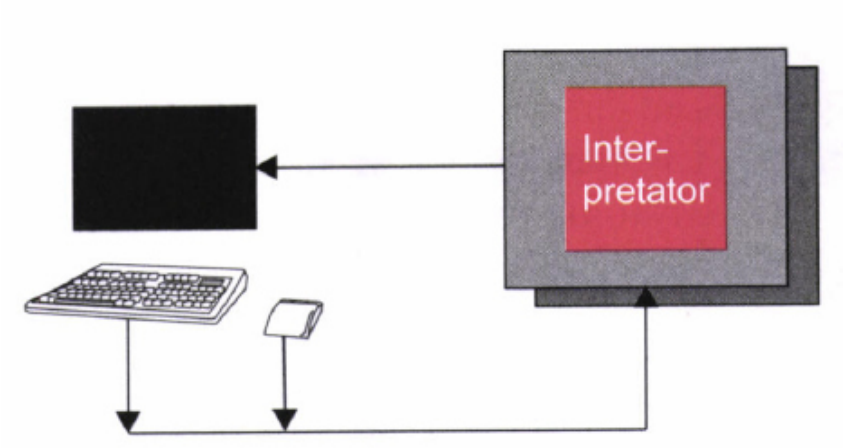
Fördelen med att använda interpreterade språk är förstås att man slipper det extra steget med kompilering. Dessutom blir de program man konstruerar

direkt flyttbara mellan olika typer av datorer. Nackdelen med interpreterade språk är att programmen ofta blir mycket långsammare än om man hade skrivit dem på traditionellt sätt och kompilerat dem till maskinkod. En annan nackdel, jämfört med kompilerade språk, är att progammen inte är syntax-kontrollerade innan man försöker köra dem, vilket ökar risken för att det kan bli fel.

I flera av de interpreterade språken kan man köra interpretatorn på två olika sätt. Man kan antingen använda den så som illustreras i figur 1.4, där den kör ett program som finns i en fil. När man använder interpretatorn på detta sätt kallas det i Python att man kör i *script mode.* Men i stället för att lägga programtexten i en fil går det ibland, t.ex. i Python, att skriva instruktionerna på tangentbordet, en instruktion i taget. Då utför interpretatorn varje instruktion direkt och visar resultatet på skärmen. Man kallar detta att köra i *interactive mode.* Detta illustreras i figur 1.5. När man använder en interpretator på det interaktiva sätt som visas i denna figur säger man ibland att man använder en *REPL,* vilket står för *read-eval-print loop.* Python-interpretatorn kan fungera som en avancerad räknedosa. Här visas ett exempel på hur det kan se ut när man kör den i *interactive mode.* Det som visas med rött har skrivits av interpretatorn.

```
>>> 5 + 9
14
```


Figur 1.5 Exekvering med interpretator i interactive mode.




```
>>> 15/6
2.5
>>> (5 + 2) * 4
28
>>> print('Hej')
Hej
```

1.2.3 Språk som kompileras just-in-time

Det finns programspråk som använder ett mellanting mellan vanlig kompilering och interpretering. Det gäller t.ex. *Java* och de språk som körs i Microsofts .NET-miljö, t.ex. C#. I denna typ av språk används också ett slags interpretator. I Java kallas den *Java Virtual Machine* och i .NET *Common Language Runtime* (CLR). Man kompilerar även här sina program, men inte till maskinkod, utan till en mellanform som i Java kallas *Java Byte Code* och i .NET *Intermediate Language* (IL).

När man ska köra ett program läser interpretatorn den kompilerade koden i mellanformen. Men interpretatorn kommer då att automatiskt kompilera programmet till maskinkod. Detta sker precis när programmet ska böra exekveras. Det kallas därför *just-in-time compilation* (JIT).

Fördelen med denna teknik är att programmen exekverar snabbare och att de program man skriver kan köras på olika typer av datorer utan att man

behöver kompilera om dem. Genom att man kompilerar till mellankod får man också en tidig kontroll av att programmen inte innehåller några syntaxfel.

Både Java och C# har breda användningsområden. De används för utveckling av användarprogram, webbtillämpningar och systemprogram. Operativsystemet i Android är t.ex. skrivet i Java och vissa bibliotek i Windows är skrivna i C#.

1.3 Installera Python

Den officiella webbplatsen för Python är `www.python.org`. Där finns mängder med information och dokumentation om Python. Från denna plats kan man också hämta en fil för att installera Python på sin egen dator. Men det är inte säkert att man behöver göra någon installation. Om man av någon anledning inte kan eller vill installera några program på sin egen dator, kan man hoppa över detta och efterföljande avsnitt och gå direkt till avsnitt 1.5. Där beskrivs hur man kan köra Python online, utan att göra några installationer.

Undersök om Python redan finns

Om man kör på en dator med Mac eller Linux är det stor chans att Python redan är installerat. I så fall behöver man inte göra någon installation. Du kan undersöka om Python är installerat genom att öppna ett terminalfönster och skriva kommandot `python3`. I Windows skriver du i stället bara `py`. Lägg märke till att du *inte* ska skriva bara `python`. Då startar nämligen eventuellt en äldre version, version 2, och den ska vi inte använda. Om Python-interpretatorn startar ska du kontrollera att det står att det är version 3. Om det stämmer, är allt redan installerat. Behåll i så fall terminalfönstret öppet med interpretatorn i gång och hoppa fram till uppgift 1.1.

Välj rätt fil att installera

Starta webbläsaren och gå till sidan `www.python.org`. Där finns en färdig distribution av Python som man kan ladda ner och installera. Denna distribution innehåller både själva Python-interpretatorn och alla de moduler som ingår i Pythons standardbibliotek. Vill man utöka sin installation med moduler som inte ingår i standardbiblioteket, går det lätt att göra detta i efterhand.

För att ladda ner rätt fil klickar du på fliken "Downloads" på sidan. Då dyker det upp en knapp på vilken det står "Python 3.7.x" eller liknande. Ovanför knappen visas vilken typ av operativsystem den gäller för, t.ex. Windows. Kontrollera att det stämmer med den dator du kör på. Stämmer det inte finns det länkar till andra typer av datorer. Det är viktigt att du väljer version 3 av Python, inte den äldre version 2.

Instruktioner för Windows och Mac

Om du kör Windows eller Mac är det nu bara att klicka på knappen. Då laddas en exekverbar installationfil ner till din dator till mappen "Downloads", "Hämtade filer" eller liknande. Starta installationen genom klicka på den nerladdade filen. Om du kör Windows, finns det längst ner i det första fönstret som dyker upp en ruta med texten "Add Python 3.7.x to PATH". Markera denna ruta innan du går vidare. Följ sedan instruktionerna.

När allt är klart öppnar du ett terminalfönster (Kommandotolken i Windows) och skriver kommandot `python3` i Mac och `py` i Windows. Då ska Python-interpretatorn starta. Kontrollera att det är version 3. Om allt ser rätt ut, behåller du terminalfönstret öppet med Pythoninterpretatorn i gång och går vidare till uppgift 1.1.

Instruktioner för Linux

Om du kör Linux, är installationen lite mer komplicerad, eftersom du då måste göra den från en fil med källkod, en s.k. *source release.* Gör så här: Ladda ner filen med den senaste versionen av Python, version 3. Det är en komprimerad fil. Gå till mappen dit den laddats ner, högerklicka på filen och

välj Expandera. Öppna sedan den nya mapp som skapas och leta reda på filen README. En bit ner i denna finns instruktioner för vilka kommandon man ska ge för att göra installationen. Öppna ett terminalfönster och flytta dig till den nya mappen. Skriv där sedan de instruktioner som står i anvisningarna. Det tar en lång stund.

När allt är klart öppnar du ett terminalfönster och skriver kommandot `python3`. Då ska Python-interpretatorn starta. Kontrollera att det är version 3. Om allt ser rätt, ut behåller du terminalfönstret öppet med Python-interpretatorn i gång och fortsätter med uppgift 1.1.

Uppgift 1.1

Oberoende av vilken typ av system du kör på bör nu Python-interpretatorn ha startat, och det ska se ut på samma sätt som det gjorde på sidan 11. Experimentera med att skriva olika matematiska uttryck. Avsluta genom att skriva kommandot `exit()` när du är klar.

1.4 Installera en IDE

I den uppgift du gjort har du kört Python i *interactive mode* där man skriver in en instruktion i taget, så som illustrerades i figur 1.5. Men nu när Python är installerat på din dator kan du också köra Python i *script mode* (se figur 1.4). Då läser interpretatorn en fil som innehåller Python-instruktioner och utför alla instruktionerna i filen.

Uppgift 1.2

Skapa en ny mapp i vilken du vill lägga dina Python-program. Öppna sedan en vanlig texteditor, t.ex. *Anteckningar* eller *Textredigerare.* Skapa en ny fil och låt den innehålla följande två rader:

```
print('Hej')
print('Välkommen till Python')
```

Spara filen i den nya mappen. Ge filen namnet `hej.py`. Filer som innehåller Python-program måste nämligen alltid ha namn som slutar på `.py`. Öppna sedan ett terminalfönster och flytta dig till den nya mappen. Det gör du med följande kommando, där *`mappnamn`* är namnet på den nya mappen.

```
cd mappnamn
```

Om du kör Windows skriver du där sedan kommandot

```
py hej.py
```

Om du kör Mac eller Linux skriver du i stället

```
python3 hej.py
```

Då kommer interpretatorn att köra programmet `hej.py` i terminalfönstret.

Egentligen finns nu allting som man behöver för att konstruera och köra Python-program. Man kan skapa programtexten i en texteditor som redan finns på datorn och man kan köra interpretatorn som nu har installerats. Men det är i längden lite opraktiskt att använda sig av två olika hjälpmedel. Därför

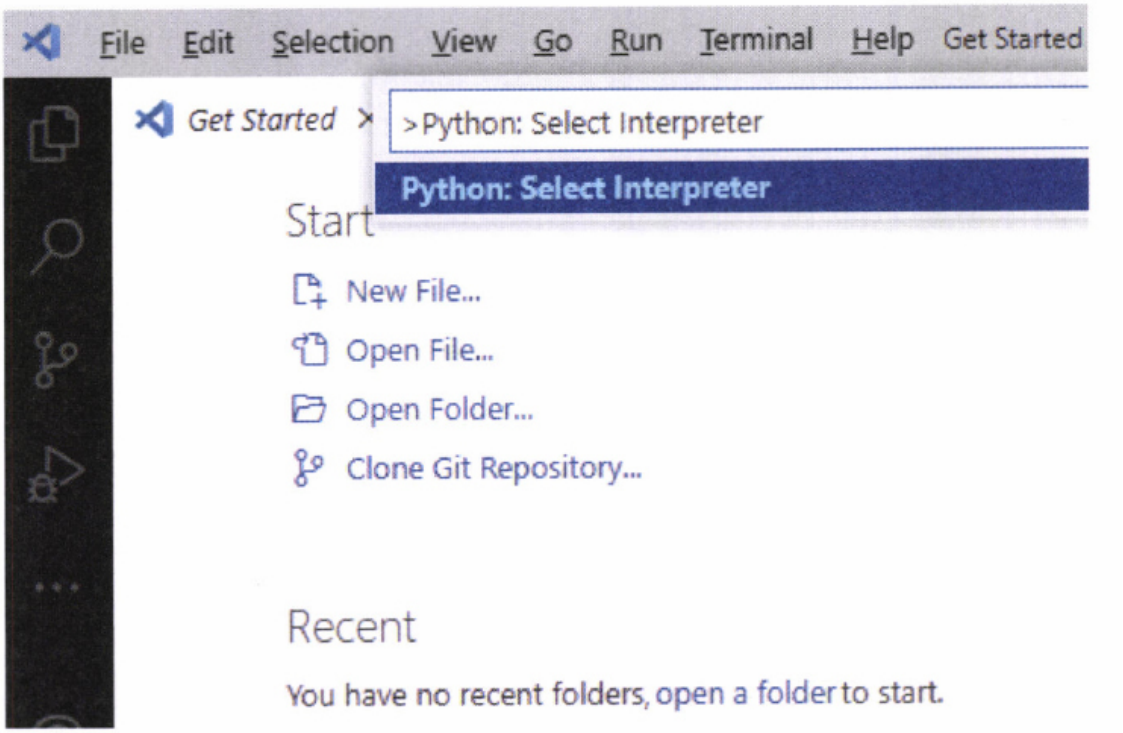
brukar man oftast använda sig av ett s.k. *integrerat programutvecklingsverktyg (integrated development environment, IDE,* på engelska). Det är ett program som innehåller alla de funktioner som behövs för att skriva, köra och hitta fel i program.

Det finns ett flertal IDE:er att välja på. Några är kommersiella, medan andra är gratis. Den IDE som ska beskrivas här heter *Visual Studio Code*

(VS *Code)* och har utvecklats av Microsoft. Det finns flera anledningar till att denna har valts: den är gratis och den lätt att ladda ner och installera. Den finns för alla de tre vanliga operativsystemen Windows, MacOS och Linux. Den är dessutom snabb och enkel att använda. Man behöver t.ex. inte skapa enskilda projekt för varje program man vill köra. Naturligtvis kan man använda en annan IDE om man skulle föredra det. Det finns flera att välja på. Ett alternativ är t.ex. *PyCharm.*

Gå till sidan `https://code.visualstudio.com`. Där finns en knapp som det står *Download* på. Kontrollera att det som står på knappen stämmer med det operativsystem du kör på. Om du t.ex. kör Windows ska det stå *Download for Windows.* Stämmer det inte, kan du klicka på nedåtpilen på knappen, så visas fler alternativ. När du klickat på knappen laddas en körbar fil till din dator. När nerladdningen är klar klickar du på filen, så startar installationen. Följ instruktionerna. När du sedan startar VS Code visas en startsida. En del av den ser ut ungefär som i figur 1.6. (Skulle sidan inte visas väljer du *Get started* på menyn *Help.)*

Figur 1.6 Visual Studio Code, startsida

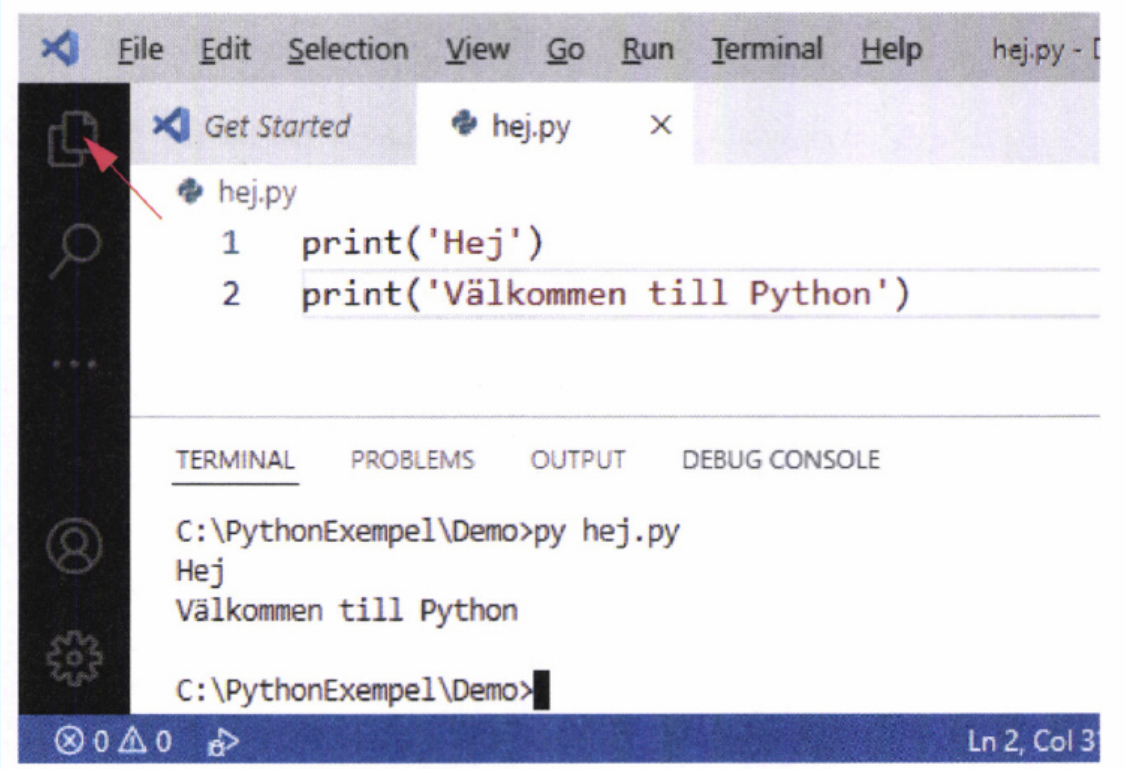


Välj alternativet *View → Command Palette* och skriv, så som visas i figuren, in texten *Python: Select Interpreter* i inmatningsfältet. Avsluta med *Enter.* Då kommer det upp en lista med installerade Python-versioner. Klicka på den version som du installerade i avsnitt 1.3.

Uppgift 1.3

Nu ska du köra programmet som du skapade i förra uppgiften. I VS Code väljer du alternativet *File → Open File.* Öppna filen `hej.py`. Den kommer att visas under en ny flik. Det ska se ut som i den övre halvan i figur 1.7. (Om orden `print` är understrukna så välj *View → Command Palette* och skriv in texten *Python: Select Linter* och klicka på *Disable Linting.)*

Figur 1.7 Visual Studio Code, första programmet.



Kör nu programmet genom att högerklicka i programtexten och välja alternativet *Run Python File in Terminal.* Du kan också välja kommandot *Run → Start Debugging* eller *Run → Run Without Debugging* eller skriva F5 resp. Ctrl+F5 på tangentbordet. Ett nytt terminalfönster ska nu öppnas under programtexten, inne i VS Code, så som det visas i den nedre halvan i i figur 1.7. Programmet kommer att köras i terminalfönstret.

(Om det istället skulle öppnas ett nytt terminalfönster *utanför* VS Code, måste du göra en justering av inställningarna. Gör så här: Välj *File → Preferences → Settings.* Skriv *Conpty* i sökrutan och tryck på Enter. Nu ska det dyka upp en ruta med rubriken *Terminal > Integrated: Windows Enable Conpty.* Ta bort markeringen i denna ruta! Kör sedan ditt program igen.)

Ändra i programmet så att det skriver ut något annat. Lägg också till en ny utskrift. Spara programmet och kör det igen. När du är klar kan du stänga terminalfönstret genom att klicka på krysset i dess högra övre hörn.

Uppgift 1.4

Välj alternativet *New File* på menyn *File.* Då skapas en ny flik i redigeringsfönstret. Skriv in följande två rader där:

```
namn = input 'Vad heter du?'
print('Hej', namn)
```

Välj sedan alternativet *Save As* på menyn *File* och spara filen i samma mapp som filen `hej.py` ligger i. Låt den nya filen heta `hej2.py`.

Kör det nya programmet genom att göra på samma sätt som i förra uppgiften. Skriv ditt namn i kommandofönstret när programmet frågar efter det.

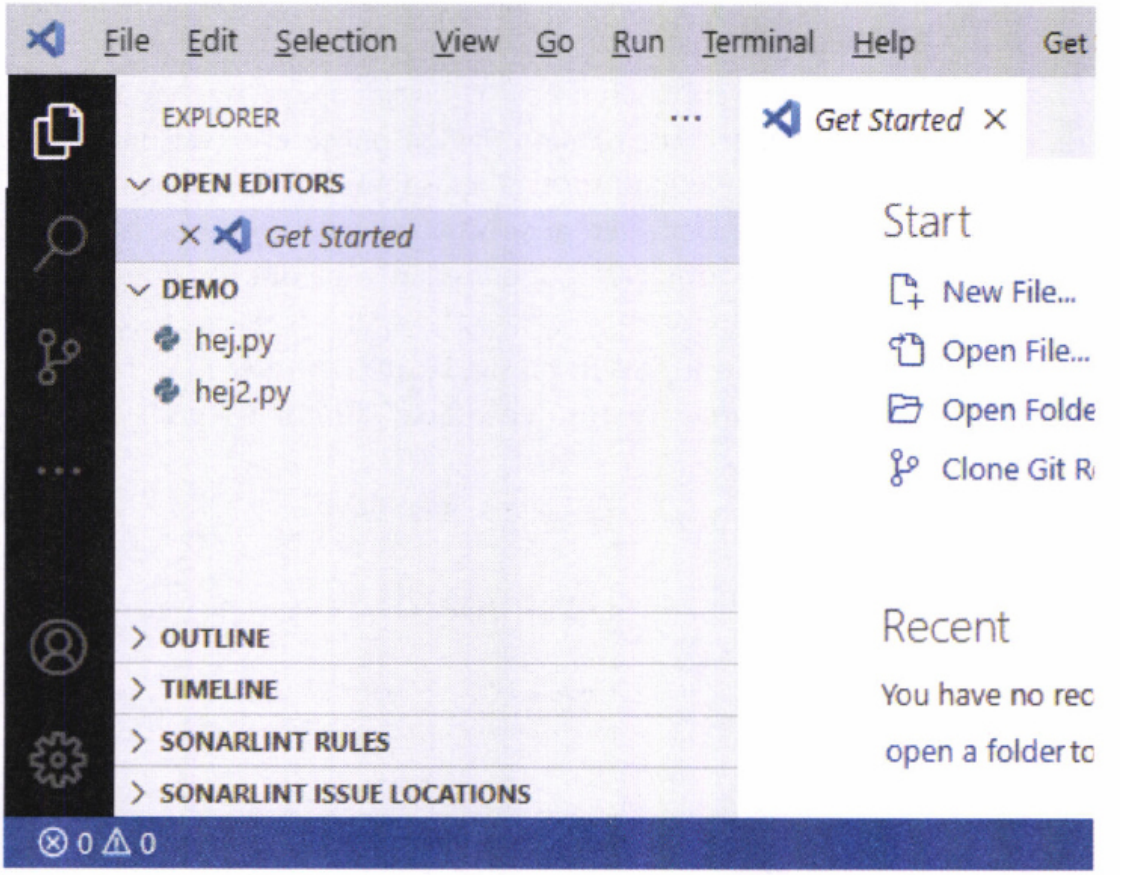
Om man klickar på symbolen med två mappar som den röda pilen pekar på i figur 1.7, öppnas ett delfält med namnet EXPLORER. Ett sådant visas i figur 1.8. Under alternativet OPEN EDITORS kan man där se vilka filer som är öppnade. Dessa filer hittar man också under de olika flikarna.

Man kan öppna mappar och visa dem i fältet EXPLORER. Det gör man genom att välja alternativet *File → Open Folder* eller genom att klicka på knappen *Open Folder* i EXPLORER-fältet. Om du gör det, och väljer den mapp som du placerade filerna `hej.py` och `hej2.py` i, kan det se ut som i figur 1.8. (Mappen som filerna ligger i heter där `Demo`.) Man kan öppna filerna i en mapp genom att dubbelklicka på dem. Man kan också köra programmet i en fil, utan att öppna filen först, genom att högerklicka på filen och välja alternativet *Run Python File in Terminal.*

Det vi hittills demonstrerat är hur man kan köra Python-program i *script mode* i VS Code. Men det går också att köra i *interactive mode.* Man väljer då alternativet *Command Palette* på menyn *View.* I fönstret som dyker upp skriver man *Python: Start REPL.* (Detta behöver man bara göra första gången. VS Code minns nämligen kommandot så man behöver bara klicka på det nästa gång.) Detta öppnar ett terminalfönster i vilket Python-interpretatorn har startats i *interactive mode.*

VS Code erbjuder många fler hjälpmedel som kan vara bra att ha när man utvecklar program. Man kan t.ex. få reda på egenskaper för en variabel eller funktion genom att hålla markören över den.

Figur 1.8 Visual Studio Code som visar fältet EXPLORER.



Man kan också debugga ett program. Då kan man gå igenom programmet stegvis och lägga in brytpunkter där man kan studera variablernas värden. Men det vi visat i detta avsnitt räcker till att börja med.

1.5 Köra Python online

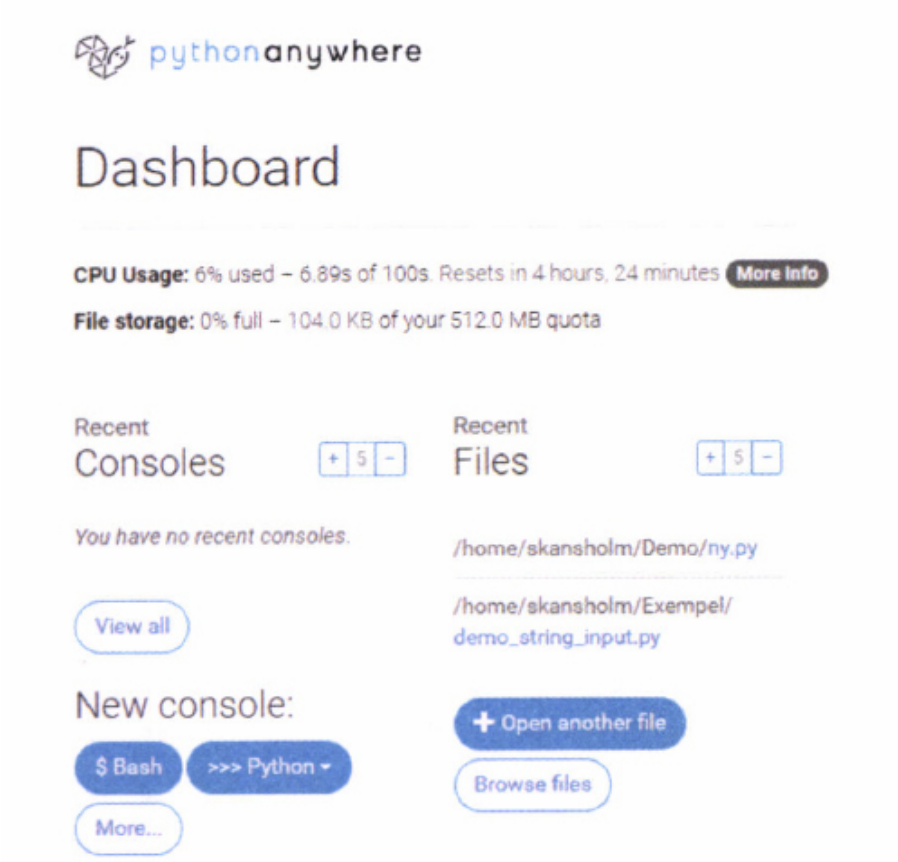
Om man inte har installerat Python på sin egen dator, finns det flera webbplatser där man kan köra online. Vill man bara köra Python i *interactive mode,* kan man använda vilken sådan webbplats som helst. Men vill man kunna spara sina programfiler, är utbudet mer begränsat, speciellt om man söker ett gratisalternativ. Här har vi valt att beskriva webbplatsen *PythonAnywhere.* På denna finns det möjlighet att skapa ett gratiskonto där man kan lagra sina egna filer. Lagringsutrymmet är begränsat, men det räcker när man ska lära sig programmera i Python.

Uppgift 1.5

Starta webbläsaren och gå till sidan `www.pythonanywhere.com`. Klicka på knappen *Start running Python online* eller välj menyalternativet *Pricing & Signup.* På sidan som då visas finns en knapp med texten *Create a Beginner account.* Det är ett avgiftsfritt konto. Klicka på knappen och fyll i dina uppgifter i formuläret. Logga sedan in på ditt nya konto.

När man loggat in på sitt konto kommer man till en sida med rubriken *Dashboard,* "instrumentbräda". I figur 1.9 visas en del av denna sida.

Figur 1.9 PythonAnywhere, Dashboard.



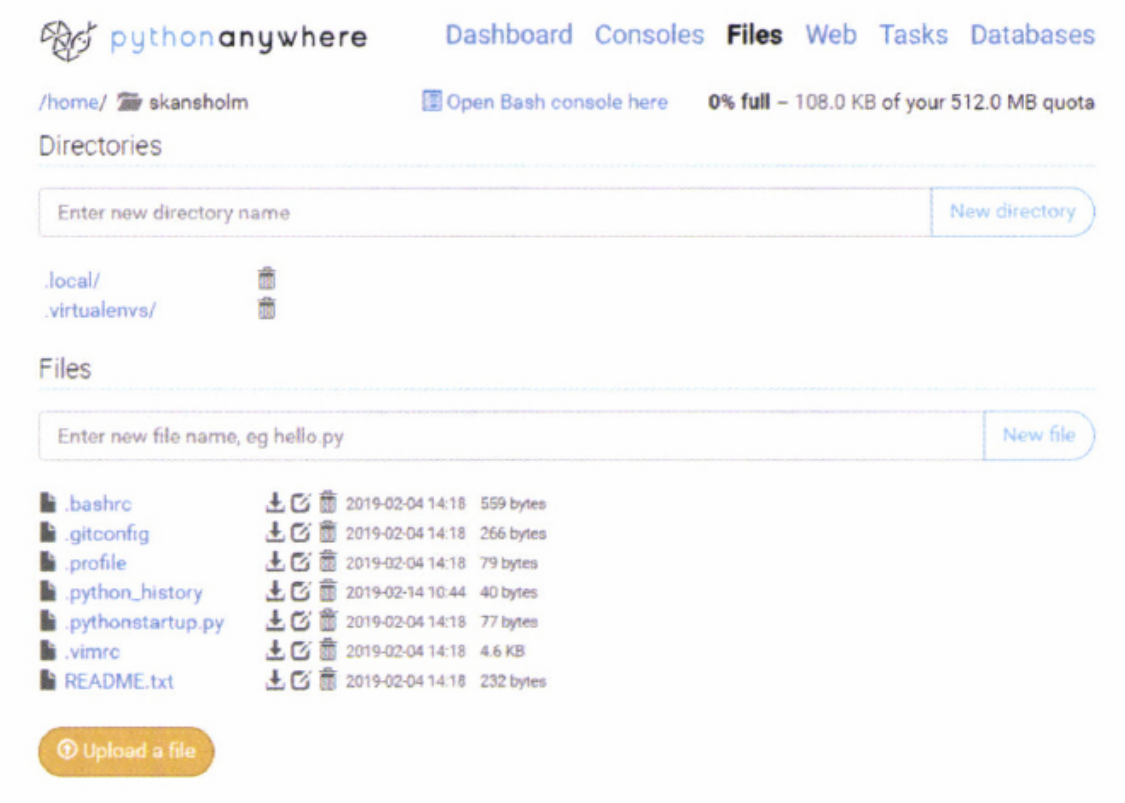
Under rubriken *New Consoles* kan man skapa och öppna nya terminalfönster. Väljer man knappen *Python* startas ett fönster där Python-interpretatorn har startats, så att man kan köra Python i *interactive mode.*

Uppgift 1.6

Tryck på knappen *Python.* (Välj den senaste versionen.) Det ska se ut på samma sätt som det gjorde på sidan 11. Experimentera med att skriva olika matematiska uttryck. Avsluta genom att skriva kommandot `exit()` när du är klar.

Man kan skapa nya mappar och filer på Python Anywhere. Om man klickar på rubriken *Files* på sidan *Dashboard,* visas en sida som ser ut ungefär som i figur 1.10.

Figur 1.10 PythonAnywhere, Files.



Längst upp till vänster ser man att man hamnat i sin hemkatalog. Där ligger redan ett antal systemfiler. (Filnamnen börjar med en punkt). Man kan lägga nya filer direkt i hemkatalogen, men det kan vara bra att lägga sina program i en egen mapp.

Uppgift 1.7

I denna uppgift ska du skapa en mapp i vilken du sedan kan lägga dina Python-program. Klicka på rubriken *Files* på sidan *Dashboard.* Under rubriken *Direcories* finns ett fält där man kan skriva in namnet på en mapp. Skriv där namnet på din nya mapp. Du kan välja vilket namn du vill. Klicka sedan på *New Directory.* Då skapas den nya mappen, och du

kommer till en ny sida som beskriver vad som finns i den. Från början är den mappen tom.

Om man står på en sida som beskriver en mapp, kan man lägga till nya filer eller undermappar. Man kan också ladda upp filer från sin egen dator. Om man lägger till en ny fil öppnas ett redigeringsfönster där

man kan skriva in den text som ska ligga i filen. I figur 1.11 visas hur det kan se ut när man skrivit ett par rader.

Figur 1.11 PythonAnywhere, Redigeringsfönster.

Uppgift 1.8

Nu ska du skapa en fil i din nya mapp. Skriv filnamnet `hej.py` i rutan under rubriken *Files* och klicka sedan på *New File.* Python-program måste alltid ha namn som slutar på `.py`. Skriv därefter in följande text i redigeringsfönstret:

```
print('Hej')
print('Välkommen till Python')
```

Klicka på knappen *Run.* Då kommer interpretatorn att startas i ett terminalfönster och instruktionerna i programmet `hej.py` kommer att utföras.

Ändra i programmet så att det skriver ut något annat. Lägg också till en ny utskrift. När du är klar kan du gå tillbaka till sidan som visar dina filer genom att klicka på mappens namn i filsökvägen längst upp till vänster.

I stället för att starta ett nytt terminalfönster varje gång man ska köra ett program, kan man starta en s.k. *Bash console.* Då får man ett terminalfönster där man kan skriva Linux-kommandon. Man kan skapa en *Bash console* på sidan Dashboard genom att klicka på knappen § *Bash,* eller så kan man, när man står i fönstret som visar mappen med filerna, klicka på texten *Open Bash console here.* Det går också att göra det när man redigerar en programtext. Detta visas i följande uppgift.

Uppgift 1.9

Skapa en ny fil med namnet `hej2.py`. Låt filen innehålla följande två rader.

```
namn = input ('Vad heter du?')
print('Hej', namn)
```

I stället för att klicka på knappen *Run* ska du nu klicka på knappen § *Bash console here.* I terminalfönstret som öppnats skriver du följade kommando:

```
python hej2.py
```

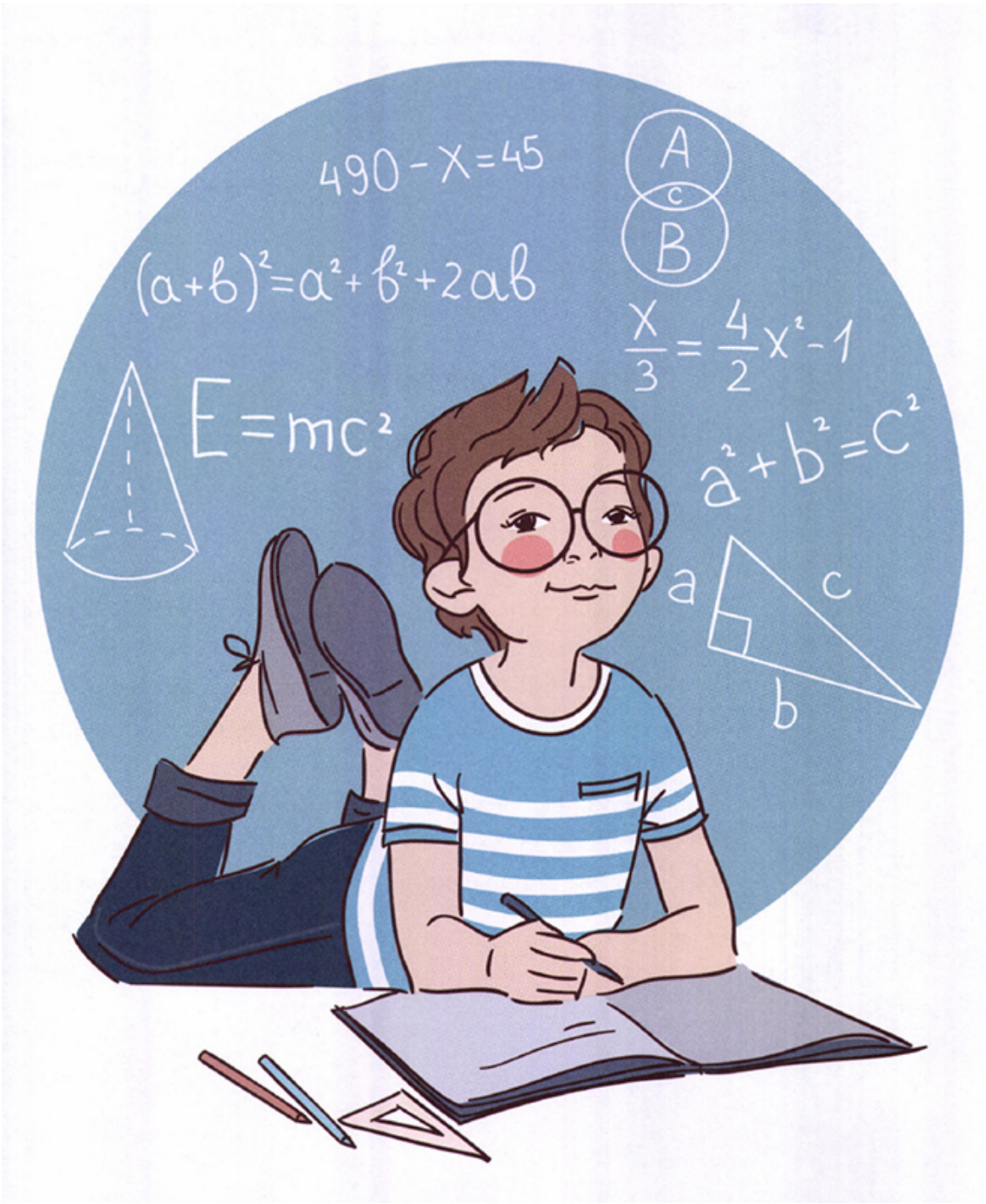
Skriv ditt namn i terminalfönstret när programmet frågar efter det.

1.6 Sammanfattning

Efter att ha läst detta kapitel bör du:

- veta vad maskinkod är,
- veta vad ett programspråk är,
- förstå skillnaderna mellan ett kompilerat och ett interpreterat programspråk och känna till några vanliga programspråk av de olika typerna,
- veta skillnaden mellan att köra i *script mode* och i *interactive mode,*
- kunna använda Python-interpretatorn i *interactive mode,* antingen på den egna datorn eller online,
- kunna skriva in en programtext med hjälp av en texteditor, en IDE eller ett verktyg online,
- kunna köra ett Python-program i *script mode,* antingen på den egna datorn eller online.

2 Att räkna



Man brukar säga att skolans grundläggande uppgift är att lära eleverna läsa, skriva och räkna. På sätt och vis gäller samma sak för programmering. Det man måste lära sig först är hur man konstruerar program som kan läsa, skriva och räkna. I detta kapitel ska vi se hur man kan räkna. För att kunna göra detta måste vi också lära oss hur man läser

in de tal man ska räkna med och hur man skriver ut de resultat man får. (Hur man hanterar text mer generellt behandlas senare i boken.)

För att kunna räkna behöver ett program använda s.k. *variabler.* Därför inleds detta kapitel med en beskrivning av hur man skapar sådana och ger dem värden. Vad som behövs för att räkna är variabler av speciella typer, avsedda för aritmetiska beräkningar. I detta kapitel kommer dessa typer att diskuteras.

2.1 Variabler och typer

När man räknar ut något kan man ibland inte göra hela beräkningen på en gång, utan man måste räkna ut en sak i taget. Efter varje steg får man fram ett mellanresultat. Man måste komma ihåg mellanresultaten eftersom man behöver dem för att räkna ut slutresultatet. Använder man papper och penna är detta inget problem; man skriver helt enkelt ner mellanresultaten.

Även i ett datorprogram måste beräkningar kunna utföras stegvis och man måste kunna spara mellanresultat för att kunna använda dem senare. Då använder man *variabler.* En variabel är som en ask, en ask som man kan lägga data i. Man kan ha många variabler i ett program. För att hålla reda på dem ger man dem därför namn. I de flesta programspråk måste man deklarera sina variabler i förväg och ibland också ange vilken typ av data de ska kunna innehålla. Men i Python är det enklare. Man behöver inga deklarationer, utan kan skapar en variabel genom att helt enkelt ge den ett värde.

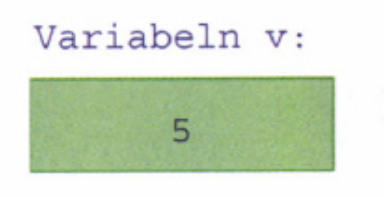
De instruktioner man ger till Python-interpretatorn kallas *satser (statements).* I normala fall skriver man en sats per rad. (Men det finns även möjlighet att skriva flera satser på en rad, eller att låta en sats omfatta

flera rader.) Det finns flera slag av satser. Den vi ska börja med är den s.k. *tilldelningssatsen (assignment statement).* Den används för att lägga ett värde i en variabel.Om man t.ex. vill skapa en variabel med namnet `v` och ge variabeln värdet 5 skriver man

$$\mathrm{v} = 5$$

När denna sats har utförts kan man tänka sig att det ser ut som i figur 2.1 där "asken" `v` innehåller värdet `5`.

Figur 2.1 En variabel som innehåller ett heltaltal.



Här har vi gett variabeln namnet `v`. Man får själv hitta på namn på sina variabler. Ett namn får bara innehålla bokstäver (även bokstäver som å, ä, ö, och é är numera tillåtna), siffror, och understrykningstecken _. Namnet får inte börja med en siffra. Man kan använda både stora och små bokstäver. När det gäller variabler är det lämpligt att man börjar med en liten bokstav. Ofta använder man stora bokstäver eller understrykningstecken för att dela upp namn som består av flera ord. Några exempel är `minText`, `ett_ord` och `minsta_värde`. Stora och små bokstäver betraktas som olika. Det betyder t.ex. att `enBok` och `EnBok` är namn på två olika variabler.

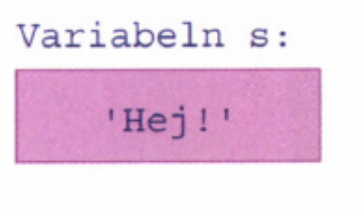
En variabel kan innehålla data av olika slag. Variabelns *typ* kan ändras. Vilken typ den får beror på vad man tilldelar den. Om man ger den ett matematiskt tal, ett s.k. *numeriskt värde,* får den typen `int` eller typen `float`. Om värdet är ett heltal får den typen `int` och om det innehåller decimaler får den typen `float`. Variabeln `v` har alltså typen `int`.

En variabel som tilldelas en text får typen `str` (förkortning av ordet *string).* Texter omges av antingen citationstecken eller enkla apostrofer. (Vi ska diskutera typen `str` mer utförligt i kapitel 5.) Vi kan t.ex. skriva

```
s = 'Hej!'
```

Då skapas en variabel som får typen `str`. Det ser ut som i figur 2.2.

Figur 2.2 En variabel som innehåller en text (en `str`).



Tilldelning

v1, v2, v3, ... = uttryck1, uttryck2, uttryck3 ...

Värdet av varje uttryck till höger om likhetstecknet beräknas först.

Detta värde placeras sedan i motsvarande variabel till vänster.

Variabelns typ bestäms av vad man tilldelar den.

Om variabeln inte finns tidigare, skapas den.

Om den redan finns, ändras dess tidigare värde och ev. även typen.

Det är faktiskt tillåtet att tilldela till flera variabler på en gång. Detta kallas *multipel tilldelning.* Vi kan t.ex. skriva

```
v, s = 5, 'Hej!'
```

Om man kör Python i interaktiv mode, kan man undersöka en variabels värde genom att helt enkelt skriva variabelns namn:

```
>>> v = 5
>>> s = 'Hej!'
>>> v
5
```

(Vi visar utskrifter från interpretatorn i *interactive mode* med röd text.)

Man kan också få reda på vilken typ en variabel har genom att använda funktionen `type`

```
>>> type(n)
<class 'int'>
```

2.2 Numeriska literaler

Du har sett att man kan tilldela ett tal till en variabel, t.ex.

$$i = 5$$

$$d = -2.9$$

Konstanta numeriska värden som man skriver i ett program kallas *numeriska literaler.* På raderna ovanför är `5` och `2.9` sådana literaler. Den första literalen har typen `int` och den andra typen `float`. En numerisk literal kan ha en decimalpunkt. Lägg märke till att man alltid använder decimal*punkt* i stället för decimal*komma.* Om man ska skriva ett tal som är mycket stort eller mycket litet kan man använda *exponentform.* Man

---

kan t.ex. skriva `2.65e15` som betyder $2,65 \times 10^{15}$ eller `1.6e-31` som betyder $1,6 \times 10^{-31}$. Man kan också ha ett minustecken framför en numerisk literal om talet är negativt. Om en numerisk literal innehåller en decimalpunkt eller är skriven på exponentform får den typen `float`, annars blir typen `int`. Observera att en literal av typen `int` inte får börja med en nolla. Det är tillåtet att lägga in ett understrykningstecken i en numerisk literal för att göra den mer läsbar. Detta påverkar inte värdet. Ett par exempel är `200_000` och `1.123_456`.

Numeriska literaler

Konstanta värden i ett program kallas literaler. Några exempel:

`123`	tal utan decimaler `(int)`
`1_234`	tal utan decimaler `(int)`
`12.3`	tal med en decimal `(float)`
`0.000_001`	tal med en decimal `(float)`
`1.2e34`	betyder 1.2×10^{34} `(float)`
`1.2e-34`	betyder 1.2×10^{-34} `(float)`

Lägg märke till att om man t.ex. skriver

$$s = '123'$$

så är `'123'` *inte* en numerisk literal. Det är en text som råkar innehålla siffror. Variabeln `s` får alltså typen `str`.

Uppgift 2.1

Starta Python-interpretatorn i *interactive mode.* Skapa variablerna `a`, `b` och `c`. Ge `a` värdet `12`, `b` värdet `12.0` och c värdet `'12.0'`. Undersök sedan vilka värden och typer de tre variablerna har. Testa sedan att skriva in några numeriska literaler på olika sätt.

2.3 Läsa och skriva numeriska data

Du har sett flera exempel på hur man kan tilldela numeriska värden till variabler när man kör i *interactive mode.* Men för att man ska kunna skriva program som räknar ut saker måste användaren kunna skriva in matematiska tal när programmet körs i *script mode.* Programmet `hej2.py` som du körde i uppgift 1.4 eller 1.9 frågade efter användarens

namn och läste sedan in namnet. Programmet såg ut så här. (I fortsättningen visar vi fullständiga program (skripts) på detta sätt, med linjer före och efter och med en markör i kanten.)

[fullständigt program]

```
namn = input('Vad heter du?')
print('Hej', namn)
```

På den första raden anropas inläsningsfunktionen `input`. Innanför parenteserna ska man skriva den fråga som man vill att programmet ska skriva ut. Programmet väntar sedan tills den som kör programmet har skrivit sitt svar. Då avslutas funktionen `input` och som resultat ger den texten som användaren skrivit. Resultatet läggs här i en variabel som ges namnet `namn`. Eftersom det användaren har skrivit är en text kommer funktionen `input` alltid att som resultat ge ett värde av typen `str`. Variabel `namn` får därför typen `str`.

Inläsning av text

Anropa `input` och ge som argument frågan som ska skrivas ut:

```
input(fråga)
```

Som resultat ger `input` det svar användaren skrivit. Typen är `str`.

På den andra raden anropas utskriftsfunktionen `print`. Innanför parenteserna anger man vad man vill ska skrivas ut. Det ska vara en text eller flera texter. Om det är flera texter räknar man upp dem med kommatecken mellan. I detta exempel är `'Hej'` den första texten som ska skrivas ut. Den andra texten ska skrivas ut är den text som finns i variabeln `namn`. Vi ger därför denna variabel som argument. Observera att det *inte*

ska vara citationstecken runt ordet `namn`. Detta ord är ju variabelnamnet, inte själva texten.

Utskrift av text

Anropa `print` och ge som argument en eller flera texter:

```
print(text1, text2, text3, ...)
```

Texterna skrivs ut efter varandra med ett blankt tecken mellan.

Varje utskrift hamnar normalt på en ny rad, men vill man att nästa utskrift ska fortsätta på samma rad kan man ge `end=''` som sista argument.

```
print(text1, text2, text3, end='')
```

30

Programmet `hej2.py` kunde läsa in och skriva ut texter, men nu vill vi kunna läsa in och skriva ut numeriska tal. Som exempel kommer här ett enkelt program som läser in ett tal och som räknar ut talet i kvadrat. Så här kan det se ut när man kör programmet i *script mode:*

```
Skriv ett heltal: 12
Talet i kvadrat är 144
```

(I *script mode* visar vi det som användaren skriver med kursiv stil.)

Så här ser programkoden ut:

[fullständigt program]

```
svar = input('Skriv ett heltal: ')
x = int(svar)
y = x * x
print('Talet i kvadrat är', y)
```

I detta program ska användaren skriva in ett matematiskt tal i stället för ett namn. Eftersom funktionen `input` alltid ger den inskrivna texten som resultat kommer variabeln `svar` att få typen `str`. I exemplet ovan kommer den att innehålla texten `'12'`. Vi vill göra om denna text till typen `int`. Det gör vi med följande rad:

```
x = int(svar)
```

Här kommer variabeln `x` att få typen `int` och den kommer att innehålla det tal som finns i texten i variabeln `svar`. Om `svar` som i exemplet innehåller texten `'12'`, kommer `x` att få värdet `12`. Om användaren skriver ett felaktigt heltal, t.ex. ett tal med decimaler, får man en felutskrift. Om man vill kunna läsa in värden med decimaler ska man i stället göra om den inlästa texten till typen `float`. Man skriver då

```
x = float(svar)
```

Omvandling av text till numeriskt värde

`int(s)` ger ett värde av typen `int`

`float(s)` ger ett värde av typen `float`

om `s` har typen `str` och innehåller ett tillåtet tal.

Man får en felutskrift om `s` innehåller ett felaktigt tal.

På nästa rad kan vi sedan räkna ut talet i kvadrat.

```
y = x * x
```


Stjärnan betyder multiplikation. Variabeln `x` ska alltså multipliceras med sig själv. Vi lägger resultatet i variabeln `y`.

Sist i programmet skriver vi ut resultatet med hjälp av `print`.

```
print('Talet i kvadrat är', y)
```

Variabeln `y` har typen `int`, men dess värde kommer automatiskt att göras om till en text vid anropet av `print`.

Uppgift 2.2

Skriv en ny version av programmet som räknar ut kvadraten av ett tal. I den nya versionen ska man kunna beräkna kvadraten av vilket tal som helst, inte bara heltal. Spara programmet i en fil med namnet `kvad.py`. Kör programmet i *script mode* ett par gånger. Låt det först räkna ut kvadraten av 1,5 och sedan av 1,6.

Den första utskriften i uppgiften du just gjorde såg bra ut. Då gav du 1,5 som indata. Men den andra utskrifen såg inget vidare ut. Med indata 1,6 fick du utskriften

```
Talet i kvadrat är 2.5600000000000005
```

De första siffrorna är rätt, men därefter kommer en massa nollor och sedan en konstig siffra på slutet. Detta hänger ihop med hur numeriska värden av typen `float` lagras i datorn. Talen lagras digitalt och alla tal kan därför inte lagras exakt. Hur lagringen går till beror på det programspråk själva Python-interpretatorn är skriven i, men man bör alltid kunna vara säker på att de första 15 decimala siffrorna är korrekta. Om du räknar siffrorna i utskriften ser du att de första 15 siffrorna faktiskt är korrekta. Vi kan därför strunta i de två sista. Men vi vill att utskriften ska vara lite snyggare. Det finns flera sätt att redigera utskrifter i Python, men vi ska visa hur man kan använda *formatted string literals,* s.k. *f-Strings.* Dessa finns från och med version 3.6 av Python.

Vi börjar med ett exempel. Vi ändrar den sista raden i programmet `kvad.py` så att det ser ut på följande sätt.

[fullständigt program]

```
svar = input('Skriv ett tal:')
x = float(svar)
y = x * x
print(f'Talet i kvadrat är {y: .2f}')
```

Om vi nu kör programmet och ger 1.6 som indata blir resultatet

```
Talet i kvadrat är 2.56
```

Det är det som i programmet markerats med rött som är intressant. Vi använder en f-String. För att det ska bli en f-String skriver man bokstaven `f` före citationstecknet eller apostrofen. En f-String kan innehålla vanlig text som ska skrivas ut precis som den är, men den kan också innehålla *platshållare (placeholders').* En platshållare omges av klamrarna `{}`.

I en platshållare kan man ange både *vad* som ska skrivas ut och *hur* det ska skrivas ut. I detta exempel har vi angivit att variabeln `y` ska skrivas ut. Man kan lägga till en *formatspecifikation* som anger *hur* utskriften ska redigeras, eller formateras som man brukar säga. Formatspecifikationen inleds med ett kolon. I exemplet ovan är formatspecifikationen `:.2f`. En formatspecifikation innehåller en bokstav som anger vilken typ av data det är fråga om. Bokstaven `f` som står här betyder att det gäller en `float` som ska skrivas ut på fast form (inte med exponent). Det som står efter punkten anger hur manga decimaler man vill ha. Här står det `2` vilket betyder att det ska vara två decimaler. Avrundning av den sista decimalen sker alltid på normalt sätt.

Om man vill kan man skriva ett tal framför punkten. Detta tal anger i så fall hur många positioner (tecken) som det *minst* ska vara i utskriften. Om det behövs fler tecken än vad man angett, kommer man ändå att få precis så många tecken som behövs. Om det behövs färre tecken än man angivit kommer det att ske utfyllnad. Om den sista raden i programmet hade sett ut på följande sätt

```
print(f'Talet i kvadrat är {y:6.2f}')
```

hade man fått två extra blanka tecken före utskriften av resultatet:

```
Talet i kvadrat är 2.56
```

Normalt sker utfyllnad med blanka tecken, men man kan också fylla ut med nollor. Då skriver man en nolla först i formatspecifikationen. Om man t.ex. kör följande program

```
z = 1.666
print(f'Resultat: {z:06.2f}')
```

kommer utskriften att bli

```
Resultat:001.67
```

Uppgift 2.3

Ändra i programmet `kvad.py` så att en f-String används i utskriften. Prova med att ange olika antal decimaler och postitioner. Testa med olika indata.

Vill man skriva ut ett heltal i stället för ett tal med decimaler skriver man inte bokstaven `f` i formatspecifikationen. Man ska i stället skriva bokstaven `d` eller inget alls. Man kan då inte ange antalet decimaler, utan bara antalet postitioner (tecken) i utskriften. Man kan också utelämna antalet positioner. Då får man precis så många som behövs.

[fullständigt program]

```
i = 19
print(f'Resultat:{i}')
j = 107
print(f'Resultat:{j:5}')
k = 4
print(f'Resultat:{k:03}')
```

Utskriften blir

```
Resultat:19
Resultat:  107
Resultat:004
```

Det får finnas flera platshållare i en f-String. I programmet ovan kan vi t.ex. lägga till raden

```
print(f'i={i} och j={j}')
```

Denna ger utskriften

$i = 19$ och $j = 107$

f-String

Inleds med bokstaven `f`. Kan innehålla vanlig text och en eller flera platshållare omgivna av klamrar `{}`.

Platshållarna ange *vad* som ska stå i texten och *hur* det ska redigeras.

```
f'text1{vad:hur} text2{vad:hur} ...'
```

vad kan vara namnet på en variabel eller ett uttryck.

hur anges med en *formatspecifikation.*

hur kan utelämnas. Då får man ett standardformat.

Reglerna för vad som kan stå i en formatspecifikation är ganska omfattande. Här har vi bara visat det allra enklaste. På sidan `https://docs.python.org/3/library/string.html#formatspec` finns en fullständig beskrivning.

2.4 Aritmetiska uttryck

Precis som i vanlig matematik kan man bilda uttryck som innehåller de fyra räknesätten addition, subtraktion, multiplikation och division. Man använder tecknen +, −, * och /. Vi kan t.ex. skriva uttrycken

```
5+3 a-1 5 * b a / 8
```

Ett uttryck som resulterar i ett numeriskt värde brukar i programmeringssammanhang kallas ett *aritmetiskt uttryck.* Tecknen `+, -, *` och `/` kallas aritmetiska *operatorer.* Det som står före eller efter en operator kallas *operander.* Du ser att operanderna i ett aritmetiskt uttryck kan vara både literaler och variabler.

I ett program kan man alltid beräkna ett aritmetiskt uttrycks *värde.* Variablerna i uttrycket har ju värden som kan användas vid beräkningen. Om vi t.ex. tidigare i programmet hade skrivit

$$a = 12$$

$$b = 2$$

skulle uttrycken ovan fått värdena 8, 11, 10 respektive 1,5.

Uttryck

```
operand operator operand
```

Uttryckets *värde* beräknas när programmet körs.

Man kan bilda mer komplicerade uttryck, med flera operatorer:

```
5 * a + b
a / b * 4.25 + 19.3
```

I uttryck med flera operatorer är det viktigt att beräkningarna utförs i rätt ordning. Operatorerna har samma prioritet som i vanlig matematik. Operatorerna `*` och `/` har högre prioritet än `+` och `-`. Anta t.ex. att vi skriver följande när vi kör i *interactive mode*

```
>>> i = 13
>>> j = 2
>>> i + 3 * j
19
```

Att uttrycket får värdet 19 beror på att multiplikationen görs först. Regeln är följande: Om ett aritmetiskt uttryck innehåller flera operatorer, utförs den eller de operatorer som har högst prioritet först. Om flera operatorer har samma prioritet beräknas operatorerna från vänster till höger. Man kan ändra denna ordning genom att sätta in parenteser.

```
>>> (i + 3) * j
32
```

I detta uttryck blir resultatet 32 eftersom additionen utförs först.

Ett uttryck har inte bara ett värde. Det har också en typ. Om någon av operatorerna i ett uttryck har typen `float` får resultatet typen `float`. Om båda operatorerna har typen `int` får resultatet typen `int`, såvida det inte gäller division. Resultatet av operatorn `/` är nämligen alltid en `float`, även om båda operatorerna är av typen `int`.

Uppgift 2.4

Skriv ett program som läser in en kvadrats sida. Programmet ska beräkna omkretsen av kvadraten samt dess area. Kör programmet i *script mode.*

Det finns fler aritmetiska operatorer i Python än de fyra räknesätten. Operatorn `//` dividerar den första operanden med den andra och ger resultatet avhugget till närmaste lägre heltal. Här är några exempel.

```
>>> 13 // 5
2
>>> 8 // 2.5
3.0
>>> 9.0 // 2
4.0
```

Operatorn `%` kan användas när man vill veta vilken rest det blir när man dividerar.

```
>>> 13 % 5
3
```

(Talet 5 går 2 hela gånger i 13 och då blir det 3 över.)

En annan aritmetisk operator är `**`. Den kan man använda för att räkna ut den första operanden upphöjd till den andra, t.ex.

```
>>> 5 ** 3
125
>>> −2**3
−8
>>> 2**−3
0.125
```

Det första uttrycket räknar ut "fem upphöjt till tre", dvs. 5*5*5. Det andra räknar ut (-2)*(-2)*(-2). Det tredje räknar ut (1/2)*(1/2)*(1/2).

Aritmetiska operatorer

`+ - * /` addition, subtraktion, multiplikation och division

`//` ger resultatet av divisionen avhugget till helt tal

`%` ger resten vid division

`**` ger den första operanden upphöjd till den andra

Operatorerna `* / // %` och `**` har högre prioritet än `+` och `-`.

Uppgift 2.5

Skriv ett program som läser in en varas pris, *inklusive* moms. Programmet ska också läsa in momssatsen som anges i procent. Programmet ska beräkna dels varans pris *exklusive* moms och dels momsen. De två resultaten ska skrivas ut på ett snyggt sätt. Kör programmet i *script mode.*

Uppgift 2.6

Skriv ett program som läser in antalet sekunder till en variabel med namnet `tid`. Indata till programmet ska vara ett helt antal sekunder, dvs. ett heltal. Programmet ska räkna om antalet sekunder så att det kan uttryckas i timmar, minuter och sekunder, där antalet minuter och sekunder ska ligga i intervallet 0 till 59. Använd tre variabler: `tim`, `min` och `sek`. Skriv satser som beräknar antalet timmar, minuter resp. sekunder och tilldelar dessa värden till variablerna. *Tips:* För att se

hur många timmar det blir kan du se hur många gånger 3600 "går i" antalet sekunder. Sedan kan du se hur många gånger 60 går i den rest som blir kvar osv. Kör programmet i *script mode.*

2.5 Matematiska standardfunktioner

Det finns några inbyggda standardfunktioner. Funktionerna `max` och `min` jämför två tal och ger som resultat det största respektive minsta:

```
x = max(a, b)
y = min(a, b)
```

Funktionen `round` avrundar ett tal till ett önskat antal decimaler:

```
z = round(a, 2)
```

Funktionen `abs` beräknar absolutvärdet av ett tal `x`, dvs. `x` om

$$x \geq 0$$

och

$$-x$$

annars:

```
y = abs(x)
```

Det finns dessutom ett antal standardfunktioner som är deklarerade i en modul som heter `math`. Modulen innehåller också definitioner av de matematiska konstanterna π och e. En sammanställning av de vanligaste funktionerna i `math` ges i faktarutan. Observera att för de trigonometriska funktionerna mäts vinklar i s.k. *radianer.* Ett helt varv (360 grader) motsvaras av 2π radianer.

Modulen `math`

`pi`	konstanten π
`e`	konstanten *e*
`exp (x)`	ger e^{x}
`log(x)`	ger den naturliga logaritmen (ln) av `x`
`log10 (x)`	ger 10-logaritmen av `x`
`sqrt (x)`	ger $\sqrt{x}$
`ceil (x)`	ger det minsta hela tal som är $\geq$ `x`
`floor (x)`	ger det största hela tal som är $\leq$ `x`
`pow (x, y)`	ger x^{y} (Om $x \leq 0$ måste `y` vara ett helt tal)
`sin(x), cos(x), tan(x)`	`x` anges i radianer
`radians (x)`	översätter grader till radianer
`degrees (x)`	översätter radianer till grader

En fullständig beskrivning av alla funktioner i modulen `math` finns på `https://docs.python.org/3/library/math.html#module-math`

För att kunna använda funktionerna i modulen `math` måste man i sitt program först importera modulen `math`. Det gör man genom att skriva

---


```
import math
```

Vi har skrivit ordet `import` med fet stil eftersom det är ett sk. *reserverat ord* eller *keyword* i Python. Sådana ord får inte användas till något annat. Vi skriver alla reserverade ord med fet stil i fortsättningen.

Här är några exempel på hur funktionerna kan anropas. Observera att man måste skriva `math.` framför funktionens namn.

```
3.6 *math.sqrt(z)
math.log(z)/math.pi
math.log10(sin(y))
```

Som exempel visas här ett program där Pythagoras sats $c^2 = a^2 + b^2$ används för att beräkna hypotenusans längd i en rätvinklig triangel. (*C* betecknar hypotenusans längd och *a* och *b* de andra sidornas längder.) Programmet läser in värdena *a* och *b* och placerar dem i två variabler med namnen `a` och `b`. Omvandlingen till typen `float` görs direkt vid inläsningen. Då slipper vi att skapa två extra variabler. Hypotenusans längd beräknas sedan och läggs i variabeln `c`. Vi använder formeln $c = \sqrt{a^2 + b^2}$.

[fullständigt program]

```
import math
a = float(input('Första sidan?'))
b = float(input('Andra sidan?'))
c = math.sqrt(a**2 + b**2)
print(f'Hypotenusans längd: {c:.2f}')
```

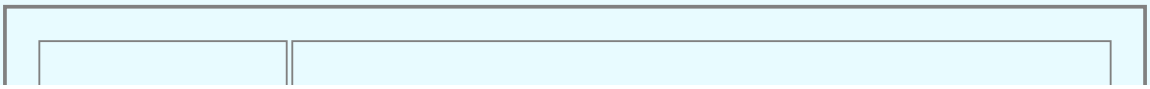
Uppgift 2.7

Skriv ett program som läser in en cirkels radie. Programmet ska beräkna cirkelns omkrets och area och skriva ut de två resultaten på ett snyggt sätt med tre decimaler. Kör programmet i *script mode.*

2.6 Modulen random

En annan standardmodul som kan vara intressant är `random`. I den finns ett antal funktioner med vilkas hjälp man kan generera slumpmässiga tal. Ett urval av funktionerna visas i faktarutan. (Funktionerna `choice`, `shuffle` och `sample` har att göra med sekvenser som vi går igenom senare i boken. De behöver du inte bry dig om ännu.)

Modulen random



`random()`	ger ett slumptal `x`, där $0 \leq x < 1$
`uniform (a, b)`	ger ett slumptal `x`, där $a \leq x \leq b$
`randint (a, b)`	ger ett slumpmässigt heltal `k`, där $a \leq k \leq b$
`choice(sek)`	ger ett slumpmässigt element från sekvensen `sek`
`shuffle(lis)`	flyttar om elementen slumpmässigt i listan `lis`
`sample(sek,n)`	ger en lista med `n` st slumpmässigt valda element från sekvensen `sek`

En fullständig beskrivning av alla funktioner i modulen `random` finns på `https://docs.python.org/3/library/random.html#module-random`

Slumptal har man nytta av när man ska konstruera program för spel eller simulering. Här kommer ett enkelt exempel som simulerar ett kast med en tärning. Modulen `random` måste också importeras.

[fullständigt program]

```
import random
print('Tärningen är kastad')
n = random.randint(1,6)
print('Du fick', n)
```

Uppgift 2.8

Utöka ovanstående program så att man slår med två tärningar samtidigt. Programmet ska skriva ut summan av de båda tärningarna.

2.7 Kommentarer

För att göra program lättare att förstå kan man lägga in *kommentarer.* En kommentar är bara till för den mänskliga läsaren av programmet. Python-interpretatorn hoppar över kommentarer och struntar i vad som står i dem.

Att använda kommentarer i sina program är nödvändigt. Man kan säga att det finns två kategorier av kommentarer: Den första kategorin är till för den som ska läsa själva programkoden och eventuellt göra förändringar i den. Avsikten med dessa kommentarer är att göra programmet mer lättläst och begripligt. De kan t.ex. förklara knepiga konstruktioner

i koden. Den andra typen av kommentarer är till för dem som vill använda färdiga funktioner, men som inte är intresserade av att veta hur koden ser ut inne i funktionerna. Denna typ av kommentarer kan t.ex. beskriva vad en funktion gör och vilka parametrar den har.

I Python använder man tecknet `#` för att markera att en kommentar börjar. Allt som står på resten av *samma* rad kommer då att uppfattas som en kommentar.

```
... # Detta är en kommentar
```

Som exempel kan vi lägga till ett par kommentarer i programmet som simulerade tärningskast i förra avsnittet.

[fullständigt program]

```
# Ett program som simulerar ett tärningskast
import random
print('Tärningen är kastad')
n = random.randint(1,6) # ett slumtal mellan 1 och 6
print('Du fick', n)
```

Uppgift 2.9

Lägg in lämpliga kommentarer i det program du skrev i uppgift 2.7

2.8 Utökade tilldelningar

När man programmerar vill man ofta ändra en variabels värde. Mycket vanligt är t.ex. att man vill använda en variabel som räknare och öka dess värde med ett. Vill man öka variabeln `n`:s värde med ett kan man förstås skriva

$$n = n + 1$$

Man kan också vilja ändra en variabels värde på andra sätt, t.ex. addera ett visst värde till variabeln, subtrahera ett viss värde eller multiplicera en variabel med ett visst tal. Några exempel är

$$i = i + 2$$

```
j = j - k

k = k * i
```

Då är det praktiskt att använda de utökade formerna av tilldelningsoperatorn *(augmented assignment* på engelska). Med hjälp av dessa kan satserna ovan skrivas som

```
i+ = 2  #  samma  som  i = i + 2
j- = k  #  samma  som  j = j - k
k* = i  #  samma  som  k = k * i
```

Det finns fler utökade tilldelningsoperatorer (se faktarutan).

Utökade tilldelningsoperatorer

```
+=  -=  *=  /=  //=  %=  **=
```

Uttrycket: $x\bullet = y$

betyder samma som: $x = x \bullet (y)$

2.9 Fel

Det är nästan omöjligt att skriva ett program utan att något blir fel. I detta avsnitt ska du få lite tips på hur man kan göra för att hitta felen. En del fel som dyker upp när man just har skrivit in sitt program är ofta rena skrivfel. Man kan t.ex. ha glömt en apostrof eller en avslutande parentes, eller så kan man ha stavat ett variabelnamn fel. Alla programspråk har bestämda regler för hur programkoden får se ut. Dessa regler kallas språkets *syntax.* Gör man enkla skrivfel innebär det ofta att man bryter mot språkets syntax. Vi kan kalla den typen av fel för *syntaxfel.*

Om man försöker köra ett program som innehåller syntaxfel i Python-interpretatorn får man en felutskrift och programmet stoppas. Men om man använder en IDE för att skriva in programkoden, kan man ofta få hjälp med att upptäcka felen redan innan man försöker köra programmet. Som exempel ska vi ta följande, avsiktligt felaktiga program.

```
# Ett felaktigt program!
tal = input('Skriv ett tal:')
rot = math.sqrt(tal
print(f'Kvadratroten av talet är {rot:.2}')
```

Vi skriver in detta program med hjälp av en IDE. I VS Code ser det då ut som i figur 2.3 och i Python Anywhere som i figur 2.4.


```
1 # Ett felaktigt program!
2 tal = input('Skriv ett tal:')
3 rot = math.sqrt(tal
```

```
4 print(f'Kvadratroten av talet är {rot:.2}')
```

Figur 2.3

```
1 # Ett felaktigt program!
2 tal = input('Skriv ett tal:')
3 rot = math.sqrt(tal
4 print(f'Kvadratroten av talet är {rot:.2}')
```

Figur 2.4

Vi ser att i båda fallen har rad 4 markerats. Om vi i VS Code håller markören över det understrukna ordet `print` visas felmeddelandet

```
unexpected token 'print'
invalid syntax line 4
```

Om vi i Python Anywhere håller markören över det röda krysset får vi kort och gott meddelandet

```
Syntax Error
```

Men är det något fel på rad 4? Nej, felet är i slutet på raden ovanför. Där fattas en högerparentes. Men detta upptäcker inte texteditorn förrän på nästa rad. Man ska därför inte alltid tro att felet finns exakt på det ställe där det markeras.

Vi rättar felet genom att lägga till en högerparentes sist på rad 3. I VS Code ser vi att ordet `math` fortfarande är understruket. Om vi håller markören över ordet visas utskriften

```
'math' used before definition
```

När vi i Python Anywhere skrivit dit högerparentesen som fattas på rad 3 försvinner inte det röda krysset direkt, men när vi sparar filen eller försöker köra den tas det röda krysset bort. I stället dyker det upp en varningstriangel på rad 3. Om vi håller markören över denna triangel får vi meddelandet

```
undefined name 'math'
```

Är det något fel på ordet `math`? Nej, det är det inte. Vad som är fel är att vi glömt att man måste importera modulen `math` först i programmet. Eftersom vi inte gjort det uppfattas `math` som en vanlig variabel.

Nu lägger vi till följande rad först i programmet

```
import math
```

och därefter sparar vi programmet. Då försvinner alla felmarkörer. Vi kör programmet och skriver in ett tal när programmet frågar efter det. Men då får vi flera rader med felutskrifter. De sista är

```
File
'c:\Users\Jan\Documents\Böcker\Python\Ex\fell.py',
line 4, in <module>
rot = math.sqrt(tal)
TypeError: must be real number, not str
```

Det vi här har råkat ut för är ett s.k. *exekveringsfel.* Det är sådana fel som inte upptäcks när man skriver in programmet (eller kompilerar det i ett kompilerat programspråk). Ett exekveringsfel upptäcks först när man kör programmet.

Den sista raden i felutskriften talar tydligt om vad felet är i vårt program. Variabeln `tal` får inte vara av typen `str`. Den måste vara numerisk. Vi har glömt att göra om det inlästa talet på rad 2 till en `float`. Vi skriver därför om rad 2:

```
tal = float(input('Skriv ett tal:'))
```

När vi nu kör om programmet fungerar det som det ska.

Goda råd när man rättar fel

- Studera alltid det första felmeddelandet först.
- Lita inte på att den rättelse som föreslås är riktig.
- Rätta det första felet och spara programmet.
- Om du får ett felmeddelande som säger att något är okänt eller används innan man definierat det: kontrollera då att

  - du har stavat namnet rätt (var noga med små och stora bokstäver),

  - du inte har glömt att importera den modul som innehåller definitionen av det aktuella namnet.

- Strunta inte i varningar.

2.10 Sammanfattning

Efter att ha läst detta kapitel bör du:

- veta hur man tilldelar ett värde till en variabel,
- veta vilka de numeriska typerna är,
- veta skillnaden mellan typen `str` och de numeriska typerna,
- veta vad en literal är och hur man skriver sådana i programmet,
- veta hur man gör för att läsa in och skriva ut texter och numeriska värden,
- veta hur man kan göra enkla redigeringar av numeriska utskrifter,
- veta vad ett aritmetiskt uttryck är,
- känna till de aritmetiska operatorerna `+ - * / // &` och `**`,
- känna till de utökade tilldelningarna,
- veta hur man skriver kommentarer i sina program,
- kunna använda de vanligast funktionerna i modulen `math`,
- kunna tolka de vanligaste felmeddelanden som man kan få när man försöker skriva och köra ett program.

2.11 Övningar

2.1 Skriv ett program som beräknar hur många mil en bil har gått under det senaste året och bilens genomsnittliga bensinförbrukning per mil. När programmet körs ska det fråga efter dagens mätarställning, mätarställningen för ett år sedan och hur många liter bensin som förbrukats under året. Det ska se ut som i följande exempel när man kör programmet.

```
Mätarställning i dag? 7891
Mätarställning för ett år sedan? 6404
Antal körda mil: 1487
Antal liter bensin: 1235.4
Förbrukning per mil: 0.83
```

2.2 Skriv ett program som beräknar volymen och arean av en sfär. Som indata ges sfärens radie. Följande formler är givna

$$V = \frac{4\pi r^3}{3} \; A = 4\pi r^2$$

2.3 I USA brukar man ange temperaturer i grader Fahrenheit $(^\circ\mathrm{F})$ i stället för grader Celsius $(^\circ\mathrm{C})$. $0^\circ\mathrm{C}$ motsvarar $32^\circ\mathrm{F}$ och $100^\circ\mathrm{C}$ motsvarar $212^\circ\mathrm{F}$. Man kan använda formeln $^\circ\mathrm{C} = (^\circ\mathrm{F} - 32) \times {}^5\!/_9$ för att omvandla från $^\circ\mathrm{F}$ till $^\circ\mathrm{C}$. Skriv ett program som läser in en temperatur uttryckt i grader Fahrenheit $(^\circ\mathrm{F})$ och som översätter temperaturen till grader Celsius $(^\circ\mathrm{C})$.

2.4 I USA brukar en bils bensinförbrukning anges i *miles/gallon.* Skriv ett program som läser in en bensinförbrukning som är angiven på

detta sätt och översätter den till det för oss vanligare *liter/mil.*
Följande gäller:
$1\ mile = 1,609\ km$ och $1\ gallon = 3,785\ liter$.

2.5 Man kan beräkna avståndet mellan två punkter (x1, y1) och (x2, y2) i ett tvådimensionellt koordinatsystem med formeln

$$s = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Skriv ett program som läser in punkternas koordinater och som beräknar avståndet mellan punkterna.

2.6 Vid radioaktivt sönderfall kan man beräkna mängden kvarvarande radioaktivt material *n* efter en viss tid *t* med formeln

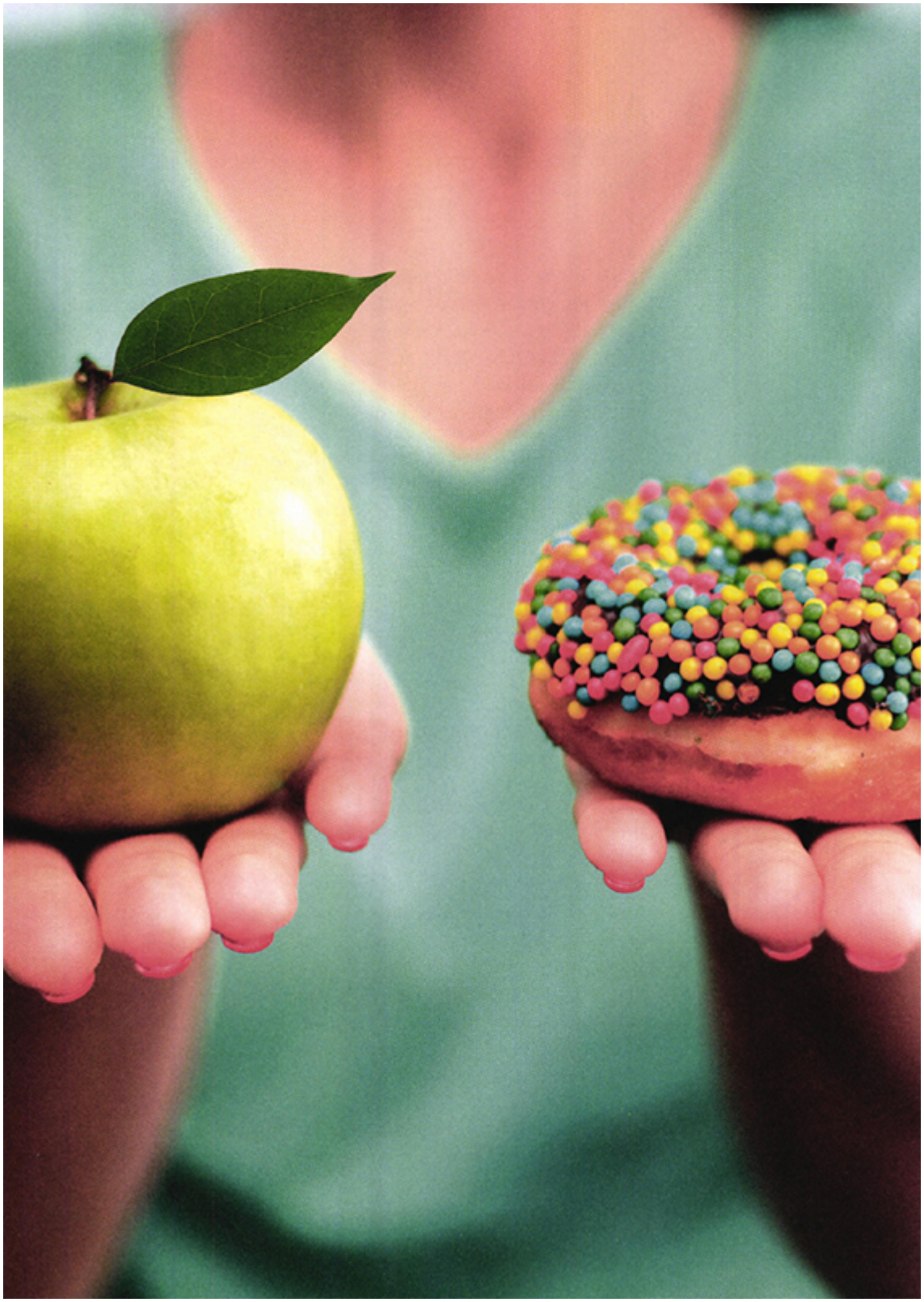
$$n = n_0 e^{-\lambda t}$$

där n_0 är mängden radioaktivt material vid tiden $t = 0$. Konstanten λ är en materialkonstant. Man brukar för det mesta ange halveringstiden (den tid det tar innan hälften av det radioaktiva materialet sönderfallit). Om halveringstiden betecknas med *T* kan man räkna ut att

$$\lambda = \frac{\ln 2}{T}$$

Halveringstiden för isotopen $^{14}\mathrm{C}$ är 5 730 år. Skriv ett program som skriver ut hur många procent av denna isotop som återstår efter *S* år. *S* ges som indata till programmet.

3 Att välja



Nu vet du hur man läser in data till ett program, gör enkla beräkningar och skriver ut resultatet. Men de program du sett hittills har alla en stor begränsning: De värden som räknas ut måste kunna uttryckas med hjälp av en formel av något slag. I praktiken är det mer komplicerat än så. Ett program måste t.ex. ibland välja olika alternativ vid en beräkning och ibland vill man kunna göra flera beräkningar i följd. Dessutom är det inte alltid så att ett program bara har till uppgift att räkna ut ett eller flera resultat. Det kan vara mycket mer komplicerat. (Tänk t.ex. på vad en texteditor eller ett mail-program gör.)

I mer generella program räcker det inte med att bara räkna upp ett antal satser som ska utföras i tur och ordning. Man måste ha bättre möjligheter att styra i vilken ordning de olika satserna ska utföras. Det visar sig att man i programmen måste kunna välja mellan olika alternativ. Man måste också kunna upprepa ett antal satser. I detta kapitel får du se hur man använder s.k. `if`-satser för att utföra val i ett program. I samband med detta diskuteras också *logiska uttryck,* uttryck som kan vara sanna eller falska. Hur man upprepar satser beskrivs i nästa kapitel.

3.1 `if`-satsen

När man ska välja mellan olika vägar i ett program använder man en `if`-sats. Denna sats kan se ut på lite olika sätt. Här kommer först ett exempel på den enklaste formen. Programmet läser in en temperatur och ber användaren sätta på värmen om temperaturen är för låg. Programmet avslutar med att skriva ut temperaturen.

[fullständigt program]

```
t = float(input('Temp?'))
```

```
if t < 18:
        print('Det är kallt')
        print('Sätt på värmen')
print(f'Det är {t:.1f} grader')
```

Om man kör programmet och ger en låg temperatur som indata ser det ut så här:

```
Temp? 15
Det är kallt
Sätt på värmen
Det är 15.0 grader
```

Men om man i stället ger en hög temperatur ser det ut på följande sätt:

```
Temp? 19.5
Det är 19.5 grader
```

I programmet är det raderna två, tre och fyra som ingår i `if`-satsen. En `if`-sats inleds alltid med ordet `if`. Detta är ett *reserverat ord (keyword),* liksom ordet `import` som vi använde i förra kapitlet Vi har därför skrivit `if` med fet stil.

Efter `if` ska det stå ett s.k. *logiskt uttryck,* ett uttryck som kan vara sant eller falskt. Här är det logiska uttrycket

$$t < 18$$

. Sedan följer ett kolon. Därefter ska det stå en eller flera rader som ska utföras om villkoret är sant. En sak som är mycket viktig är att dessa rader måste vara indragna en bit på raden. Man kan antingen skriva ett eller flera mellanslag eller ett tabulatortecken. När man drar in en rad på det sättet säger man att *indenterar* raden. I programmet ser vi att den tredje och fjärde raden är indenterade. Därför utförs dessa rader bara om villkoret är sant. Den första raden som följer efter `if`-satsen måste börja *exakt* lika långt in på raden som ordet `if`. Det gör den sista raden i vårt program. Denna rad ingår därför inte i `if`-satsen, utan kommer att alltid utföras.

Om man skulle glömma att indentera den första raden efter raden med ordet `if`, får man en felutskrift:

```
print('Det är kallt')
        ^
IndentationError: expected an indented block
```

I exemplet ovan körde vi programmet i *script mode.* Om man kör i *interactive mode* och ska skriva in en `if`-sats, måsta man skriva en tom rad (bara Enter) efter den sista raden som ska utföras villkorligt. Så här kan det se ut.

```
>>> t = 17
>>> if t < 18
... print('Det är kallt')
... print('Sätt på värmen')
...
```

```
Det är kallt
Sätt på värmen
>>> print(f'Det är {t:.1f} grader')
Det är 17.0 grader
>>>
```

Som vanligt använder vi röd text för att visa det som interpretatorn skriver. I detta exempel har vi använt tabulatortecken för att dra in raderna, men det går också att använda mellanslag. I *interactive mode* utför interpretatorn en sats i taget och skriver ut resultatet. Den väntar därför med att skriva något tills hela `if`-satsen är inskriven. Men den kan inte veta att `if`-satsen är slut förrän man markerar att det inte kommer några flera indenterade rader. Det gör man genom att skriva en tom rad. En sak att lägga märke till är att interpretatorn skriver ut . . . när den är inne i `if`-satsen i stället för `>>>` som den skriver när den är beredd att ta emot en ny sats.

Ytterligare ett exempel kommer här. Anta att man i en idrottstävling har uppmätt två tider och läst in dem till variablerna `t1` och `t2`. Man vill få reda på hur stor skillnaden är mellan tiderna och skriver därför följande programrader:

```
d = t1 - t2
if d < 0 :
        d = -d
print(f'Tidsskillnad: {d:.2f}')
```

Man beräknar skillnaden mellan tiderna och sparar den i en ny variabel med namnet `d`. Om `t2` är större än `t1` kommer `d` att

innehålla ett negativt värde, men man vill att tidsskillnaden ska anges som ett positivt värde i utskriften.

I det logiska uttryck som ska stå i en `if`-sats kan man använda följande vanliga jämförelseoperatorer:

```
< <= > >= == !=
```

De står för *mindre än, mindre än eller lika med, större än, större än eller lika med, lika med* respektive *icke lika med.* Observera att *lika med* skrivs med **två** likhetstecken `==`. Detta beror på att enkelt likhetstecken `=` används för att beteckna tilldelning. Man kan ha mer komplicerade logiska uttryck. Detta kommer att diskuteras i avsnitt 3.3.

Uppgift 3.1

Skriv ett program som beräknar kostnaden för att ringa med en mobiltelefon med ett enkelt abonnemang. Programmet ska läsa in hur många minuter man ringer per månad och hur mycket det kostar per minut. Man får 10 % rabatt om man ringer för minst 300 kr.

I den enkla version av `if`-satsen som du sett hittills kan man välja mellan att utföra vissa satser eller att *inte* utföra dem. Men det är kanske vanligare att man har två alternativ och vill välja att *antingen*

det ena eller det andra. Då har man nytta av en `if`-sats med en `else`-del. En `else`-del inleds med det reserverade ordet `else` följt av ett kolon.

Här kommer ett exempel som räknar ut priset för en charterresa. Resebyrån har ett erbjudande där alla vuxna får 10 % av grundpriset i rabatt och alla barn under 12 år 50 %.

[fullständigt program]

```
pris = float(input('Grundpris?'))
ålder = int(input('Ålder?'))
if ålder < 12:
        pris = pris * 0.5
else :
        pris = pris * 0.5
print(f'Pris: {pris:.2f} kr')
```

Här hade vi ett val mellan två alternativ: barn eller vuxen, men ibland kan man ha flera alternativ att välja bland. Då kan man använda sig av en eller flera `elif`-delar. Hur det ser ut framgår bäst av ett exempel. Anta att resebyrån utökar sitt erbjudande genom att ge pensionärer som är 65 år eller äldre 25 % rabatt. Vi kan då utöka vårt program med en `elif`-del:

[fullständigt program]

```
pris = float(input('Grundpris '))
ålder = int(input('Ålder? '))
if ålder < 12:
        pris = pris * 0.5
elif ålder >= 65:
        pris = pris * 0.75
else :
```

$$\text{pris} = \text{pris} * 0.9$$

```
print(f'Pris: {pris:.2f} kr')
```

51

Efter ordet `elif` ska det, precis som efter ordet `if`, finnas ett logiskt uttryck och ett kolon. Om villkoret är sant kommer satserna i `elif`-delen att utföras. Man kan ha flera `elif`-delar i en `if`-sats, men observera att det är den första av dem som är sann som kommer att väljas. Har man en `else`-del måste den komma allra sist. Den väljs om inga av de logiska uttrycken i `if`-delen eller `elif`-delarna är sanna.

`if`-sats, olika former

Tabellbeskrivning

Tabellen har 3 rader och 2 kolumner. Sista raden sträcker sig över två kolumner.

`if logiskt uttryck:` *`en eller flera satser`*	**`if logiskt uttryck:`** *en eller flera satser* **`else:`** *en eller flera satser*

```
if logiskt uttryck:
      en eller flera satser

elif logiskt uttryck:
      en eller flera satser

elif logiskt uttryck:
      en eller flera satser flera
      elif-delar
```

```
if logiskt uttryck:
      en eller flera satser

elif logiskt uttryck:
      en eller flera satser

elif logiskt uttryck:
      en eller flera satser flera
      elif-delar

else:
      en eller flera satser
```

Alla villkorliga satser måste dras in (indenteras) lika mycket.

Den första satsen efter `if`-satsen måste vara indenterad exakt lika mycket som ordet `if`.

Uppgift 3.2

På ett gym kan man antingen köpa ett årskort eller köpa en biljett vid varje besök. Skriv ett program som läser in priset för ett årskort, priset för en biljett samt antalet gånger man planerar att besöka gymmet under ett år. Därefter ska programmet ange om det lönar sig att köpa årskort eller inte.

Uppgift 3.3

På ett matteprov kunde man få högst 50 poäng. Gränsen för betyget E var 25 poäng och för betygen D, C, B och A 30, 35, 40 respektive 45 poäng. Skriv ett program som läser in poängen

för en elev och som visar vilket betyg hon eller han fick på provet.

3.2 Nästlade `if`-satser

En sats som ligger i en villkorlig del i en `if`-sats kan vara vilken sats som helst. Det kan till och med vara en ny `if`-sats. När man har sådan konstruktioner där `if`-satser ligger inne i andra `if`-satser säger man att man har nästlade if-satser. Vi visar ett exempel. Vi utökar programmet som läser in en temperatur.

[fullständigt program]

```
t = float(input('Temp? '))
if t < 18:
        print('Det är kallt')
        print('Sätt på värmen')
        if t < 12:
                print('Sätt på jackan')
else:
        print('Det är varmt')
        if t >= 22
                print('Stäng av värmen')
print(f'Det är {t:.1f} grader')
```

De rader som har markerats med rött är inre `if`-satser. När man har nästlade `if`-satser är det mycket viktigt att man är noggrann och indenterar raderna på rätt sätt. Lägg t.ex. märke till att raden som börjar med ordet `else` ligger på samma nivå som den första raden med `if`. Av detta ser man att `else`-delen hör ihop med den yttersta `if`-satsen. Hade man gjort fel och lagt raden med `else` på samma nivå som det `if` som är markerat med rött, hade programmet visserligen gått att köra, men det hade fungerat på fel sätt. Om man då t.ex. hade gett temperaturen 17 hade man fått både utskriften `'Det är kallt'` och `'Det är varmt'`.

Uppgift 3.4

Lägg till ett par nya alternativ, t.ex. `'Svinkallt'` och `'Ökenhett'`. Undersök också vad som händer om du indenterar på ett felaktigt sätt.

3.3 Logiska uttryck och typen `bool`

I avsnitt 2.4 fick du lära dig att varje numeriskt uttryck alltid har ett visst *värde* och en viss *typ.* Detta gäller inte bara numeriska uttryck. Alla

uttryck har ett visst värde och en viss typ. I en `if`-sats ska det som står på första raden vara ett uttryck som kan vara sant eller falskt. Ett sådant uttryck kallas ett *logiskt uttryck.* Logiska uttryck har alltid

typen `bool` (uppkallat efter den engelske matematikern George Boole). Detta är en speciell typ som används för att beskriva sanningsvärden. Ett uttryck av typen `bool` kan bara ha något av de två värdena `True` och `False`.

Om man använder de vanliga jämförelseoperatorerna `<, >` etc., bildas uttryck av typen `bool`. Här följer några exempel på jämförelseuttryck. I alla exemplen är operanderna av numerisk typ, men resultatet av hela uttrycket är av typen `bool`. Så här kan det t.ex. se ut om man kör i *interactive mode:*

```
>>> x = 5
>>> x > 3
True
>>> x <= 5
True
>>> x == 0
False
```

Jämförelseoperatorer

== lika med != icke lika med	< mindre än <= mindre än el. lika med	> större än >= större än el. lika med

Förutom jämförelseoperatorerna finns det tre operatorer som man kan använda för att bilda mer komplicerade logiska uttryck. Det är

operatorerna `and`, `or` och `not`. Dessa operatorer har operander av typen `bool` och ger (i normalfallet) som resultat ett värde av typen `bool`.

Logiska operatorer

Tabellbeskrivning

Tabellen har 2 rader och 3 kolumner. Sista raden sträcker sig över alla kolumner.

`A and B` sant om *både* `A` och `B` är sanna	`A or B` sant om *minst en av* `A` eller `B` är sann	`not A` sant om `A` är falsk

`not` har högre prioritet än `and` och `or`
`and` har högre prioritet än `or`
`and`, `or` och `not` har lägre prioritet än jämförelseoperatorerna		

Här kommer några exempel:

```
a > b and a < c
```

Detta uttryck är sant om `a` är större än `b` *och* `a` är mindre än `c`.

```
n == 0 or n == 100
```

Detta uttryck är sant om `n` är lika med 0 *eller* `n` är lika med 100.

```
not (n == 0 or n == 100)
```

Detta uttryck är sant om `n` *inte* är lika med 0 eller 100. Här ser du att man kan använda parenteser för att bilda deluttryck. I detta exempel beräknas deluttrycket innanför parenteserna först och sedan negeras resultatet eftersom operatorn `not` står framför.

De logiska operatorerna har lägre prioritet än de aritmetiska operatorerna `+`, `-`, `*` och `/` och jämförelseoperatorerna `<`, `>` etc. Detta innebär att uttryck som

```
i + j > k * l and m == n
```

tolkas som

```
(i + j) > (k * 1) and (m == n)
```

Operatorerna `and` och `or` beräknas från vänster till höger och beräkningen avslutas så fort det är möjligt. Det innebär att om den vänstra operanden till operatorn `and` blir falsk, beräknas aldrig den högra och om den vänstra operanden till operatorn `or` blir sann, beräknas aldrig den högra. Detta är ibland viktigt, som t.ex. i följande uttryck där man vill undvika att göra divisionen om variabeln `k` är lika med noll.

$$k! = 0 \texttt{ and } {}^{n}/_{k} > 10$$

En sak som är speciell i Python (det brukar inte vara tillåtet i andra programspråk) är att man kan koppla ihop jämförelseoperatorer. För att t.ex. undersöka om talet `x` ligger i intervallet 1 till 5 kan man skriva

$$1 <= x <= 5$$

Detta tolkas som om man hade skrivit

$$1 <= x \texttt{ and } x <= 5$$

Som exempel på logiska uttryck visas ett program som undersöker om ett visst årtal är ett skottår eller inte. Det årtal som ska undersökas läses in till programmet. Det kan t.ex. se ut så här när man kör programmet.

```
Skriv ett årtal: 2020
Skottår
```

Följande regel gäller för skottår (i Sverige fr.o.m. år 1754): *Skottår är sådana år som är jämnt delbara med 4, med undantag för sådana år som är jämnt delbara med 100. Dock är sådana år som är jämnt delbara med 400 skottår.* Programmet har följande utseende:

[fullständigt program]

```
år = int(input('Skriv ett årtal:'))
if (år % 4 == 0 and år % 100 != 0) or år % 400 == 0:
        print('Skottår')
else :
        print('Inte skottår')
```

Det är förstås det logiska uttrycket i `if`-satsen som är det intressanta i detta program. Där används operatorn `%` som ju ger resten vid division. Uttrycket `år % 4` ger t.ex. resten när man dividerar årtalet med 4. Om årtalet är jämnt delbart med 4 blir resten lika med noll. Om uttrycket `år % 100` inte är lika med noll, är årtalet inte jämnt delbart med 100. Deluttrycket

```
(år % 4 == 0 and år % 100 != 0)
```

är alltså sant för sådana årtal som är jämnt delbara med 4, men inte med 100. För att ett år ska vara skottår måste antingen detta deluttryck vara sant eller så måste året vara jämnt delbart med 400. Det är därför det står `or år % 400 == 0` sist i det logiska uttrycket.

Precis som man kan skapa variabler av de numeriska typerna `int` och `float` kan man också skapa variabler av typen `bool`. Det går t.ex. att göra skriva

```
billig = False
barn = True
```

Man kan ha ett uttryck av typen `bool` till höger om likhetstecknet:

$$\text{billing} = \text{pris} <= 100$$

$$\text{barn} = \text{ålder} < 12$$

Variabler av typen `bool` kan användas i logiska uttryck som t.ex.

```
if barn:
        pris = 3145
```

(Många nybörjare brukar i uttryck som dessa skriva `if barn == True`, men detta är helt onödigt. Uttrycket efter `if` ska ha typen `bool`. Variabeln `barn` har denna typ och innehåller alltså antingen värdet `True` eller värdet `False`.)

3.4 Satser och radstruktur

Det finns två kategorier av satser i Python: *enkla satser* och *sammansatta satser.* Några slag av enkla satser som du redan sett är tilldelningssatser, uttryckssatser och `import`-satser. (Funktionsanrop räknas som uttryckssatser.) Ett annat exempel är satsen `pass`, som inget gör.

```
y = x * x
print(f'Resultatet är {y:.2f}')
import random
pass
```

`if`-satsen är ett exempel på en sammansatt sats. Andra vanliga sammansatta satser är `while`-satsen och `for`-satsen som vi kommer att diskutera i nästa kapitel. Det som skiljer en sammansatt sats från en enkel är att den sammansatta satsen inne i sig kan innehålla andra satser. En `if`-sats kan ju innehålla flera satser som ska utföras om ett villkor är sant. Även efter orden `else` och `elif` kan det finnas inre satser.

I de flesta vanliga programspråk, t.ex. Java och C, får man skriva i fritt format, vilket betyder att man kan redigera sina program som

man själv vill och bestämma hur man delar upp satserna på olika rader. För att hålla reda på när satser är avslutade och vilka satser som ingår som delar i sammansatta satser använder man sig där av semikolon och speciella start- och stopptecken, ofta `{` och `}`.

I Python använder man inga start- och stopptecken. I stället är det radstrukturen som avgör hur programmet är uppbyggt. Därför är det viktigt att man har klart för sig hur man ska skriva. Grundregeln är att en enkel sats alltid ska skrivas på en enda rad. Men det är faktiskt tillåtet att skriva flera enkla satser på samma rad med semikolon mellan.

```
a = 5;  b = 6;  c = 7
```

Om man har rader som blir långa är det tillåtet att fortsätta en sats på nästa rad. Man måste då skriva tecknet `\` sist på den första raden. Det markerar att satsen fortsätter på nästa rad. Här kommer ett exempel:

```
summa = random.randint(1, 6) + \
        random.randint(1,6)
```

En sammansatt sats skrivs på flera rader och de satser som ingår som delar i den sammansatta satsen ska dras in på raden (indenteras). Hur det ska se ut beror på vilken sats det är. Vi har sett hur `if`-saten är uppbyggd. Den första raden i satsen ska börja med ordet `if` och sedan ska det komma ett logiskt villkor. Ibland kan

detta villkor vara lite långt och komplicerat. Då är det även här tillåtet att använda tecknet `\` för att fortsätta på en extra rad. I programmet som undersökte om ett år var ett skottår hade vi t.ex. kunna skriva

```
if (år % 4 == 0 and år % 100 != 0) or \
        år % 400 == 0:
```

Detsamma gäller förstås om man har långa villkor efter `elif`.

När det gäller indentering i sammansatta satser räknar Python-interpretatorn hur många mellanslag man drar in texten. Tab-tecknet omvandlas automatiskt till ett eller flera mellanslag. När man sedan ska gå tillbaka till den yttre nivån måste man ta bort exakt lika många mellanslag.

Radstruktur

Enkla satser ska normalt skrivas på en rad, men kan stå på samma rad med semikolon mellan resp. sats.

Sammansatta satser omfattar flera rader. De inre satserna ska indenteras. Om en rad blir för lång kan man forstätta satsen på nästa rad om man skriver tecknet `\` sist på raden.

Undantag: Uttryck inom parenteser, dvs. `()`, `[]` eller `{}`, får fortsätta på nästa rad utan att man skriver tecknet `\` sist på raden.

Uppgift 3.5

I Postnords anvisningar finns följande föreskrifter för hur stora och små vanliga brev får vara:

Maximimått: Längd 600 mm, Tjocklek 100 mm, Bredd+längd+tjocklek högst 900 mm.

Minimimått: Längd 140 mm, Bredd 90 mm.

Skriv ett program som läser in ett brevs längd, bredd och tjocklek och som undersöker om brevet har tillåtna mått eller inte. Använd tecknet `\` i `if`-satsen för att dela upp de logiska uttrycket så att det blir mer lättläst.

3.5 Villkorsuttryck

[överkurs]

Det finns en konstruktion som man kan klara sig utan, men som man i alla fall bör känna igen. Det är s.k. *villkorsuttryck.* Anta t.ex. att vi ska undersöka vilken av variablerna `x` och `y` som är störst och tilldela det största värdet till variabeln `z`. Detta görs naturligtvis enkelt med hjälp av en `if`-sats:

```
if x > y:
        z = x
else:
```

```
        z = y
```

men man kan också använda ett villkorsuttryck:

```
z = x if x > y else y
```

De olika delarna kommer i en annan ordning än i en `if`-sats. Om det logiska uttrycket som står efter ordet `if` är sant, beräknas det uttryck som står först, framför ordet `if` (`x` i detta exempel) och resultatet av hela villkorsuttrycket blir lika med detta värde. Om logiska uttrycket i stället blir falskt, beräknas det uttryck som står efter `else` (`y` i detta exempel) och resultatet av villkorsuttrycket blir lika med detta värde.

Uppgift 3.6

[överkurs]

Kör interpretatorn i *interactive mode* och testa villkorsoperatorn. Lägg in några aritmetiska uttryck i stället för bara `x` och `y` som i exemplet ovan.

3.6 Sammanfattning

Efter att ha läst detta kapitel bör du:

- veta hur man använder `if`-satser för att välja olika alternativ,

- känna till de olika sätt en `if`-sats kan se ut på och veta hur man ska indentera (dra in) programtexten,
- veta hur man bildar nästlade `if`-satser,
- veta vad ett logiskt uttryck är,
- känna till de operatorer som kan ingå i logiska uttryck,

- känna till typen `bool` och veta hur man skapar variabler av denna typ och tilldelar dem värden,
- känna till skillnaden mellan enkla och sammansatta satser,
- känna till reglerna för hur man skriver satser på olika rader och hur man kan använda tecknet `\` för att förlänga en rad.

3.7 Övningar

3.1 En operatör för mobiltelefoni erbjuder tre olika abonnemang: *Kontant, Normal* och *Plus.* Om man jämför villkoren för abonnemangen visar det sig att abonnemanget *Kontant* är billigast om man ringer högst 33 minuter per månad, *Normal* lönar sig bäst om man ringer mellan 33 och 66 minuter per månad och *Plus* är mest förmånligt om man ringer ännu mer. Skriv ett program som läser in antalet minuter man uppskattar att man kommer att ringa per månad. Programmet ska tala om vilket abonnemang man bör välja.

3.2 I uppgift 2.7 på sidan 38 skulle du skriva ett program som läste in en cirkels radie och beräknade cirkelns omkrets och area. Utöka programmet med en `if`-sats som kontrollerar att den inlästa radien är större än 0. Skulle så inte vara fallet, ska programmet visa en felutskrift i stället för att göra beräkningar.

3.3 I en triangel kan man beteckna sidorna med *a, b* och *c.* Om man känner till längderna för sidorna *a* och *b* samt vinkeln α mellan dessa sidor så kan man räkna ut längden av sidan *c* med formeln $\mathrm{c} = \sqrt{\mathrm{a}^2 + \mathrm{b}^2 - 2\mathrm{abcos}\alpha}$. Skriv ett program som läser in längderna på två sidor i en triangel och vinkeln mellan sidorna. Programmet ska avgöra om triangeln är *liksidig* (alla sidor lika), *likbent* (två sidor lika) eller *oliksidig* (inga sidor lika). Tips: När man jämför två variabler av typen `float` för att se om de är lika bör man inte använda operatorn == direkt eftersom talen lagras i en approximativ form. Beräkna i stället skillnaden mellan talen och undersök om absolutvärdet (se faktarutan på sidan 37) av denna skillnad är mindre än ett litet tal, t.ex. 10^{-10}.

4 Att upprepa

I ett program måste man kunna välja mellan olika alternativ och kunna göra upprepningar. I förra kapitlet fick du lära dig hur man väljer. I detta kapitel får du veta hur man gör upprepningar med s.k. *repetitionssatser.* Du kommer att lära dig `while`-satsen och `for`-satsen.

4.1 `while`-satsen

Om man vill upprepa en eller flera satser är det enklast att använda en `while`-sats. Den enklaste versionen av denna har formen:

```
while logiskt uttryck:
        en eller flera indragna satser
```

Som du ser har den samma form som den enklaste `if`-satsen, men det står `while` i stället för `if`. De satser som står med början på den andra raden måste vara indragna, indenterade, precis som de villkorliga satserna i en `if`-sats. Dessa satser kommer att utföras om och om igen (eller ev. ingen gång alls). Man säger att ett visst antal *varv* kommer att utföras. Det logiska uttrycket efter `while` bestämmer hur många varv det blir. Uttrycket beräknas *varje gång ett nytt varv ska påbörjas.* Om uttrycket då är sant utförs det nya varvet, dvs. satsen exekveras en gång till. Om uttrycket däremot är falskt utförs inga fler varv. Programmet fortsätter då med nästa sats efter `while`-satsen.

Här kommer ett enkelt exempel.

[fullständigt program]

```
print(' [', end='')
k = 0
while k < 6:
        print(f'{k:2}', end='')
        k = k + 2
print(' ]')
```

När man kör detta program får man utskriften

```
[ 0 2 4 ]
```

Programmet börjar med att skriva ut texten `'['`. Argumentet `end=''` betyder att inget radbyte ska läggas in efter denna utskrift. (Se faktarutan på sidan 29.) Det innebär att nästa gång `print` anropas så kommer utskriften att fortsätta på samma rad.

`while`-satsen innehåller två satser som ska utföras gång på gång så länge det logiska uttrycket

$$k < 6$$

är sant.Variabeln `k` används som räknare. Från början tilldelas den värdet 0. När programmet kommer fram till `while`-satsen första gången blir därför uttrycket är

$$k < 6$$

sant. Därför utförs ett första varv. På detta varv skriver man först ut variabeln `k`. Värdet 0 skrivs alltså ut. I anropet av `print` används

argumentet `end=''` för att nästa utskrift ska fortsätta på samma rad. Den andra satsen på varvet ökar `k` med 2, så att `k` får värdet 2.

Efter det första varvet beräknas det logiska uttrycket

$$\mathrm{k} < 6$$

på nytt. Detta uttryck är fortfarande sant. Då kommer värdet 2 att skrivas ut och `k` ökas sedan till 4.

Det logiska uttrycket

$$\mathrm{k} < 6$$

beräknas därefter på nytt. Det är fortfarande sant, varför ett tredje varv utförs. På detta varv skrivs värdet 4 ut och `k` ökas till 6. Efter det tredje varvet blir därför uttrycket

$$\mathrm{k} < 6$$

falskt och inget fjärde varv kommer att utföras.

Den sista raden i programmet ligger utanför `while`-satsen, eftersom den är indenterad till samma nivå som ordet `while`. Då skrivs den avslutande texten `']'` ut.

Eftersom det inte finns någon slutmarkör som visar vilken som är den sista satsen på ett varv i en `while`-sats, är det mycket viktigt att man drar in texten på rätt sätt.

Uppgift 4.1

Kör programmet ovan och undersök vad som händer om man ger andra startvärden till variabeln `k`. Se sedan vad som händer om du drar in den sista raden så att den hamnar på samma nivå som raden ovan. Vad händer om du i stället drar ut den näst sista raden så att den hamnar under ordet `while`.

I nästa exempel visas ett program vars uppgift är att läsa in ett heltal `n` och därefter beräkna summan $1 + 2 + 3 \ldots + n$. Om talet `n` t.ex. är 5, ska programmet räkna ut summan $1 + 2 + 3 + 4 + 5$ som ju blir 15.

[fullständigt program]

```
# Beräkning av summan 1+2+3...+n
n = int(input('n? '))
summa = 0
k = 1
while k <= n:
        summa = summa + k # öka summan med k
        k = k + 1 # öka k med 1
print('Summan blir', summa)
```

Uppgift 4.2

Skriv ett program som läser in ett heltal *n* och som beräknar summan, dvs. $1 + 4 + 9 + 16 + \ldots n^2$. Tips: k^2 är samma sak som *k*k.*

Uppgift 4.3

[överkurs]

Skriv ett proqram som läser in ett heltal *n* och som beräknar den sk. harmoniska serien $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$.

Det är inte alltid man har en räknare som bestämmer hur många varv som ska utföras i en `while`-sats. Här följer ett lite annorlunda exempel. Anta att en viss typ av vattenalger under gynnsamma förhållanden förökar sig så fort att de varje dygn fördubblar den area de täcker. Anta vidare att man i en liten sjö råkat få in sådana alger och att de efter en dag täcker en area av $1\ \mathrm{dm}^2$, dvs. $0,01\ \mathrm{m}^2$. Sjön är ungefär 100 gånger 100 m stor, dvs. dess area är $10000\ \mathrm{m}^2$. Följande program beräknar hur lång tid det tar innan hela sjön blir täckt med alger.

[fullständigt program]

```
# Utbredning av alger
totalArea = 10000
area = 0.01
dygn = 1
while area < totalArea:
        area = area * 2
        dygn = dygn + 1
print(f'Sjön blir täckt efter {dygn} dygn')
```

Variabeln `totalArea` innehåller sjöns totala `area`. Variabeln area används för att hålla reda på hur stor area algerna täcker. Den initieras till det värde som gäller efter ett dygn, dvs. 0,01. Variabeln `dygn`

används för att räkna antalet dygn som behövs. Den initieras till 1, vilket betyder att det från början gått ett dygn. Varje varv i `while`-satsen motsvarar ytterligare ett dygn. Under varje varv fördubblas den täckta arean och antalet dygn räknas upp med 1. Villkoret `area < totalArea` är sant så länge som sjön ännu inte är helt täckt. Eftersom variabeln `area` ökar under varje varv kommer så småningom villkoret att blir falskt. Då utförs satsen efter `while`-satsen. Där skrivs det ut hur många dygn som behövdes. Utskriften blir:

```
Sjön blir täckt efter 21 dygn
```

Uppgift 4.4

Anta att en boll som släpps ovanför ett golv vid varje studs förlorar 30 % av sin höjd. Skriv ett program som beräknar hur många gånger en sådan boll studsar innan den blir stilla. (Med stilla menar vi att den inte studsar högre än 1 cm.) Som indata till programmet ska användaren ange den höjd, mätt i meter, som bollen släpps ifrån.

4.2 `break`- och `continue`-satsen

Som du lärde dig i förra avsnittet testas det logiska uttrycket i en `while`-sats före varje varv och om uttrycket är sant utförs ytterligare ett varv. Detta betyder att ett *helt* antal varv kommer att utföras. 1

vissa situationer kan det emellertid vara bekvämt att avbryta exekveringen *mitt* i ett varv. Då kan man använda en `break`-sats. En sådan är enkel. Man skriver bara `break`. När en `break`-sats exekveras avbryts `while`-satsen och programmet hoppar till den första sats som kommer efter `while`-satsen.

Studera som exempel följande programrader

```
k = 0
while k < 6:
        print(f'{k:2}', end='')
        k = k + 2
```

I dessa rader löpen `while`-satsen tre varv och man får utskriften

```
0  2  4
```

Anta nu att vi skriver om dessa satser på följande sätt:

```
k = 0
while True:
        print(f'{k:2}', end='')
        if k >= 6:
                break
        k = k + 2
```

Vi har nu lagt in en `break`-sats. Denna ligger i en `if`-sats och utförs bara om `k >= 6` (dvs. om `k` *inte* är mindre än 6). Ordet `True` betecknar ett logiskt värde som alltid är sant. Det betyder att det logiska uttrycket som testas i början av varje varv alltid är sant. Hade det inte funnits en `break`-sats hade därför ett oändligt antal varv utförts i `while`-satsen.

I början på det första varvet har `k` värdet 0, på det andra varvet värdet 2, på det tredje varvet värdet 4 och i början på det fjärde varvet värdet 6. Detta betyder att `break`-satsen kommer att utföras på det fjärde varvet som alltså avbryts mitt i. Vi får därför utskriften

```
0 2 4 6
```

En sats som liknar `break`-satsen är `continue`-satsen, men till skillnad från `break`-satsen avbryter den inte *hela* repetitionssatsen. Den avbryter i stället bara det aktuella *varvet.* Följande programrader skriver t.ex. inte ut udda värden på `k`. Talen 2, 4 och 6 skrivs ut.

```
k = 0
while k < 6:
        k = k + 1
        if k % 2 != 0:
                        continue
        print(f'{k:2}', end='')
```

För det mesta ska man försöka undvika att använda `break`- och `continue`-satser eftersom dessa kan göra programmet svårare att förstå. Men det finns ett tillfälle när det är naturligt att använda en `break`-sats. Det är när man läser indata och ska testa om användaren vill avbryta beräkningarna. Alla program du sett hittills i boken har bara gjort *en* enda beräkning. När resultatet av denna visats har programmet avslutats. Du ska nu få lära dig hur man

utformar sina program så att de kan utföra ett *godtyckligt* antal beräkningar. Med detta menas att de inte ska avslutas, utan när en beräkning gjorts ska användaren ha möjlighet att ge indata till en ny beräkning. Detta ska fortsätta tills användaren anger att han eller hon vill avbryta beräkningarna.

Som exempel visas en ny version av programmet på sidan 63. Den nya versionen klarar av att göra upprepade beräkningar av summan $1 + 2 + 3 \ldots + n$ för olika värden på `n`.

[fullständigt program]

```
# Beräkning av summan 1+2+3...+n
while True:
        n = int(input('n?, skriv ett tal <= 0 för
att sluta '))
        if n<=0:
                break
        summa = 0
        k = 1
        while k<=n:
                summa = summa + k # öka summan med
k
                k = k+1 # öka k med 1
        print('Summan blir', summa)
```

Som du ser har beräkningen lagts in i en `while`-sats. Varje varv i denna motsvarar en ny beräkning. I början av varje läser man in ett nytt värde till variabeln `n`. Om användaren ger ett värde som är

mindre än eller lika med 0 utförs `break`-satsen, vilket innebär att `while`-satsen avbryts.

break-sats och continue-sats

Får placeras bland de satser som utförs på varje varv i en repetitionssats. `break`-satsen avbryter omedelbart repetitionssatsen.

Hopp sker till första sats efter repetitionssatsen.

`continue`-satsen avbryter omedelbart det aktuella varvet.

Hopp sker till repetitionssatsens början. Ett ev. nytt varv påbörjas då.

Uppgift 4.5

Skriv en ny version av programmet från uppgift 4.4 så att man kan göra upprepade beräkningar där man släpper bollen från olika höjder. Programmet ska avslutas när användaren skriver in ett negativt tal.

4.3 `for`-satsen

Den enklaste satsen för att göra upprepningar är `while`-satsen, men det finns en annan sats som också används ofta, nämligen `for`-satsen. Man

brukar använda en `for`-sats när man har en räknare som ska räknas upp eller ner för varje varv. Som första exempel ska du få se en ny version av följande programrader från sidan 61:

```
print('[', end='')
k = 0
while k < 6:
        print(f'{k:2}', end='')
        k = k + 2
print(']')
```

Den nya versionen fungerar på exakt samma sätt och den kommer också att ge utskriften

```
[ 0 2 4 ]
```

Den använder emellertid en `for`-sats i stället för en `while`-sats och ser ut på följande sätt:

```
print('[', end='')
for k in range(0, 6, 2):
        print(f ' {k: 2}', end='')
print(f' ]')
```

Om du jämför de två versionerna ser du att det efter ordet `for` kommer namnet på en variabel och sedan ordet `in`. Variabeln

fungerar som en räknare och kommer att få olika värden på varje varv i repetitionen. I detta exempel har vi gett variabeln namnet `k`.

Efter ordet `in` ska man tala om vilka värden räknaren ska ha under de olika varven. Det vanligaste är att man vill att den ska löpa igenom ett visst intervall. Då kan man använda sig av funktionen `range`. Den har tre argument som alla måste vara heltal. Det första anger starten på intervallet; här har vi angivit 0. Det andra argumentet anger intervallets övre gräns. Men man får se upp här. Det värde man ska ge är det första tal som är för stort och alltså *inte* ingår i intervallet. Här står det 6, vilket innebär att intervallets övre gräns är 5. Det tredje argumentet anger hur mycket räknaren ska räknas upp på varje varv, det s.k. *steget.* Här har vi gett värdet 2. Uttrycket `range(0, 6, 2)` betyder alltså att räknaren `k` ska ha värdet 0 på första varvet, 2, på det andra och 4 på det tredje. Något fjärde varv blir det inte eftersom `k` då skulle fått värdet 6 och detta ligger utanför intervallet.

Det är tillåtet att ha ange ett negativt steg. Följande programrader

---


```
print(' [', end='')
for k in range(10, 0, −1):
        print(f'{k:2}', end='')
        print(' ]')
```

ger t.ex. utskriften

```
[10 9 8 7 6 5 4 3 2 1]
```

Lägg märke till att det första argumentet till `range` då ska vara större än det andra. Lägg också märke till att det sista värdet som räknaren får blir *större* än det andra argumentet, inte mindre som när man räknar uppåt. I detta exempel är det andra argumentet lika med 0, men räknaren sista värde blir 1.

for-sats med range

```
for i in range(forst, sist, steg):
    en eller flera indragna satser
```

`forst, sist` och `steg` måste vara heltal.

Om man utelämnar argumentet `steg`, blir `steg` automatiskt `1`.

Om man bara ger *ett* argument tolkas det som argumentet `sist`.

Argumenten `forst` och `steg` blir då automatiskt `0` respektive 1.

Variabeln `i` får värdet `forst` på första varvet, `forst+steg` på det andra, `forst+steg+steg` på det tredje osv.

Om `steg > 0` fortsätter repetitionen bara så länge som `i < sist`.

Om `steg < 0` fortsätter repetitionen bara så länge som `i > sist`.

Efter sista varvet har variabeln `i` det värde som den hade på sista varvet.

I nästa exempel visas ett program som beräknar värdet av

$$x^n$$

. Du får anta att $n \geq 0$. (Naturligtvis hade vi kunna använda operatorn ** i stället, men avsikten med detta program är att visa hur `for`-satsen fungerar.)

[fullständigt program]

```
# Beräkning av x upphöjt till n
x = float(input('x? '))
n = int(input('n? '))
r = 1
for i in range(n):
        r = r * x
print('Resultat:', r)
```


Lägg märke till att i detta program har `range` bara ett argument. Det står `range(n)`. Det betyder samma sak som om man skulle ha skrivit `range(0, n, 1)`. Det är underförstått att intervallets första värde är 0 och att räknaren ska räknas upp med 1 på varje varv. I programmet kommer därför `for`-satsen att snurra `n` st varv och räknaren löper från 0 till

$$n - 1$$

.

Uppgift 4.6

Skriv en ny variant av programmet i uppgift 4.2. I den nya varianten ska du använda en `for`-sats i stället för en `while`-sats.

Det sista exemplet i detta avsnitt visar ett program som beräknar hur mycket pengar man får på ett bankkonto om man sätter in 1000 kr och låter pengarna stå inne utan att röra dem under ett visst antal år. När man kör programmet frågar det efter räntesatsen, som anges i procent, och hur många år pengarna ska stå inne. Som resultat skriver programmet ut en tabell där man kan se hur mycket kapitalet har växt efter varje år. Det kan t.ex. se ut så här när man kör programmet:

```
Räntesats? 1.5
Antal år? 5
        1 1015
        2 1030
        3 1046
        4 1061
        5 1077
```

Här följer programmet:

[fullständigt program]

```
# Ränteberäkning
ränta = float(input('Räntesats?'))
n = int(input('Antal år?'))
kapital = 1000
for år in range(1, n+1, 1):
```

```
        kapital = kapital + kapital * 0.01 * ränta
        print(f'{år:3}{kapital: 6.0f}')
```

I `for`-satsen utförs ett varv för varje år. För varje år adderas räntan till kapitalet. (Eftersom räntan anges i procent måste man multiplicera med 0,01.) På varje varv skriver man sedan ut årets nummer och det kapital man har efter det året.

Uppgift 4.7

Skriv ett program som visar en tabell med värden för uttrycket $2x^2 - 5x + 1$. Låt programmet skriva ut värdet av uttrycket för *x*-värdena -10, -9, -8, -7, osv. upp till 10.

Uppgift 4.8

Ändra i programmet från övning 4.7 så att värdet av uttrycket i stället skrivs ut för alla *x* i intervallet -1 till $+1$ med steget 0,1, dvs. för $-1, 0$, $-0, 9$, ... 0,9, 1,0. *Tips:* Dividera räknaren med 10.

4.4 Nästlade repetitionssatser

De satser som står inne i en repetitionssats får vara vilken sorts satser som helst. Det betyder att det kan finns en repetitionssats inne i en annan repetitionssats. Sådana programkonstruktioner är vanliga. Du ska få se ett enkelt exempel. Målet är att konstruera ett program som skriver ut ett antal plustecken. På första raden skrivs ett plustecken, på andra två plustecken osv. Programmet börjar med att fråga hur många rader som ska skrivas ut. Det kan t.ex. se ut så här:

```
Antal rader? 5
+
++
+++
++++
+++++
```

Här följer programmet:

[fullständigt program]

```
# Plustecken
n = int(input('Antal rader?'))
for i in range, n+1) :
        for j in range (1, i+1) :
                print('+', end='')
        print() # avslutar raden
```

Det finns två `for`-satser. Den yttre löper `n` st varv. Varje varv motsvarar en rad i utskriften. En variabel med namnet `i` används som räknare. På första varvet har `i` värdet 1, på andra varvet värdet 2 osv.

På *varje* varv i den yttre `for`-satsen utförs följande tre rader:

```
for j in range(1, i+1) :
        print('+', end='')
print() # avslutar raden
```

De första två raderna är en inre `for`-sats. I denna används en räknare `j` som löper mellan 1 och `i`. På första varvet i den yttre `for`-satsen har `i` som du såg värdet 1. Det betyder att variabeln `j` då kommer att löpa mellan 1 och 1. Den inre `for`-satsen utförs med andra ord bara ett varv. Då anropas `print` en gång och tecknet `+` kommer att skrivas ut. Lägg märke till att det står `end=''`, vilket innebär att utskriftsraden inte avslutas. Därefter anropas `print` på sista raden. Den har inget argument och kommer bara att skriva ut ett radslutstecken så att raden avslutas. Efter första varvet i den yttre `for`-satsen har alltså en rad med ett plustecken skrivits ut.

På det andra varvet i den yttre `for`-satsen har `i` värdet 2, vilket innebär att `j` kommer att löpa mellan 1 och 2. Den inre `for`-satsen löper alltså två varv. Det betyder att tecknet `+` kommer att skrivas ut *två* gånger efter varandra på samma rad. Den andra raden avslutas när den sista `print`-satsen utförs.

På motsvarande sätt kommer tre plustecken att skrivas ut på det tredje varvet i den yttre `for`-satsen, fyra på det fjärde varvet osv.

Uppgift 4.9

Ändra i programmet så att det i stället skrivs n st plustecken på första raden, $n - 1$ plustecken på andra raden osv. tills det på sista raden skrivs ett plustecken. Tips: Använd ett negativt steg i den yttre `for`-satsen.

4.5 `else`-del

[överkurs]

En sak som skiljer Python från andra vanliga programspråk är att `while`-satser och `for`-satser märkligt nog kan ha en `else`-del. Det är en

---

`else`-del där det inte finns någon mostsvarande `if`-del. Så här ser strukturen ut i en `while`-sats

```
while logiskt uttryck:
        en eller flera indragna satser
else :
        en eller flera indragna satser
```

och så här i en `for`-sats:

```
for räknare in värden:
```

```
        en eller flera indragna satser
else :
        en eller flera indragna satser
```

Det fungerar på följande sätt: Om repetitionssatsen har avslutas på "normalt" vis genom att det logiska uttrycket i en `while`-sats blivit falskt eller genom att räknaren löpt igenom alla värdena i en `for`-sats, så kommer de satser som ligger i `else`-delen att utföras. Om däremot repetitionssatsen har avslutas på en "onormalt" sätt genom att man exekverat en `break`-sats, så kommer inte `else`-delen att utföras.

Vi ska demonstrera hur man kan använda en `else`-del genom att skriva ett program där man ska gissa ett tal. Programmet tar fram ett slumptal och den som kör programmet får ett antal försök på sig för att gissa talet. Gissar man fel anger programmet om det tal man gissat är för litet eller för stort. Det kan t.ex. se ut så här när man kör programmet. Först frågar programmet vilket som ska vara det största möjliga talet. Det tar sedan fram ett slumptal i intervallet 1 till detta tal. Programmet frågar sedan hur många försök man har på sig.

```
Det största möjliga talet? 12
Hur många försök får man? 3
Gissa talet? 6
För litet
Gissa talet? 9
För litet
Gissa talet? 10
För litet
Inga fler försök
Talet var 12
```

I detta exempel lyckades man inte gissa talet, men om man lyckas kan det se ut så här:

```
Det största möjliga talet? 12
Hur många försök får man? 3
Gissa talet? 6
För stort
Gissa talet? 3
För litet
Gissa talet? 4
Rätt gissat
```

Här kommer hela programmet. Lägg märke till att den avslutande `else`-delen utförs bara om `for`-satsen har löpt igenom alla varven. Detta sker om man har förbrukat alla försöken utan att kunna gissa talet. Om man däremot gissar rätt, avslutas `for`-satsen genom att satsen `break` utförs. Däremot utförs då inte satserna i `else`-delen.

[fullständigt program]

```
# Gissa talet
import random
största = int(input('Det största möjliga talet?
'))
försök = int(input('Hur många försök får man? '))
tal = random.randint(1, största)
for i in range(försök):
        gissning = int(input('Gissa talet? '))
        if gissning < tal:
```

```
            print('För litet')
        elif gissning > tal:
            print('För stort')
        else :
            print('Rätt gissat')
            break
else :
        print('Inga fler försök')
        print('Talet var', tal)
```

4.6 Sammanfattning

Efter att ha läst detta kapitel bör du:

- veta hur man använder `while`-satser för att göra upprepningar,
- veta hur `break`-satsen fungerar,
- veta hur `continue`-satsen fungerar,

- kunna skriva program som gör beräkningar ett godtyckligt antal gånger tills användaren vill sluta,
- kunna utnyttja `for`-satser för att göra upprepningar och veta hur funktionen `range` kan användas,
- förstå hur nästlade repetitionssatser fungerar,

- om du läst avsnitt 4.5, förstå hur `else`-delen i en repetitionssats fungerar.

4.7 Övningar

4.1 Skriv ett program som läser in ett godtyckligt antal positiva tal. När användaren skriver in ett negativt tal ska programmet skriva ut det största och det minsta av de positiva talen.

4.2 Skriv ett program som skriver ut en tabell för talen 1 till 12. På varje rad i tabellen ska talet visas liksom talet i kvadrat och talet i kubik.

4.3 På en mycket farlig arbetsplats erbjuds man följande lön: första dagen tjänar man ett öre och sedan fördubblas lönen varje dag. Skriv ett program som räknar ut hur länge man behöver arbeta för att tjäna ihop 10 miljoner kronor.

4.4 Skriv ett program som visar en multiplikationstabell enligt följande modell. Programmet ska utformas så att man läser in antalet rader som ska skrivas ut. *Tips:* Använd nästlade `for`-satser.

Tabellbeskrivning
Tabellen har 9 rader och 9 kolumner.

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27

4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

4.5 En kommun har gjort följande prognos för befolkningsutvecklingen de närmaste åren:

- Vid början av år 2022 hade kommunen 26 000 invånare.

- Antalet födda och avlidna under ett år uppskattas vara 0,7 % respektive 0,6 % av befolkningen vid årets början.
- Antalet inflyttade och antalet utflyttade uppskattas till 300 respektive 325 varje år.

Skriv ett program som beräknar kommunens uppskattade invånarantal i början av ett visst år. Vilket år det gäller ska läsas som indata till programmet.

[överkurs]
4.6 Skriv ett program som beräknar summan $\frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots$. Upprepningen ska avslutas när den sista termen man lagt till har ett absolutbelopp som är mindre än 0,00001.

[överkurs]
4.7 Vid nationella simhoppstävlingar bedöms varje hopp av minst tre domare som ger poäng på en skala från 0 till 10. Beräkning av hoppets poäng görs på följande sätt: Man bortser från den högsta och den lägsta domarpoängen. Därefter beräknas medelvärdet av de återstående domarpoängen. Hoppets poäng får man fram genom att multiplicera det erhållna medelvärdet först med 3 och sedan med ett tal som anger hoppets svårighetsgrad.
Skriv ett program som beräknar simhoppspoäng på detta sätt. Programmet ska först läsa in antalet domare och kontrollera att det är minst tre. Programmet ska utformas så att det sedan kan beräkna poäng för ett godtyckligt antal hopp. Varje ny beräkning ska inledas med att programmet läser in hoppets svårighetsgrad. Därefter ska domarpoängen läsas in. Det ska sedan beräkna och skriva ut hoppets poäng. Användaren ska kunna ange att programmet ska avslutas genom att skriva ett negativt tal vid någon inläsning.

5 Att hantera text

Hittills i boken har vi bara använt oss av variabler som innehållit numeriska data, men en stor mängd av de data som behandlas i datorer är text. I detta kapitel ska vi titta närmare på typen `str`. Vi ska se hur man bildar texter och hur man hanterar dem i sitt program.

5.1 Textliteraler

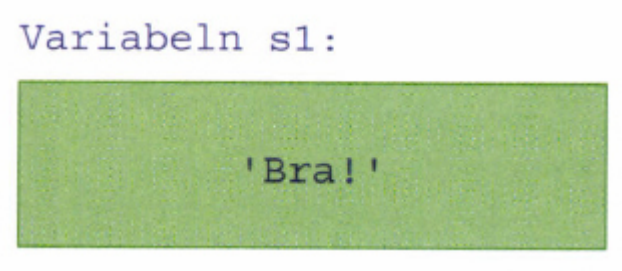
Konstanta värden som man skriver i programkoden kallas *literaler.* I avsnitt 2.2 diskuterade vi hur man kunde skriva konstanta numeriska värden. Sådana kallades numeriska literaler. Men texter som man skriver i programkoden är också en form av literaler. Dessa kallas *textliteraler.* En textliteral ska innehålla texten omgiven av citationstecken (dubbla apostrofer), t.ex. `"Python"`, eller enkla apostrofer, t.ex. `'Python'`. (I denna bok använder vi för det mesta enkla apostrofer eftersom Python-interpretatorn själv i sina utskrifter använder sådana.) En text består av en följd av tecken. Varje tecken representerar en grafisk symbol, t.ex. en bokstav, en siffra eller ett specialtecken (`% * =` etc.).

Om man tilldelar en text till en variabel kommer den att få typen `str`. Om vi t.ex. skriver

```
s1 = 'Bra!'
```

kommer `s1` att innehålla en text som består av de fyra tecknen `B`, `r`, `a` och `!`. Det kan illustreras som i figur 5.1.

Figur 5.1 En str-variabel.



Man kan slå samman två textliteraler till en enda genom att skriva dem intill varandra. Skriver vi t.ex. `'Hej' 'då'`, blir det samma sak som om vi hade skrivit `'Hej då'`. Detta kan vara användbart om man vill skriva delar av texten på olika rader i programmet, t.ex.

```
s2 = 'Hej ' "då"
s3 = 'Python '\
        'från början'
print(s2)
print(s3)
```

Utskriften blir

```
Hej då
Python från början
```

Lägg på första raden märke till att det går att använda antingen citationstecken eller enkla apostrofer. Observera också att vi måste skriva tecknet `\` sist på andra raden för att markera att satsen fortsätter.

I en textliteral skriver man för det mesta ett tecken för varje tecken man vill ska ingå i texten, som t.ex. när vi tilldelade texten `'Bra!'` till variabeln `s1`. Men tecknet `\` (backslash) är lite speciellt. Om man skriver ett sådant inne i en textliteral ska det inte läggas i texten. Det markerar i stället att nästa tecken som kommer är speciellt. Om vi t.ex. vill att `s4` ska innehålla texten *Anropa heter 'call' på engelska,* kan vi *inte* skriva

```
s4 = 'Anropa heter 'call' på engelska' # FEL !!
```

Då tror nämligen Python-interpretatorn att texten tar slut efter ordet `heter` och resten av raden blir felaktig. Ett sätt att lösa problemet är att använda tecknet `\`.

```
s4 = 'Anropa heter \'call\' på engelska'
```

Då markerar tecknet `\` att nästa tecken, dvs. ', ska behandlas speciellt och inte uppfattas som början eller slutet på en text. Ett lite enklare sätt hade varit att använda citationstecken runt hela texten:

```
s4 = "Anropa heter 'call' på engelska"
```

Man får också göra på ett speciellt sätt om man faktiskt vill lägga tecknet `\` i en text. Vi kan t.ex. skriva

```
s5 = 'Tecknet \\ kallas "backslash"'
```

Då kommer `s5` att innehålla texten *Tecknet \ kallas "backslash".* Vi måste alltså skriva två backslash-tecken. Det första säger att nästa tecken ska tolkas som vilket tecken som helst.

Teckenkombinationer som börjar med tecknet `\` kallas *escape-sekvenser.* Ett par escape-sekvenser som är vanliga inom programmering är `\n` och

---

`\t`. Sekvensen `\n` betyder att ett tecken för ny rad *(newline)* ska läggas i texten och `\t` betyder ett tabulatortecken. Om man skriver ut en text som innehåller sådana tecken, kommer man att få nya rader och tabulatorer. Om vi t.ex. har satserna

```
s6 = 'Detta är \trad 1\noch detta rad 2'
print(s6)
```

så ser utskriften ut så här:

```
Detta är rad 1
och detta rad 2
```

Man kan skriva ett s.k. *prefix* först i en textliteral. Det är en bokstav som man skriver före första citationstecknet eller apostrofen. Detta har vi faktiskt sett flera exempel på redan. I kapitel 2 introducerade vi *formatted string literals,* s.k. *f-Strings.* (Se sidan 31.) Där skrev vi bokstaven `f` först i literalen för att markera att det var en text som

kunde innehålla platshållare för redigering av utskrift. Ett exempel var textliteralen

```
f'Talet i kvadrat är {y:.2f}'
```

Här är prefixet `f`. Det hade också varit tillåtet att skriva ett stort `F`.

Ett annat prefix man kan ge är bokstaven `r` eller `R`. Då får man en s.k. *raw string.* I en sådan betraktas backslash-tecknet som ett vanligt tecken och man kan alltså inte använd escape-sekvenser i en sådan text. Vi kan t.ex. skriva

```
s7 = r'c:\windows\py.exe'
```

Om vi inte hade haft med prefixet `r`, hade vi varit tvungna att skriva dubbla backslash-tecken `\\`.

Det finns ett alternativt sätt att skapa texter som ska omfatta flera rader. Man kan använda s.k. *triple-quoted strings.* Då omger man texten med tre citationstecken eller apostrofer, både före och efter.

```
s8 = '''Detta är den första raden,
detta den andra, "the second"
och detta tredje, 'the third'.'''
print(s8)
```

Tecknen för ny rad kommer då att hamna i texten automatiskt. Lägg märke till att man får ha enkla citationstecken och apostrofer inne i en triple-quoted string. De betraktas då som vanliga tecken. Utskriften blir

```
Detta är den första raden,
detta den andra, "the second"
och detta tredje, 'the third'.
```

5.2 Teckenkoder och Unicode

En text består av en följd av enstaka tecken som illustrerades i figur 5.1. För att programmet ska veta vilka grafiska tecken som texten innehåller använder man sig av nummer. Varje grafisk symbol har ett speciellt nummer, en s.k. *teckenkod.* Tecknet `a` har t.ex. nummer 97 och tecknet `@` nummer 64. Det som lagras i programmet är tecknets nummer, som ju är ett heltal. Om man t.ex. tilldelar variabeln `c` tecknet `a`, kommer `c` att innehålla värdet 97.

Vilket nummer ett viss tecken har bestäms av internationella standarder som de flesta datorsystem ansluter sig till. Den första standarden var *ASCII-standarden.* I den fanns bara 128 olika tecken, numrerade från 0 till 127. Anledningen till att det bara fanns 128 tecken var att man endast använde sju databitar (nollor och ettor) för att lagra tecknens nummer, och sju databitar kan bara kombineras på 128 olika sätt. I ASCII-standarden fanns bara bokstäverna a–z. De svenska bokstäverna å, ä och ö saknades, liksom många andra bokstäver, t.ex. ñ, ü och ê.

Lite bättre blev det när man började använda 8 databitar i stället för sju. Då kunde man nämligen representera 256 olika tecken. Det finns en internationell standard *ISO 8859-1,* även kallad *LATIN_1,* som beskriver vilka 256 tecken som ingår. Denna standard är en ren

utökning av ASCII-standarden. I LATIN_l-standarden ingår tecken som används i de romanska och germanska språken, men andra typer av skrivtecken saknas. I operativsystemet Windows kodas tecken enligt en standard som kallas *CP-1252,* vilken är en variant av LATIN_1-standarden. I vissa redigeringsprogram, t.ex. *Anteckningar (Notepad),* kallas CP-1252 något felaktigt för *ANSI.* (Det gamla MS-DOS hade däremot en annan kodning, vilket förklarar att en del tecken blir konstiga när man visar dem i ett kommandofönster.)

Att använda LATIN_1 i stället för ASCII löser förstås bara problemet för de länder där de västeuropeiska språken används. Därför har man tagit fram den internationella *Unicode*-standarden. Teckenkoderna i Unicode kallas *code points;* varje tecken har alltså ett unikt sådant nummer.

Koderna 0 – 255 överensstämmer med LATIN_l-standarden. I Unicode är det möjligt att definiera mer än en miljon olika tecken. (För närvarande är ca 140 000 definierade.) En förteckning över alla tecken i Unicode finns på `www.Unicode.org`.

För att man ska kunna lagra teckenkoder för så många olika tecken räcker det förstås inte med de 8 bitar (1 byte) som används i ASCII- och LATIN l-standarderna. Därför finns i Unicode-standarden också angivet tre olika tekniker som man kan använda för att lagra ett teckens nummer. Dessa tekniker är *UTF-8, UTF-16* och *UTF-32.* I UTF-32, som är enklast lägger man helt enkelt tecknets nummer i en heltalsvariabel som är 32 bitar lång. Problemet med detta är att man slösar med minne. I Unicode ligger de vanligaste (åtminstone om man använder det latinska alfabetet) tecknen i början. De första 128 tecknen är identiska med ASCII-koden och det räcker med en byte (8 bitar) för att lagra dessa. I tekniken UTF-8 används en enda byte om

tecknet har en kod som ligger i intervallet 0–127. Om tecknet har en högre kod används ytterligare en, två eller tre byte. Olika tecken kan alltså lagras med olika många byte i UTF-8. Tanken är att de vanligaste tecknen ska ligga först och därför lagras med en eller två byte, medan de ovanligare kräver tre eller fyra byte. UTF-8 lagrar därför text på ett kompakt sätt. Tekniken UTF-16 är ett mellanting mellan UTF-8 och UTF-32. Där lagras tecken med antingen en eller två grupper med 16 bitar.

Om man inte anger något annat, förutsätter Python-interpretatorn att textfilen som innehåller programkoden är kodad med tekniken UTF-8. Detta betyder att alla tecken som ingår i unicode-standarden kan ingå i en textsträng. När man skriver sitt program kan man alltså använda sådana tecken. Detta är inget problem, om de tecken man vill lägga i en text går att skriva på tangentbordet, t.ex.

```
t1 = 'Price: £2 '
t2 = '§9 Resumé'
t3 = 'Não!'
```

Vill man ange ett annat tecken måste man använda en *escape*-sekvens. Det finns några fördeklarerade escape-sekvenser för vanliga specialtecken. Teckenkombinationen `\n` som betyder tecknet för ny rad och `\t` som betyder tabulatortecknet stiftade du bekantskap med i förra avsnittet. Fler escape-sekvenser visas i faktarutan.

Escape-sekvenser för icke-skrivbara tecken

`\a` (alert) Ger en ljud- eller ljussignal.

`\b` (backspace) Flyttar utskriftspositionen ett steg till vänster.

`\f` (form feed) Flyttar utskriftspositionen till början på nästa sida.

`\n` (newline) Flyttar utskriftspositionen till början på nästa rad.

`\r` (carriage return) Flyttar utskriftspositionen till början på raden.

`\t` (horisontal tab) Flyttar fram utskriftspositionen till nästa tabulatorstopp.

`\v` (vertical tab) Flyttar utskriftspositionen till början på den rad som är markerad som nästa vertikala tabulatorstopp.

`\ccc` tecknet med den oktala koden `ccc`

`\xcc` tecknet med den hexadecimala koden `cc`

`\N{name}` tecknet med namnet `name` i Unicode

`\ucccc` tecknet med den hexadecimala koden `cccc`

`\uccccccccc` tecknet med den hexadecimala koden `cccccccc`

Escape-sekvenser är användbara om man vill ange ett tecken som inte finns på tangentbordet, t.ex. tecknet ♣. Då kan man skriva en escapesekvens som består av teckenkombinationen `\u` följd av tecknets nummer enligt Unicode-standarden. I denna standard anges numren i s.k. hexadecimal form, vilket kanske ser lite konstigt ut. Men det är inget problem eftersom de ska skrivas i programmet på

samma sätt. Anta t.ex. att vi vill lägga in klövertecknet ♣ i en text. På webbsidan `www.unicode.org/charts/` kan vi se att tecknet ♣ har numret `2663`. Vi kan då t.ex. skriva

```
t4 = 'Jag har \u2663 Kung'
print(t4)
```

Utskriften blir då

```
Jag har ♣ Kung
```

I stället för att ange tecknets hexadecimala kod kan man använda formen `\N` och ange tecknets namn i Unicode. Vi kan t.ex. skriva


```
t5 = 'Det kostar 10 \N{EURO SIGN}'
t6 = 'Det regnar \N{UMBRELLA}'
```

Uppgift 5.1

Skriv ett program som skriver ut en text som innehåller symbolen för svart kung i schack och tecknet π *Tips:* Gå till `www.unicode.org/charts/` och leta efter *Other Symbols* → *Miscellaneous Symbols* respektive *European Scripts*→ *Greek.*

Det finns också ett par funktioner som man kan använda för att ta reda på teckenkoden för ett visst tecken och för att skapa ett visst tecken om man vet teckenkoden. Funktionerna heter `chr` respektive `ord`:

```
chr(97) # ger värdet 'a'
ord ('ö') # ger värdet 246
```

5.3 Operationer på sekvenser

I Python finns en grupp typer som beskriver s.k. *sekvenser.* En variabel som är av någon av dessa typer kan innehålla *flera* värden, inte bara *ett* värde. Värdena ligger i en följd efter varandra. Variabler av typen `str` är sekvenser. Andra slag av sekvenser är t.ex. *listor* och *tupler* vilka vi behandlar i nästa kapitel. I detta avsnitt diskuteras olika operationer som man kan utföra på alla slag av sekvenser. I exemplen använder vi typen `str`, men allt som sägs i detta avsnitt gäller också för andra typer av sekvenser, t.ex. listor.

5.3.1 Indexering

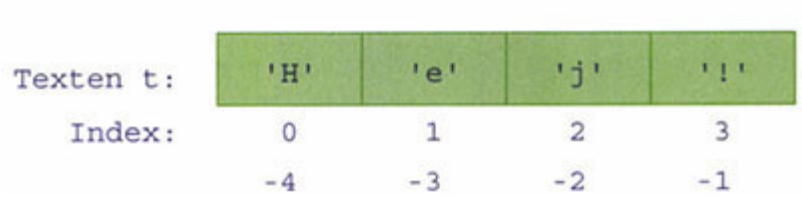
De ingående värdena i en sekvens brukar kallas *element (elements* eller *items).* Elementen är numrerade och deras nummer kallas *index.* Det första elementet i en sekvens har index 0, det andra index 1, osv. Anta t.ex. att vi gör tilldelningen

```
t = 'Hej!'
```

Då kan innehållet i variabeln `t` illustreras som i figur 5.2.

I figuren visas också elementens index. Eftersom texten `t` innehåller fyra tecken och indexering sker från 0 har det sista elementet indexet 3.

Figur 5.2 En str-variabel med index.



Bilden visar en tabell med 3 rader och 5 kolumner. Första raden visar variabeln Texten t och sedan listan 'H'; 'e'; 'j'; och '!'. Andra raden visar Index: 0, 1, 2, och 3. Tredje raden visar -4; -3; -2; och -1.

Man kan avläsa ett visst element i en sekvens om man skriver variabelnamnet följt av elementets index innanför hakparenteser.

```
t[2] # ger värdet 'j'
```

När man skriver på detta sätt kallas det att man utför *indexering* eller att man *indexerar.*

Vill man få reda på hur många element som ingår i en sekvens kan man använda funktionen `len`. Här undersöker vi t.ex. hur många tecken som finns i variabeln `t`:

```
len(t) # ger värdet 4
```

När man indexerar måste man ange ett index som ligger innanför gränserna. Det finns inga negativa index, men man kan i Python faktiskt ange ett negativt index. Då kommer nämligen sekvensens längd att automatiskt adderas till det man angett. Det betyder att man enkelt kan räkna bakifrån i stället för framifrån. Om man t.ex. skriver `t[-1]`, så kommer detta att automatiskt översättas till `t[len(t) -1]`, vilket blir `t[3]` som är indexet för det sista elementet i `t`. Detta visas på sista raden i figur 5.2. Det sista elementet har index -1, det näst sista -2 osv.

```
t[-3] # ger värdet 'e'
```

Man behöver inte använda ett konstant värde när man indexerar. Det går bra att skriva ett variabelnamn eller ett uttryck innanför hakparenteserna, men uttrycket måste vara ett heltal. Vi kan t.ex. skriva:

```
i = 1
j = 2
t[i]) # ger värdet 'e'
```

```
t[i+j] # ger värdet '!'
```

Man måste vara försiktig så att man inte hamnar utanför gränserna. Om variabeln `j` har värdet 2, och vi t.ex. skriver `t[2*j]` blir indexeringen felaktig eftersom det inte finns något element i `t` med index 4. Vi får då en felutskrift:

```
t[2*j]
Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Indexering

Elementen i en sekvens `s` numreras från 0 och uppåt.

Antalet element i en sekvens kan beräknas med funktionen `len`:

```
len(s)
```

Man kan komma åt ett enstaka element i en sekvens genom att *indexera:*

`s[i]`

Indexet `i` ska vara ett heltal i intervallet `0` till

$$\text{len(t)} - 1$$

.

Man kan alternativt ge negativa index. Då räknar man bakifrån.

$$\text{s}[-1]$$

är det sista värdet,

$$\text{s}[-2]$$

det näst sista osv.

Uppgift 5.2

Skriv ett program som läser in en text. Programmet ska undersöka om textens första och sista bokstav är samma.

5.3.2 Skivor

När man indexerar väljer man ut ett enda värde i en sekvens. Men det går också att välja ut flera element. Man säger då att man bildar en *skiva (slice).* En skiva har formen *`sekvens`* `[forst: sist: steg]`. Innanför hakparenteserna kan man ange tre heltal: första index i skivan, slutindex och steget. Man ska skriva kolon mellan de tre värdena. Det är tillåtet att utelämna ett eller två av heltalen, men

man måste ändå ha med kolontecknet om man utlämnar något av de två första. Utelämnar man första index antas att detta är 0, utelämnar man slutindex antas att detta är lika med sekvensens längd, och utelämnar man steget antas att detta är lika med 1. Lägg märke till att slutindex är index för det första värdet som *inte* ska vara med i skivan. Här kommer några exempel:

```
s = 'Programspråk'
s [3:7] # ger värdet 'gram'
s [7:] # ger värdet 'språk'
s [:3] # ger värdet 'Pro'
s[::2] # ger värdet 'Pormpå'
s[6:2:−1] # ger värdet 'marg'
s[::−1] # ger värdet 'kårpsmargorP'
```

Det är faktiskt tillåtet att gå utanför gränserna när man bildar en skiva:

```
s[7:100] # ger värdet 'språk'
s[12 :] # ger värdet ''
```

Skivor

Man kan bilda en skiva från en sekvens `s`. En skiva blir en ny sekvens.

```
s[forst:sist:steg]
```

`forst`, `sist` och `steg` ska vara ett heltal.

`forst` anger skivans startindex, `sist` anger första index som inte ska vara med i skivan.

`steg` anger vilket steg man ska använda för att välja nästa element från sekvensen. Om man har ett negativt steg, väljs elementen bakifrån.

Man kan utelämna någon eller några av `forst, sist` och `steg`, men om man utelämnar `forst` eller `sist` måste kolontecken ändå vara med. Utelämnas `forst` antas att den är lika med 0, utelämnas `sist` antas att den är lika med `len(s)` och utelämnas `steg` antas att den är lika med 1.

Uppgift 5.3

Kör i *interactive mode* och skapa en variabel som innehåller texten `'pytonorm'`. Använd sedan denna för att skapa tre skivor som innehåller texterna `'ton', 'norm'` och `'not'`.

5.3.3 Jämförelser

Man kan jämföra sekvenser med varandra. Man kan undersöka om två sekvenser är lika eller olika. De uppfattas som lika om de innehåller lika många element och motsvarande element är lika.

```
'abc' == 'abc' # ger värdet True
'abc' == 'ab' # ger värdet False
'abc' == 'Abc' # ger värdet False
```

Lägg märke till att stora och små bokstäver betraktas som olika.

För en del sekvenser, t.ex. texter och listor, kan man också undersöka om en sekvens är "mindre" eller "större" än en annan. Då används s.k. *lexikografisk ordning.* Det är samma ordning som man använder för att ordna ord alfabetiskt. Det går till så här: Man jämför elementen parvis från början. Om de första elementen i sekvenserna är lika, men den ena sekvensen är kortare än den andra, betraktas den kortare som minst:

```
abc' < 'abcd' # ger värdet True
```

Om ingen av sekvenserna är slut, avgör de första elementen som är olika vilken sekvens som betraktas som minst. Den sekvens vars element är minst blir då den minsta sekvensen.

```
'abcde ' < 'abdc ' # ger värdet True
'abc' < 'abba' # ger värdet False
```

När det gäller texter är det de ingående tecknens teckenkoder som jämförs. Detta innebär att en jämförelse inte alltid ger det resultat man kan förvänta sig:

```
'å' < 'ä' # ger värdet False
```

Teckenkoderna för a-z kommer i "rätt" ordning, men koderna för å och ä ligger tyvärr inte i samma ordning som i det svenska alfabetet.

5.3.4 Operatorn `in`

När man arbetar med sekvenser, t.ex. texter, behöver man ofta löpa igenom alla elementen. Då kan man använda en `for`-sats. Här är ett exempel där vi räknar ut hur många blanka tecken det finns i en inläst text.

[fullständigt program]

```
# Räkna blanka tecken, version 1
s = input('Skriv en text: ')
n = 0
for i in range(0, len(s)):
        if s [i] == '':
                n = n + 1
print(f'Texten innehåller {n} blanka tecken')
```


Här har vi i `for`-satsen använt variabeln `i` som index och låtit den löpa i intervallet `0` till

$$\text{lens}(s) - 1$$

. (Kom ihåg att i anropet av `range` ska det andra argumentet vara det sista värdet plus 1. Se sidan 68.)

I själva verket ska det efter ordet `in` i en `for`-sats stå en sekvens. Funktionen `range` genererar en sekvens där elementen är heltal. Uttrycket `range (1, 4)` ger t.ex. sekvensen `1, 2, 3`. Räknaren i `for`-satsen, variabeln `i` i vårt exempel, löper igenom sekvensen. På första varvet tilldelas den sekvensens första element, på andra varvet dess andra osv.

Eftersom en variabel av typen `str` innehåller en sekvens finns ett enklare sätt att löpa igenom en text än vad vi visade ovan. Vi kan skriva `str`-variabelns namn direkt efter ordet `in` i `for`-satsen:

[fullständigt program]

```
# Räkna blanka tecken, version 2
s = input('Skriv en text: ')
n = 0
for c in s:
        if c == ' ':
                n = n + 1
print(f'Texten innehåller {n} blanka tecken')
```

Raden som är markerad med rött betyder att man ska löpa igenom alla element i `str`-variabeln `s`. På det första varvet tilldelas variabeln `c` värdet `s[0]`, på det andra värdet `s[1]` osv.

Ett enkelt sätt att få både indexet och värdet när man löper igenom en sekvens är att använda funktionen `enumerate`. Så här kan det se ut:

```
s = 'hej'
for i, c in enumerate(s):
        print(i, c)
```

Utskriften blir

```
0 h
1 e
2 j
```

Operatorn `in` kan man också använda för att undersöka om ett visst element finns i en sekvens. Vi kan t.ex.undersöka om bokstaven `'e'` finns i variabeln `v`:

```
v = 'avdelning'
'e' in v # ger värdet True
```

Speciellt för typen `str` gäller också att man kan använda operatorn in för att undersöka om en viss text ingår som en del i en annan text:

```
'del' in v # ger värdet True
'dela' in v # ger värdet False
```

Man kan förstås använda sig av operatorn `in` i `if`-satser. Följande program läser in en text till en `str`-variabel `s` och undersöker om den innehåller något *vitt tecken.* Med ett vitt tecken menas här ett *mellanslag* eller ett *tabulatortecken.* (Som vitt tecken räknas egentligen också `\n`.) Ett vitt tecken kan stå mellan olika ord och skilja dem åt.

[fullständigt program]

```
# Undersök om en text innehåller något vitt tecken
s = input('Skriv en text: ')
if ' ' in s or '\t' in s:
        print('Innehåller vitt tecken')
else:
        print('Inget vitt tecken')
```

Detta program undersöker bara om det finns minst ett vitt tecken. Vill man få reda på var i texten det första vita tecknet finns kan, man köra följande program. Om det finns minst ett vitt tecken i `s`, talar programmet om indexet för detta tecken, annars skriver det ut meddelandet `Inget vitt tecken`.

[fullständigt program]

```
# Leta efter det första vita tecknet i en text
s = input('Skriv en text: ')
i = 0 # i används som räknare
for c in s :
        if c == ' ' or c == '\t' :
                break
        i = i + 1
if i < len(s) :
        print(f'Första vita tecken finns på plats
nr {i}')
```

```
else :
        print('Inget vitt tecken')
```

Här används en räknare `i` som löper igenom tecknen i variabeln `s`. Under varje varv undersöks om tecken nummer `i` innehåller ett mellanslag eller ett tabulatortecken. I så fall utförs en `break`-sats. Detta betyder att `for`-satsen avbryts om man hittar ett vitt tecken, och att variabeln `i` då innehåller indexet för detta tecken. I detta fall innehåller

---

`i` ett tal som är mindre än `len(s)`. Om det inte finns några vita tecken, löper `for`-satsen alla varven och `i` blir till slut lika med `len(s)`. Anledningen till att vi inte använder en `for`-sats med `range` är att i en sådan skulle räknaren ha haft värdet

$$\text{len(s)} - 1$$

efter sista varvet.

Operatorn `in`

Operatorn `in` kan användas för att bilda logiska uttryck:

```
e in s
```

Uttrycket blir sant om elementet `e` ingår i sekvensen `s`.

Operatorn `in` kan också användas i `for`-satser:

```
for e in s:
```

```
    en eller flera indragna satser
```

På det första varvet tilldelas `e` det första elementet i sekvensen `s`, på det andra varvet det andra elementet osv.

När `for`-satsen avslutats innehåller `e` det sista elementet i sekvensen `s`.

Uppgift 5.4

Gör om programmet så att man i stället letar efter det *sista* vita tecknet.

Uppgift 5.5

[överkurs]

Skriv en ny version av programmet som letar efter det första vita tecknet. Använd en `for`-sats med `range` och `else`-del.

5.3.5 Operatorerna + och *

I inget av exemplen vi visat hittills har vi har försökt ändra i en text. Man får ändra en `str`-variabel så att den innehåller en helt ny text, men själva *texten kan inte ändras.* Om vi försöker får vi en felutskrift:

```
s = 'gammal'
s = 'ny' # OK
s[1] = 'u' # FEL, ger en felutskrift
```

Vissa slag av sekvenser är ändringsbara, t.ex. listor, medan vissa, t.ex. sekvenser av typen `str`, inte är det.

Man kan alltså inte ändra i en text, men det går att skapa *nya* texter genom att kopiera delar av texter och sätta samman dem. Ett sätt att göra detta är att använda operatorn `+`. Denna används förstås när man vill utföra additioner och har numeriska operander, men om båda operanderna i stället är sekvenser skapar operatorn `+` en ny sekvens genom att slå samman de två sekvenserna:

```
s1 = 'av'
s2 = 'sluta'
s3 = s1 + s2 # s3 får värdet 'avsluta'
s1 += 'dela' # s1 får värdet 'avdela'
```

Ingen av operanderna till operatorn `+` ändras. Variabeln `s3` kommer att innehålla en ny sekvens. På sista raden tilldelas `s1` en helt ny text.

Som exempel kommer här ett program som översätter datum, skrivet i den form som används i USA, till den form vi brukar använda. I Sverige skriver vi `åååå-mm-dd`, t.ex. `2022-11-14`. I USA skriver man i stället datum som `mm/dd/åå`, t.ex. `11/14/22`.

[fullständigt program]

```
# Översätt amerikanskt datum till svensk form
a = input('Skriv ett amerikanskt datum som mm/dd/
åå: ')
månad = a[:2]
dag = a [3:5]
år = a [6:]
s = '20' +år+ '-' + månad + '-' + dag
print('Svenskt datum: ' + s)
```

Uppgift 5.6

Skriv ett program som översätter ett svenskt datum till amerikansk form.

En annan operator som man kan använda för att skapa nya sekvenser är operatorn `*`. Den ena operanden ska då vara en sekvens och den andra ett heltal. Detta demonstreras här.

```
n = 'Nej'
j = 'Ja'
a = 3 * n # a får värdet 'NejNejNej'
j = *5 # j får värdet 'JaJaJaJaJa'
```

Den nya sekvens som skapas innehåller alltså den ursprungliga sekvensen så många gånger som heltalet anger.

Operatorerna + och *

Operatorn `+` kan användas för att bilda en ny sekvens som är en sammanslagning av de två operanderna.

```
sekvens1 + sekvens2
```

Operatorn `*` kan användas för att bilda en ny sekvens som upprepar en sekvens ett visst antal gånger

```
antal * sekvens
sekvens * antal
```

De ingående operanderna i dessa operationer ändras inte.

5.3.6 Funktioner

Det finns några funktioner som är gemensamma för alla slag av sekvenser. En av dem har du redan sett, nämligen funktionen `len` som beräknar antalet element i en sekvens. Två andra är `min` och `max` som ger det minsta respektive största värdet i en sekvens.

```
a = 'programspråk'
len(a) # ger värdet 12
min(a) # ger värdet 'a'
max(a) # ger värdet 'å'
```

För sekvenser av typen `str` är det tecknens koder som jämförs i funktionerna `min` och `max`.

För att få reda på hur många gånger ett visst element finns i en sekvens kan man använda funktionen `count`:

```
a.count('r') # ger värdet 3
```

Lägg märke till att man ska skriva sekvensens namn *före* funktionsnamnet när man anropar `count`. En funktion som ska anropas på det sättet kallas egentligen i Python en *metod,* till skillnad från "vanliga" funktioner som anropas enligt modellen `len(s)`. Men för enkelhets skull använder vi ordet *funktion* även för metoder än så länge.

En annan funktion som finns för alla sekvenser är `index`. Med hjälp av denna kan man få reda på var i sekvensen ett visst element finns.

```
a.index('r') # ger värdet 1
```

Om det element man söker finns på flera ställen i sekvensen får man reda på det första stället. Om man anropar funktionen `index` och det element man söker efter inte finns, får man en felutskrift. Därför kan det vara klokt att anropa `count` först.

Gemensamma funktioner för sekvenser

I denna ruta betecknar `s` och `t` sekvenser och `n` och `m` heltal.

`len(s)` ger antalet element i `s`

`min(s)` ger det minsta elementet i `s`

`max(s)` ger det största elementet i `s`

`sum(s)` ger summan av elementen i `s` (ej för typen `str`)

`s.count(t)` anger hur många gånger `t` finns i `s`

`s.count(t,n,m)` anger hur många gånger `t` finns i intevallet `n` till

$$\mathrm{m}-1$$

i `s`

`s.index(t)` ger index för första förekomsten av `t` i `s`, ger en felutskrift om `t` inte finns

`s.index (t,n,m)` som ovan, men söker bara i intervallet `n` till

$$\mathrm{m}-1$$

`sorted (s)` ger som resultat en sorterad lista, `s` ändras inte

5.4 Mer om typen `str`

Förutom de operationer för sekvenser som behandlades i förra avsnittet finns det för typen `str` ett stort antal funktioner som ingår i Pythons standardbibliotek. Vi ska diskutera några av dem i detta avsnitt. I faktarutan ser du en sammanställning. Du behöver inte förstå alla detaljer i denna sammanställning nu. Den har gjorts omfattande för att du ska kunna hitta allt på ett ställe i boken.

I faktarutan finns två kategorier av funktioner: sådana som undersöker texter och sådana som bildar nya texter. Det finns inga funktioner som *ändrar* i någon text eftersom detta inte är tillåtet. Du ska nu få se exempel på hur några av funktionerna för typen `str` kan användas.

Funktionerna `find` och `rfind` finns i flera varianter. De används när man vill leta efter ett visst tecken eller en viss deltext i en `str`-variabel. De ger alla som resultat numret på positionen där det man letar efter finns. Skulle det man letar efter inte finnas, ger de värdet -1.

Funktioner för typen `str`

I denna ruta betecknar `s` och `t` variabler av typen `str` och `n` och `m` heltal.

`s.find(t)` ger index för första förekomsten av `t` i `s`, ger -1 om `t` inte finns

`s.find(t,n,m)` som föregående, men söker bara i intervallet `n` till

$$m-1$$

`s.rfind(t)` som `s.find(t)` men söker bakifrån

`s.rfind(t,n,m)` som `s.find(t,n,m)` men söker bakifrån

`s.startswith(t)` ger `True` om `s` inleds med deltexten `t`, `False` annars

`s.startswith(t,n,m)` som föregående, men undersöker bara intervallet `n` till

$$m-1$$

`s.endswith(t)` ger `True` om `s` avslutas med deltexten `t`, `False` annars

`s.endswith (t,n,m)` som föregående, men undersöker bara intervallet `n` till

$$m-1$$

`s.islower()` ger `True` om alla bokstäver i `s` är små, `False` annars

`s.isupper()` ger `True` om alla bokstäver i `s` är stora, `False` annars

`s.isspace ()` ger `True` om alla tecken i `s` är vita, `False` annars

`s.isdecimal ()` ger `True` om alla tecken i `s` är decimala siffror, `False` annars

`s.isalpha ()` ger `True` om alla tecken i `s` är alfabetiska, `False` annars

`s.lstrip ()` ger en kopia av `s` där inledande vita tecken tagits bort

`s.lstrip (t)` ger en kopia av `s` där inledande tecken som ingår i `t` tagits bort

`s.rstrip ()` ger en kopia av `s` där avslutande vita tecken tagits bort

`s.rstrip (t)` ger en kopia av `s` där avslutande tecken som ingår i `t` tagits bort

`s.strip ()` som anrop av både `s.lstrip()` och `s.rstrip()`

`s.strip (t)` som anrop av både `s.lstrip (t)` och `s.rstrip(t)`

`s.replace (s1,s2)` ger en kopia av `s` där texten `s1` bytts ut mot `s2` överallt

`s.replace (s1,s,n)` som ovan, men byter bara de `n` första

`s.lower ()` ger en kopia av `s` där alla stora bokstäver bytts ut mot små

`s.upper ()` ger en kopia av `s` där alla små bokstäver bytts ut mot stora

`s.capitalize()` ger en kopia av `s` där den första bokstaven är stor, resten små

`s.title ()` ger en kopia av `s` där alla ord börjar med stora bokstäver

`s.swapcase ()` ger en kopia av `s` där alla små bokstäver bytts ut mot stora och tvärtom

`s.ljust (n)` ger `s` vänsterjusterad i en text med längd `n`, utfyllnad med `''`

`s.ljust (n, t)` ger `s` vänsterjusterad i en text med längd `n`, utfyllnad med `t`

`s.rjust (n)` ger `s` högerjusterad i en text med längd `n`, utfyllnad med `''`

`s.rjust(n, t)` ger `s` högerjusterad i en text med längd `n`, utfyllnad med `t`

`s.center (n)` ger `s` centrerad i en text med längd `n`, utfyllnad med `''`

`s.center (k, t)` ger `s` centrerad i en text med längd `n`, utfyllnad med `t`

`s.split ()` ger en lista med element som avgränsas med blanka tecken i `s`

`s.split (t)` ger en lista med element som avgränsas med `t` i `s`.

`s.splitlines ()` ger en lista med element som avgränsas med radsluttecken i `s`.

`s.join(tlist)` `tlist` är en lista med texter. Ger en text med där texterna i listan fogats samma med `s` mellan varje deltext

`s.partition (t)` letar efter `t`, ger en tupel: (del före `t`, `t`, del efter `t`)

`s.rpartition (t)` som ovan, men letar bakifrån

```
a = 'programspråk'
a. find('språk') # ger värdet 7
a.find('Python') # ger värdet −1
```

Funktionerna `startswith` och `endswith` undersöker om en viss text finns i början eller slutet av en text eller i en skiva av en text.

```
a.startswith('pr')) # ger värdet True
a.endswith('!') # ger värdet False
a.startswith('rog', 1, len(a)) # ger värdet True
```

Funktionen `replace` skapar en kopia av texten där en viss deltext bytts ut mot en annan på alla ställen. Anta t.ex. att du vill skapa en kopia av texten i variabeln `a` där alla `'r'` ersatts med `'l'`.

```
b = a.replace('r', 'l') # b får värdet
'ploglamsplåk'
```

Du har förhoppningsvis noterat att i alla exempel har `str`-variabeln `a` blivit oförändrad. Ingen av funktionerna i faktarutan påverkar nämligen den `str`-variabel man undersöker. Vill man ändra en `str`-variabel måste man tilldela en helt ny text till den. Anta t.ex. att du vill ändra alla stora bokstäver i `str`-variabeln `w` till små bokstäver. Då kan du använda funktionen `lower` som i följande exempel

```
w = 'Windows 10 Pro'
w = w.lower() # w får värdet 'windows 10 pro'
```

Ett något mer komplicerat exempel visas nu. Anta att variabeln `a` tilldelats namn och personnummer för en person, t.ex.

```
a = ' Erik Andersson 990314-2714 '
```

Man vill nu plocka ut födelsedatum, dvs. `'990314'` i detta exempel, och lägga det i en annan variabel `b`. Det kan göras med följande satser

```
a = a.strip() # a blir 'Erik Andersson 990314-
2714'
i = a.rfind(' ') +1 # i får värdet 15
j = a.find('-') # j får värdet 21
b = a[i:j] # b blir '990314'
```

I den första satsen tas de blanka tecken bort som finns först och sist i `a`. På andra raden ger anropet av `rfind` positionen för det sista blanka tecknet i texten. Genom att addera 1 till denna position får man startpositionen för datumet. Slutpositionen får man på tredje raden med hjälp av `find` som letar reda på positionen för minustecknet. I den sista satsen skär man ut den skiva som innehåller datumet.

Uppgift 5.7

Ändra det sista exemplet så att `b` kommer att innehålla födelsedagen enligt modellen *dd/mm,* dvs. `'14/03'`.

Uppgift 5.8

Skriv ett program som läser in en text. Programmet ska sedan ta bort alla blanka tecken (mellanslag) från texten och skriva ut resultatet. *Tips:* Man kan byta ut en deltext till en text med längden noll.

Uppgift 5.9

En *palindrom* är en text som blir samma när man läser den baklänges som när man läser den framlänges. Ett välkänt exempel är *ni talar bra latin.* När man läser hoppar man över alla mellanslag. Utöka programmet du skrev i förra uppgiften så att det undersöker om texten du läst in är en palindrom. *TipsT.* Ta först bort alla blanka tecken. *Tips2:* Skapa en skiva där tecknen ligger baklänges och jämför med den.

5.5 Datum och tid

I Python finns en standardtyp som heter `datetime`. Den kan man använda om man vill veta hur mycket klockan är eller vilket datum det

är. För att kunna använda typen `datetime` måste man först importera en modul som också heter `datetime`:

```
import datetime # måste vara med
```

Man skapar sedan en variabel av typen `datetime` genom att skriva:

```
dt = datetime.datetime.now() # datum och tid just
nu
```

(Lägg märke till att `datetime` ska skrivas två gånger.) Variabeln `dt` kommer då att innehålla både aktuellt datum och klockslag. För att välja ut datumet kan man skriva

```
d = dt.date() # datum i dag
```

Vill man hantera datumet som en text kan man göra om det till en `str` med hjälp av funktionen `str`:


```
dtext = str(d)
print(dtext) # skriver ut datum som 'åååå-mm-dd'
```

Variabeln `dtext` kommer då att innehålla aktuellt datum som en text.

Det går också att använda attributen `year, month` och `day` för att avläsa årets, månadens och dagens nummer från variabeln `dt` eller `d`:

```
år = dt.year # årets nummer
mån = dt.month # månadens nummer
dag = dt.day # dagens nummer
```

Är man i stället intresserad av klockslaget kan man hämta detta från `datetime`-variabeln `dt`:

```
t = dt.time() # aktuell tid
```

Variabeln `t` innehåller då timmar, minuter, sekunder och mikrosekunder. Man kan göra om klockslaget till en text med hjälp av funktionen `str`, men denna text kommer då också att innehålla mikrosekunderna. Är man inte intresserad av dessa kan man skära ut en skiva av texten:

```
ttext = str(t) [:8]
print(ttext) # skriver ut klockslag som 'tt-mm-ss'
```

Det går också att använda attributen `hour, minute` och `second` för att avläsa timmar, minuter och sekunder från variabeln `dt` eller `t`:

```
tim = dt.hour # timmar
min = dt.minute # minuter
sek = dt.second # dagens nummer
```

Uppgift 5.10

Skriv ett program som visar aktuellt datum och klockslag på detta sätt:

```
Dagens datum: aaaa-mm-dd
Klockan är: tt:mm:ss
```

5.6 Sammanfattning

Efter att ha läst detta kapitel bör du:

- veta hur man skapar variabler av typen `str` och hur kan skriver textliteraler,

- veta vad teckenkoder och Unicode är och hur man anger "konstiga" tecken som inte finns på tangentbordet,
- veta hur man indexerar och skär ut skivor i sekvenser, t.ex. texter,
- veta hur man kan jämföra sekvenser,

- veta hur man kan använda operatorn in för att löpa igenom sekvenser eller testa om ett element ingår i en sekvens,
- veta hur man kan använda operatorerna `+` och `*` för att bilda nya sekvenser,
- kunna använda funktionerna för typen `str` för att undersöka texter och bilda nya texter,
- veta hur man kan få reda på aktuell tid och dagens datum.

5.7 Övningar

5.1 Skriv ett program som läser in en mening, bestående av minst två ord. Programmet ska sedan visa ett meddelande där det dels talar om hur många tecken användaren skrev och dels talar om vilket som var det första resp. det sista ordet. Du får anta att det står (minst) ett blankt tecken mellan varje ord. Det kan också finnas blanka tecken före det första ordet och efter det sista.

5.2 Skriv ett program som läser in ett personnummer och skriver ut meddelandet: *Grattis!* om den aktuella personen har födelsedag. Personnummer anges med 10 siffror (utan minustecken).

5.3 Skriv ett program som läser in ett personnummer och avgör om personen är en man eller en kvinna. (Den näst sista siffran är udda för män och jämn för kvinnor.)

5.4 Skriv ett program som läser in en text och översätter den till det s.k. rövarspråket. I detta dubbleras alla konsonanter och ett 'o' placeras mellan de dubblerade konsonanterna. Vokaler och övriga tecken ändras inte. Om man t.ex. översätter `'Hej! Kom`

`in.'` till rövarspråket, får man `'Hohejoj! Kokomom inon.'.`

5.5 Skriv ett program som läser in en text som innehåller rövarspråk. Programmet ska översätta rövarspråket tillbaka till vanligt språk.

Du kan för enkelhets skull förutsätta att texten i innehåller korrekt rövarspråk. (Se övning 5.4.)

5.6 Ett *anagram* får man om man utgår från ett ord eller en mening och kastar om bokstäverna så att ett nytt ord eller en ny mening framträder. Av meningen C *är lurigt* kan man t.ex. bilda anagrammet *Curt Ärlig.* Skriv ett program som undersöker om en text innehåller ett anagram av en annan text. Det två texterna ska läsas in av programmet. *Tips:* Innan texterna jämförs kan man ta bort alla mellanslag. Därefter kan man gå igenom alla bokstäver i den första texten och för varje bokstav kontrollera att den finns lika många gånger i båda texterna.

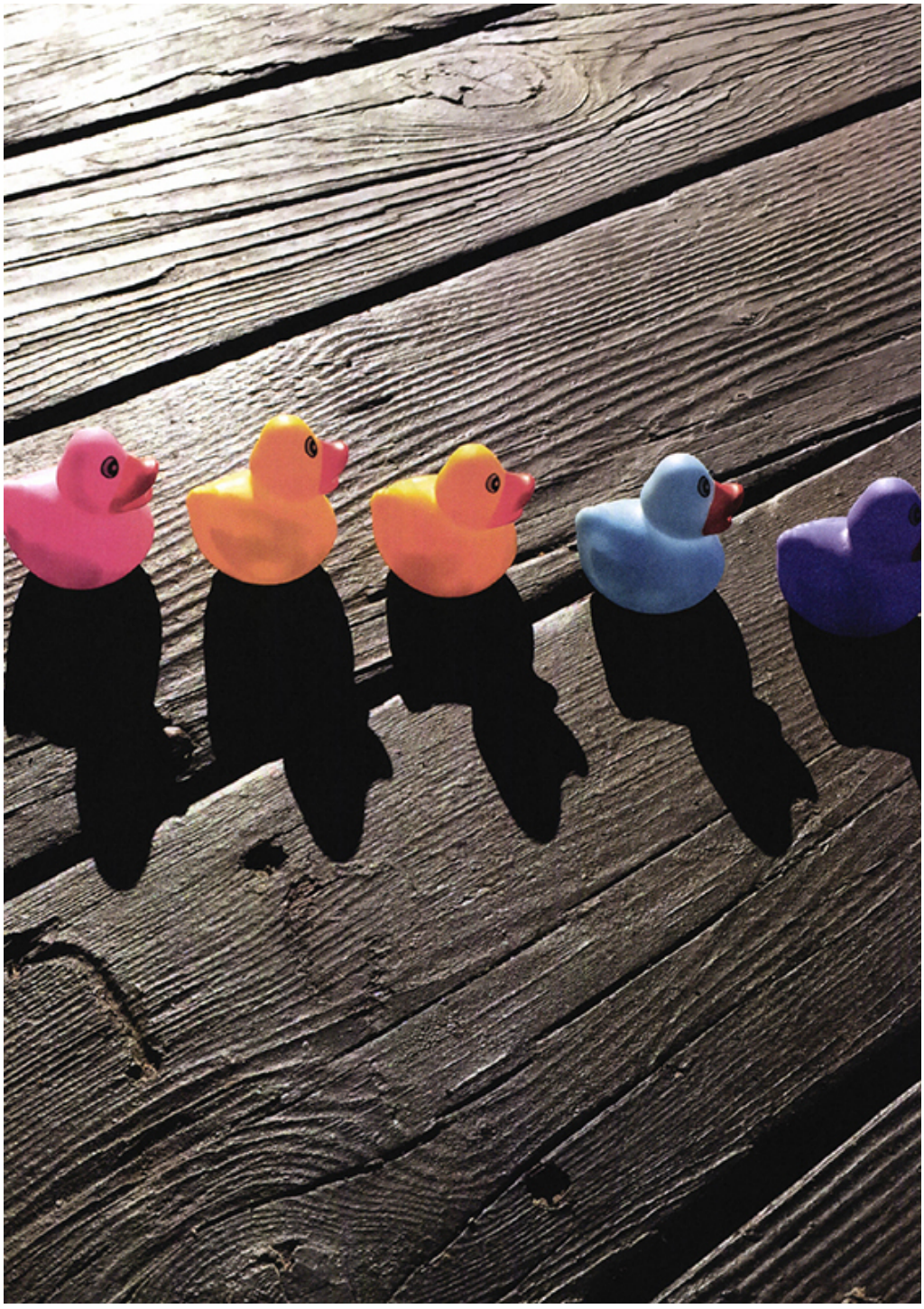
5.7 Vissa envisa användare av amerikanskt tangentbord brukade förr i tiden göra livet surt för sina vänner genom att inte skriva vanliga å, ä och ö i sina mejl. I stället skrev de konstiga bokstavskombinationer. Skriv ett program som läser in texten från ett sådant mejl och som skriver ut texten så att den blir läslig. Alla förekomster av bokstavskombinationerna aa, ae och oe ska översättas till å, ä respektive ö. (För enkelhets skull kan du ändra den inlästa texten så att den bara innehåller små bokstäver innan du gör översättningen.)

5.8 Skriv ett program som läser in två tidpunkter med formen *tt:mm.* Programmet ska visa hur många minuter det är mellan

de två tidpunkterna. *Tips:* Räkna om de båda tiderna till antalet minuter som gått sedan det aktuella dygnets start och gör en justering om den senare tidpunkten infaller under det följande dygnet.

[överkurs]
5.9 Ett personnummer, t.ex. 561231-4913, består av tio siffror. Efter de sex första siffrorna ska det finnas ett minustecken. Den näst sista siffran är udda för män och jämn för kvinnor. Den sista siffran är en kontrollsiffra. Den beräknas på följande sätt: De nio första siffrorna i personnumret multipliceras omväxlande med 2 och 1, den första med 2 den andra med 1 den tredje med 2 osv. *Siffrorna* i de värden man då får adderas. I personnumret ovan blir det
$(1 + 0) + 6 + 2 + 2 + 6 + 1 + 8 + 9 + 2 = 37$. (Observera att 10 räknas som $1 + 0$.) Kontrollsiffran bestäms sedan av att den summa man fått plus kontrollsiffran ska vara jämnt delbar med 10. I exemplet blir alltså kontrollsiffran lika med 3. Skriv ett program som läser in ett personnummer och undersöker om det är korrekt.

6 Listor och tupler



I Python finns flera olika standardtyper. Om man använder sådana kan man konstruera avancerade program utan att den kod man själv skriver behöver vara speciellt komplicerad. Du har redan stiftat bekantskap med typen `str`. Den tillhör den grupp av typer som man kan använda för att bilda *sekvenser.* I en sekvens ligger de ingående värdena, de s.k. *elementen* i en numrerad

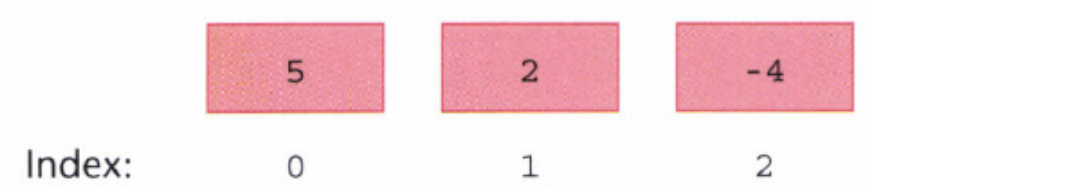
följd. Två andra slag av sekvenser är *listor* och *tupler.* Dessa typer ska vi diskutera i detta kapitel.

6.1 Yttre egenskaper för listor

6.1.1 Listor i allmänhet

En lista är en datasamling som innehåller ett godtyckligt antal element. En lista kan också vara tom. Det som är speciellt för listor är att elementen är *ordnade.* Man kan också säga att de förekommer i en viss sekvens. Varje element i listan har en viss plats och man kan därför numrera elementen i listan. Man kan säga att varje element har ett visst *index.* I figur 6.1 visas ett exempel på en lista. Observera att figuren inte säger något om hur listan är uppbyggd. Den visar bara att listan innehåller elementen 5, 2 och -4 och att de är numrerade från 0 till 2.

Figur 6.1 En lista.



Bilden visar en tabell med 2 rader och 4 kolumner. Första raden visar listan 5, 2, och -4. Andra raden visar Index: 0, 1, och 2.

I en lista är det för det mesta tillåtet att ändra och ta bort element och att skjuta in nya element var som helst.

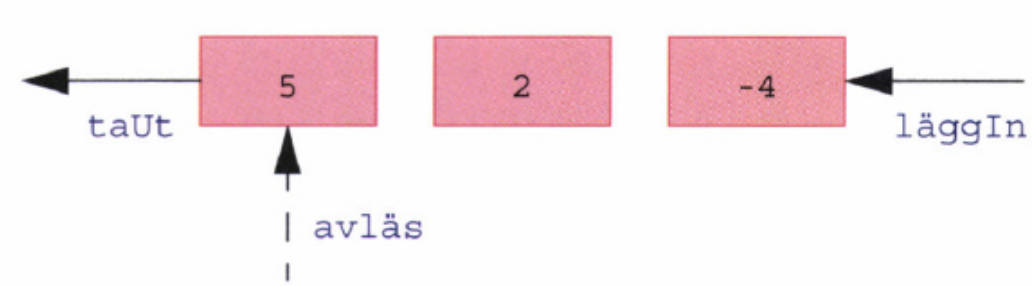
6.1.2 Köer

En *kö* är en lista i vilken nya element bara får läggas till i slutet på listan och element bara får tas ut i början. Man säger att en kö fungerar enligt

---

principen "först in först ut". Det brukar förkortas FIFO, *first in first out.* Det kan också vara tillåtet att avläsa värdet av det första elementet utan att ta bort det. I figur 6.2 visas principen för en kö.

Figur 6.2 En kö.



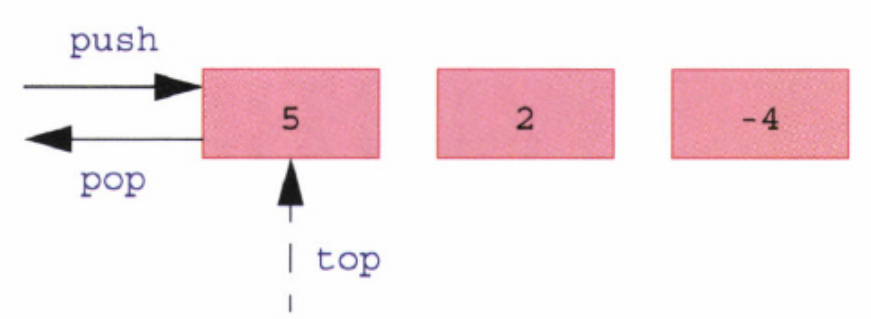
Bilden visar listan 5, 2, och -4. En pil riktad mot -4 visar var en variabel läggs in. En annan pil från 5 visar var variabeln tas ut. En streckad pil visar att värdet avläses vid 5.

Ibland har man köer i vilka det är tillåtet att lägga in och ta ut element i båda ändar. En sådan kö kallas en *deque* (double-ended queue).

6.1.3 Stackar

En *stack* är en lista i vilken man bara får lägga till och ta ut element i början på listan. En stack fungerar därför enligt principen "sist in först ut". Det förkortas LIFO, last *in first out.* Om du vill kan du tänka dig att det fungerar ungefär som en tallrikshög i en självservering. Man kan bara ta den översta tallriken. Operationerna för att lägga till respektive ta ut ett element brukar kallas *push* respektive *pop.* Ofta finns det också en operation med namnet *top* som avläser det översta elementet i stacken utan att ta bort det. I figur 6.3 visas en stack.

Figur 6.3 En stack.



Bilden visar listan 5, 2, och -4. En pil riktad mot 5 visar push medan en annan pil från 5 visar pop. En streckad pil visar att värdet 5 avläses som top.

6.2 Grundläggande operationer

I Python finns det två olika slags listor: vanliga *listor* och *tupler.* De standardtyper som man använder är `list` och `tuple`. Det viktigaste som skiljer dem åt är att i variabler av typen `list` är det tillåtet att ändra enskilda element, medan detta inte är tillåtet i tupler.

Man kan skapa en lista genom att räkna upp de ingående elementen:

```
la = [7, 14, 21, 28] # typen blir: list
ta = (5, 10, 15) # typen blir: tuple
```

Använder man hakparenteser, får listan typen `list` och använder man runda parenteser eller inga parenteser alls, får den typen `tuple`:

```
tb = 3, 6, 9, 12 # typen blir: tuple
```

Det går enkelt att skapa tomma listor och listor med bara ett element (s.k. *singleton).* Observera att för en tupel med bara ett element måste man ange ett extra kommatecken efter elementet.

```
l0 = [] # en tom lista
t0 = () # en tom tupel
l1 = ['ensam'] # en singleton
t1 = ('singel',) # en singleton
```

Värdena kan ha vilken typ som helst. De behöver inte vara konstanter:

```
k = 7
lb = [k, 2*k, 3*k, 4*k]
ls = ['ole', 'dole', 'doff']
```

Alla elementen behöver inte ens vara av samma typ:

```
tc = ('Erik', 30)
```

För både typen `list` och typen `tuple` gäller att man kan använda alla de operationer som vi beskrev i avsnitt 5.3. Man kan t.ex. indexera:

```
la[2] # blir: 21
ta[1] # blir: 10
tb[3] # blir: 12
```

och skära ut skivor:

```
lb[1:3] # blir: [14, 21]
ls[1:] # blir: ['dole', 'doff']
tb[:3] # blir: (3, 6, 9)
```

Ibland kan det vara praktiskt att använda multipel tilldelning:

```
namn, ålder = tc # namn = 'Erik', ålder = 30
```

Man kan undersöka om två listor är lika:

```
la == lb # blir: True
```

De båda listorna bör då vara av samma typ. Är de av olika typ, blir nämligen resultatet alltid `False`, även om de innehåller samma värden.

Man kan undersöka om en lista är "mindre" eller "större" än en annan:

```
ta < tb # blir: False
```

Operatorn `in` kan användas för att löpa igenom en lista. För att beräkna summan av elementen i listan `la` kan vi förstås använda standardfunktionen `sum` (se faktarutan på sidan 93), men vi kan också skriva:

$\mathrm{tot} = 0$

```
for e in la:
        tot = tot + e
print(tot) # skriver ut: 70
```

Lägg märke till att när man löper igenom en lista på detta sätt blir variabeln `e` en kopia av elementen i listan. Det betyder att om vi försöker ändra `e` så har detta ingen effekt på listan:

```
for e in la:
        e = 0 # Fel, fungerar inte!
```

Vill man ändra i listan kan man använda `range` och indexera:

```
for i in range(0, len(la)):
        la[i] = 0 # OK
```

Nu ändras listan `la` till `[0, 0, 0, 0]`.

Operatorn `in` kan också användas för att undersöka om ett visst element ingår i en lista.

```
'doff' in ls # blir: True
```

En speciell konstruktion som kallas *list comprehension* gör att man kan bilda en ny lista genom att utgå från en befintlig lista och utföra ett viss uttryck på alla elementen. Här är ett exempel:

```
tal = [1, 2, 3, 4, 5]
v = [x * x for x in tal] # v blir: [1, 4, 9, 16, 25]
```

Detta kan också kombineras med ett villkor, t.ex.

```
w = [x * x for x in tal if x > 2] # w blir: [9, 16, 25]
```

Funktionerna `len, min` och `max` som vi använt för texter kan också användas för listor:

```
len(la) # blir: 4
min(la) # blir: 7
max(la) # blir: 28
```

Funktionen `count` ger antalet gånger ett visst element finns i en lista:

```
tz = (0, 1, 0, 2, 3, 0)
tz.count(0) # blir: 3
tz[1:3].count(0) # undersöker en skiva av tz, blir: 1
```

Funktionen `index` kan användas för att söka efter ett visst element:

```
tz.index(3) # blir: 4
```

Operatorerna `+` och `*` och de utökade tilldelningarna `+=` och `*=` fungerar på samma sätt som för texter:

```
la = [7, 14, 21, 28]
lc = la + [35, 42] # lc blir: [7, 14, 21, 28, 35, 42]
lz = [0] * 5 # lz blir: [0, 0, 0, 0, 0]
lb = [4, 5]
la += lb # la blir: [7, 14, 21, 28, 4, 5]
lb *= 3 # lb blir: [4, 5, 4, 5, 4, 5]
```

Att skriva ut en lista är lätt. Man kan använda funktionen `print`:

```
print(lc)
print(ls)
```

Utskriften blir

```
[7, 14, 21, 28, 35, 42]
['ole', 'dole', 'doff']
```

Om man har en lista med element av typen `str` kan man använda funktionen `join`. Denna skapar en enda text som innehåller alla elementen i listan. Man anger själv vilket eller vilka tecken som ska finnas mellan elementen i texten. Här kommer ett par exempel:

```
print(';'.join(ls))
print(' '.join(ls))
```

Den första raden säger att listan `ls` ska skrives ut med ett semikolon mellan elementen och den andra att det ska vara två blanka tecken mellan elementen. Så här kommer utskriften att se ut:

```
ole;dole;doff
ole dole doff
```

106

I modulen `random` (se faktarutan på sidan 39) finns några funktioner som är användbara ihop med sekvenser. Man kan t.ex. avläsa ett slumpmässigt valt element med hjälp av funktionen `choice`:

[fullständigt program]

```
import random
e = random.choice(la)
```

Listor kan användas för att bilda *talföljder.* Som exempel ska du få se hur man kan bilda Fibonaccis berömda talföljd. I denna är varje tal i följden summan av de två föregående talen. Det första talet i följden har värdet 0 och det andra värdet 1. Talföljden användes av italienaren Fibonacci på 1200-talet när han försökte beskriva hur kaniner förökade sig. Fast då utgick han från att de första två värdena båda var 1. Här kommer ett program som skapar en lista med de första fibonaccitalen.

```
# Fibonacci
n = int(input('Hur många tal önskas? '))
f = [0,1]
for i in range(2, n+1) :
        f += [f[i-2] + f[i-1]] # lägg till nästa tal
print(f) # skriv ut alla talen
```

Vi lägger talen i listan `f`. Eftersom de två första talen i serien är givna till `0` och `1` initierar vi listan så att den innehåller dessa två tal. Sedan skriver vi en `for`-sats som lägger till de övriga elementen. Som du ser bildas varje nytt

element som summan av de två föregående. Det som ska stå till höger om operatorn `+=` måste vara en lista. Därför har vi skrivit hakparenteser runt uttrycket.

Uppgift 6.1

I en geometrisk talföljd får man ett tal i följden genom att multiplicera det föregående talet med en viss konstant `k`. Skriv ett program som konstruerar en geometrisk talföljd där konstanten `k` är lika med `3` och det första talet i följden är `2`. Talföljden ska läggas i en lista som ska skrivas ut. Låt den som kör programmet bestämma hur många tal som ska tas med i listan.

6.3 Inläsning till listor

I fortsättningen i detta kapitel kommer vi att diskutera typen `list` och inte typen `tuple`. Typen `list` är nämligen mer generell eftersom det går att ändra i en variabel av denna typ, vilket inte går för variabler av typen `tuple`. Vi börjar med att se hur man kan läsa in värden till en lista. Som exempel visar vi ett program som läser in flera ord och gör om dem till en lista. När man kör programmet kan det se ut så här:

```
Skriv ett antal ord: snart är det rast
Du skrev 4 ord
Det första var snart
Det sista var rast
```

Här kommer programtexten:

[fullständigt program]

```
# Läs ett antal ord
s = input('Skriv ett antal ord: ')
ordlist = s.split()
print(f'Du skrev {len(ordlist)} ord')
print('Det första var', ordlist[0])
print('Det sista var', ordlist[len(ordlist)-1])
```

Funktionen `input` ger en text som resultat. Variabeln `s` får därför typen `str`. Den röda raden är intressant. Där anropas funktionen `split`. Denna undersöker den inlästa texten `s` och skapar en lista där varje ord i `s` blir ett element i listan. När man anropar `split` utan argument förutsätter den att varje ord i texten omges med ett eller flera vita tecken. I exemplet har texten `'snart är det rast'` lästs in till `s`. Därför kommer `ordlist` att innehålla `['snart', 'är', 'det', 'rast']`.

Orden i den text man läser in kan avgränsas av något annat tecken än blanka, t.ex. ett komma- eller semikolontecken. Anta att den inlästa texten som finns i variabeln `s` är `'gul; blå; röd; vit'`. För att dela upp en sådan text till en lista kan man använda funktionen `split` och ange att semikolon ska vara avgränsare:

```
ls = s.split(';') # ls blir: ['gul', 'blå', 'röd',
'vit']
```

Avgränsaren behöver inte vara ett enda tecken. Anta t.ex. att den inlästa texten är `'small, medium, large'`. Då kan vi ange att avgränsaren består av två tecken, ett kommatecken och ett blankt tecken:

```
ls = s.split (', ') # ls blir: ['small', 'medium',
'large']
```

Något krångligare blir det om det kan finnas ett godtyckligt antal blanka tecken efter kommatecknen. Den inlästa texten kan t.ex. vara `'a, b, c, d'`. Då kan vi börja med att ta bort alla blanka tecken:

```
s = s.replace(' ', '') # s blir: 'a,b,c,d'
```

Sedan kan vi anropa `split` och ange kommatecken som avgränsare:

```
ls = s.split(',') # ls blir: ['a', 'b', 'c', 'd']
```

I alla exempel hittills där vi skrivit program som räknar ut något har vi läst in ett numeriskt värde i taget. Men om man använder sig av en lista kan man läsa flera värden på en gång. Som exempel skriver vi ett program där man ska läsa in ett antal tal och beräkna deras medelvärde.

[fullständigt program]

```
# Beräkna medelvärde, version 1
s = input(' Skriv ett antal tal: ')
ls = s.split()
sum = 0
for e in ls:
        sum = sum + float(e)
medel = sum / len(ls)
print(f'Medelvärdet är {mede:.2f}')
```

Så här kan det t.ex. se ut när man kör programmet. Man skriver in talen med blanka tecken emellan.

```
Skriv ett antal tal: 1.5 2.8 1.9 2.1
```

```
Medelvärdet är 2.07
```

När man i programmet har läst in det användaren skrivit till variabeln `s` skapar man en lista `ls` där varje element innehåller ett "ord" i texten. Om vi skrivit talen i exemplet kommer listan `ls` alltså att innehålla elementen `['1.5', '2.8', '1.9', '2.1']`.

Problemet är att varje element i `ls` har typen `str`. Vi måste alltså göra om dem till en numerisk typ, i det här fallet till typen `float`. Det gör vi inne i `for`-satsen. Detta är markerat med rött.

Ett annat alternativ är att bilda en ny lista utgående från listan `ls`. Vi kan använda *list comprehension* (se sidan 104):

```
talen = [float(e) for e in ls]
```

Här bildas varje element i den nya listan `talen` genom att `float` anropas för motsvarande element i listan `ls`. Vi får alltså en ny lista med element av typen `float`. Vi använder nu detta i en ny version av programmet:

[fullständigt program]

```
# Beräkna medelvärde, version 2
s = input(' Skriv ett antal tal: ')
ls = s.split()
talen = [float(t) for t in ls]
medel = sum(talen) / len(talen)
print(f'Medelvärdet är {medel:1.2f}')
```

Eftersom vi nu har en lista med `float` kan vi dessutom anropa standardfunktionen `sum` och slipper skriva därför skriva någon `for`-sats.

Skulle man i stället vilja läsa in ett antal heltal, ersätter man `float` med `int` på den röda raden.

Uppgift 6.2

Skriv ett program som läser in ett godtyckligt antal heltal. Programmet ska sedan undersöka hur många av de inlästa talen som är mindre än noll.

6.4 Operationer för typen `list`

En lista är ändringsbar. Vi kan därför ändra elementen i en lista med hjälp av indexering. Vi kan t.ex. skriva

```
la = [7, 14, 21, 28]
la[3] = 19 # la blir: [7, 14, 21, 19]
```

Det går också att ändra en skiva i en lista:

```
la[0:2] = [1, 2, 3] # la blir: [1, 2, 3, 21, 19]
```

Här har vi ersatt de första två elementen i `la` med tre nya element.

Förutom de operationer vi diskuterat hittills i detta kapitel och i avsnitt 5.3 finns det ett antal funktioner och funktioner som kan användas speciellt för listor. Dessa visas i faktarutan.

Fler funktioner för typen list

Förutom det som visas i denna ruta finns också alla de operationer som beskrivits i avsnitt 5.3.

I denna ruta betecknar `l` en lista av typen `list` och `k, m` och `n` heltal.

`list(sekv)` funktion som skapar en lista från sekvensen `sekv`

`l.append(e)` lägger till elementet `e` sist i `l`

`l.extend(l2)` lägger till listan `l2` sist i `l`

`l.insert(k,e)` skjuter in elementet `e` på plats nr `k` i listan `l`

`l.clear()` tar bort alla elementen i listan `l`

`dell[k]` tar bort elementet på plats nr `k` i listan `l`

`dell[n:m:k]` tar bort elementen i skivan `n:m:k` från listan `l`

`l.pop()` tar bort det sista elementet i listan `l` ger det borttagna elementet som resultat

`l.pop(k)` som ovan, men tar bort elementet på plats nr `k`

`l.remove(e)` tar bort första elementet som är lika med `e` från `l`

`l.copy()` ger en kopia av listan `l`

`l.reverse()` lägger elementen baklänges i listan `l`

`l.sort()` sorterar elementen i listan `l` som ändras

`sorted(sekv)` ger som resultat en sorterad lista; `sekv` ändras inte

Operationen `list` kan användas för att skapa en lista. Argumentet ska vara en annan sekvens, eller en följd av element som t.ex. genereras av funktionen `range`.

```
v = list() # v blir en tom lista: []
t = (1, 2, 3)
v = list(t) # v blir: [1, 2, 3]
s = 'abc'
v = list(s) # v blir: ['a', 'b', 'c']
v = list(range(0, 10, 2)) # v blir: [0, 2, 4, 6, 8]
v = list(range(−2, 2)) # v blir: [−2, −1, 0, 1]
```

Det går också att lägga till nya element med funktionen `append`:

```
la = [7, 14, 21, 28]
la.append(−2) # la blir: [7, 14, 21, 28, −2]
```

Vill man i stället lägga in ett nytt element på en viss plats kan man använda funktionen `insert`. Den har två argument. Det första argumentet är indexet och det andra argumentet det nya elementet.

```
la.insert(3, −9) # la blir: [7, 14, 21, −9, 28, −2]
```

Operatorn `del` tar bort enstaka element eller skivor:

```
del la[2] # la blir: [7, 14, −9, 28, −2]
del la[1:3] # la blir: [7, 28, −2]
```

Vill man avläsa ett element och ta bort det på samma gång kan man använda funktionen `pop`:

```
n = la.pop() # la blir: [7, 28] och n blir: −2
n = la.pop(0) # la blir: [28] och n blir: 7
```

Vill man ta bort alla elementen, kan man använda funktionen `clear`:

```
la.clear() # la blir: []
```

Man kan kopiera en hel lista med `copy`:

```
lx = [10, 11, 12]
ly = lx.copyO # ly blir: [10, 11, 12]
```

Funktionen `reverse` vänder en lista bak och fram:

```
ly.reverse() # ly blir: [12, 11, 10]
```

Uppgift 6.3

Skriv ett program som skapar en lista med 100 slumpmässiga heltal i intervallet 1 till 1 000. Programmet ska sedan skriva ut det minsta och det största av talen samt beräkna och skriva ut slumptalens medelvärde.

Uppgift 6.4

Skriv ett program som läser in en lista med heltal. Programmet ska sedan ta bort alla element från listan som är lika med noll. *Tips.* Tänk på att det kan finnas flera sådana element.

Som exempel på användning av listor ska du nu få se ett fullständigt program som jämför olika typer av kontantkort för smarta mobiler. Det läser för varje kort in kortets namn och priset per månad. När all information lästs in skriver programmet ut vilket kort som var billigast och priset per minut för detta kort. Här ser du ett exempel på hur det kan se ut när man kör programmet. Fyra olika typer av kontantkort jämförs.


```
Namn och pris för ett kort: Telemax 149
Namn och pris för ett kort: Blå fast 99
Namn och pris för ett kort: Pop student 149
Namn och pris för ett kort: Q fast 95
Namn och pris för ett kort:
Q fast är billigast
Kostnad: 95.00 kr/månad
```

Programmet ser ut på följande sätt. (Kommentarer och förklaringar får du efter programmet.)

[fullständigt program]

```
# Kontantkort, lägsta pris
namn = []
pris = []
while True:
        s = input('Namn och pris för ett kort: ')
        if s == '':
                break
        s = s.strip() # ta bort ev. blanka först och
sist
        i = s.rfind(' ') # sök index före priset
        namn.append(s[0:i]) # bilda skiva med namnet
```

```
        pris.append(float(s[i+1:])) # bilda skiva med
priset
m = min(pris) # sök lägsta pris
k = pris.index(m) # index för lägsta pris
print(namn[k] + ' är billigast')
print(f'Kostnad: {m:1.2f} kr/månad')
```

Två listor används. Listan `namn` innehåller namnen för de olika kontantkorten och listan `pris` kortens priser

`while`-satsen kommer att avslutas genom en `break`-sats. Därför står det `True` efter `while`. På varje varv läser man in namnet för ett kort och priset per månad för detta kort. Allt skrivs på en rad av användaren. Programmet läser in raden till variabeln `s` som får typen `str`. När det inte finns fler kort att jämföra markerar användaren detta genom att bara trycka på Enter. Då blir den inlästa raden tom. Programmet testar detta. Om raden är tom avslutas `while`-satsen med `break`.

För varje inläst rad tar programmet först bort eventuella inledande och avslutande blanka tecken. Därefter letar det reda på indexet `i` för det sista blanka tecknet. Det är detta tecken som står omedelbart före priset. Kortets namn finns i en skiva av den inlästa texten som slutar före tecknet på plats i. Denna skiva läggs till som ett nytt element i slutet av

listan namn. På motsvarande sätt skär man sedan ut den skiva av texten som innehåller priset. Denna skiva börjar i index

$$i + 1$$

. Eftersom denna skiva är en text måste den göras om till ett numeriskt tal av typen `float` innan man lägger till priset sist i listan `pris`. När `while`-satsen avslutats finns det lika många element i listorna `namn` och `pris`.

Att hitta det lägsta priset `m` i listan `pris` är enkelt. Vi använder standardfunktionen `min`. För att få reda på indexet `k` för det billigaste kortet söker vi sedan efter värdet `m` med hjälp av funktionen `index`. När vi har

indexet `k` kan vi avslutningsvis skriva ut namn och pris för kortet med index `k`, det billigaste kortet.

Uppgift 6.5

[överkurs]

Skriv en ny version av programmet som räknar ut det billigaste kontantkortet. Använd funktionen `rpartition` (se faktarutan på sidan 94) i stället för funktionerna `strip` och `rfind`.

6.5 Listor och referenser

Vill man kopiera ett värde från en enkel variabel till en annan enkel variabel, är det mycket lätt. Man använder förstås en vanlig tilldelning:

```
x = 5.75
y = x # vårdet 5.75 kopieras från x till y
```

Men när man använder listor är det inte riktigt lika enkelt. Tänk dig att vi har skapat två listor, `v` och `w`:

```
v = [1.5, 2.8, 4.3]
w = [0, 0, 0]
```

Vi provar nu om det går att göra en tilldelning från `v` till `w`:

```
w = v # tillåtet, men fungerar nog inte som du tänkt
```

Det verkar gå bra. Om vi skriver ut listan `w` får vi precis som väntat:

```
[1.5, 2.8, 4.3]
```

Samma värden skulle vi fått, om vi i stället hade skrivit ut listan `v`. Det verkar också rimligt eftersom vi inte ändrat något i `v`. Men om vi nu gör en ändring i `w`:

$$\mathtt{w}[1] = 9.9$$

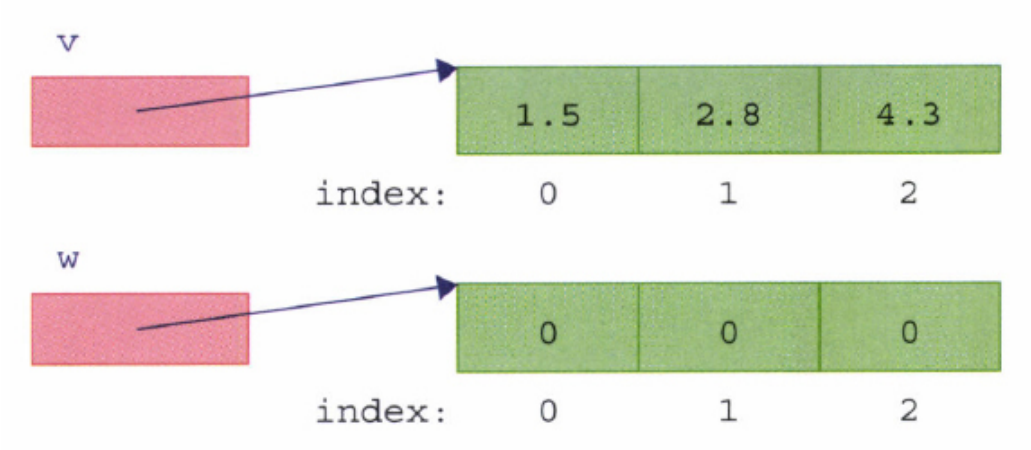
Då händer något konstigt. Det ser vi om vi skriver ut `v` (alltså inte `w`):

```
[1.5, 9.9, 4.3]
```

Vi ändrade alltså ett av elementen i `w`, men även motsvarande element i `v` ändrades! Lika konstigt hade det blivit om vi i stället hade ändrat ett av elementen i `v`. Då hade också motsvarande element i `w` ändrats.

Det bästa sättet att förklara hur detta hänger ihop är att använda en bild. Studera figur 6.4. I själva verket är variablerna `v` och `w` *referenser,* pekare, till själv listorna såsom visas i figuren.

Figur 6.4 Listor före tilldelning.



Bilden visar först variabeln v som pekar på listan 1.5, 2.8, och 4.3, vilka har index 0, 1, respektive 2. Sedan visas variabeln w som pekar på listan 0, 0, och 0, vilka också har index 0, 1 respektive 2.

I figuren visas hur det ser ut precis innan vi gör tilldelningen från `v` till `w`. När Vi nu gör tilldelningen

$$\mathrm{w} = \mathrm{v}$$

kommer det att se ut som i figur 6.5.

Figur 6.5 Listor efter tilldelning till listvariabel.

Bilden visar först variabeln v som pekar på listan 1.5, 2.8, och 4.3, vilka har index 0, 1, respektive 2. Sedan visas variabeln w som också pekar på den första listan och inte på 0, 0, och 0 (index 0, 1 respektive 2).

Som du ser har variabeln `w` ändrats så att den pekar på *samma* lista som `v`. När man gör en tilldelning till en listvariabel ändras nämligen bara själva pekaren, inte det den pekar på. När man gör en tilldelning från en listvariabel till en annan kommer därför pekaren att kopieras. I figur

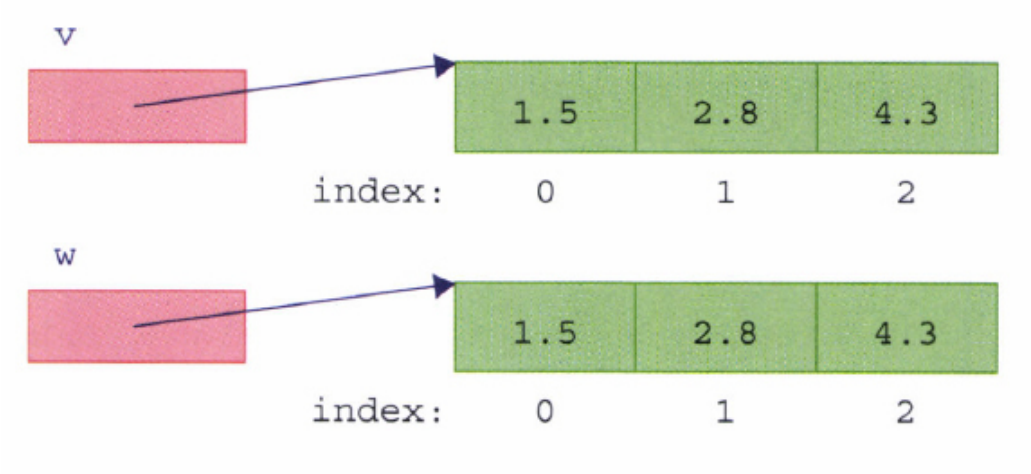
6.5 har pekaren i variabeln `v` kopierats till `w`. Detta betyder att `v` och `w` efter tilldelningen pekar på *samma* lista. Här har du förklaringen till att ändringar i `w` påverkar `v` och tvärtom.

Men hur gör man, om man verkligen vill kopiera själva *listan* från `v` till `w`? Ett sätt är att kopiera en komponent i taget i en snurra. Men vi har faktiskt redan sett en enklare lösning; vi använder funktionen `copy`:

```
w = v.copy()
```

Om man gör på detta sätt kommer det i stället att se ut som i figur 6.6.

Figur 6.6 Listor efter anrop av copy.



Bilden visar först variabeln v som pekar på listan 1.5, 2.8, och 4.3, vilka har index 0, 1, respektive 2. Sedan visas variabeln w som pekar på den egna listan 1.5, 2.8, och 4.3, vilka också har index 0, 1 respektive 2.

6.6 Flerdimensionella listor

I de listor du sett hittills sett ligger elementen i en enda lång rad. Men data presenteras ofta i form av tabeller. Ett exempel visas i tabell 6.1.

Tabellbeskrivning

Tabellen har 8 rader och 7 kolumner. Kolumnrubrikerna visar Borås, Göteborg, Luleå, Malmö, Norrköping, Stockholm, och Umeå.

Tabell 6.1 En avståndstabell.

	Borås	**Göteborg**	**Luleå**	**Malmö**	**Norrköping**	**Stockholm**	**Umeå**
Borås	0	64	1228	284	253	411	962
Göteborg	64	0	1249	273	317	475	983
Luleå	1228	1249	0	1476	1022	906	266
Malmö	284	273	1476	0	458	615	1210
Norrköping	253	317	1022	458	0	161	756
Stockholm	411	475	906	615	161	0	640
Umeå	962	983	266	1210	756	640	0

Med hjälp av en enkel lista hade man kunnat beskriva en rad eller en kolumn i tabellen, men inte hela tabellen. För att beskriva data av detta slag kan man i stället använda sig av listor där varje element i sin tur är en lista. Vi kan kalla en sådan lista en *flerdimensionell lista.* Speciellt intressanta är flerdimensionella listor med två dimensioner.

Man kan skapa en tvådimensionell lista på samma sätt som man skapar en vanlig lista, men skillnaden är att de element man räknar upp ska vara listor, inte enkla element. Här kommer ett exempel.

```
m = [[1, 2], [3, 4], [5, 6]]
```

Variabeln `m` innehåller en lista med tre element. Varje element är i sin tur en lista med två element.

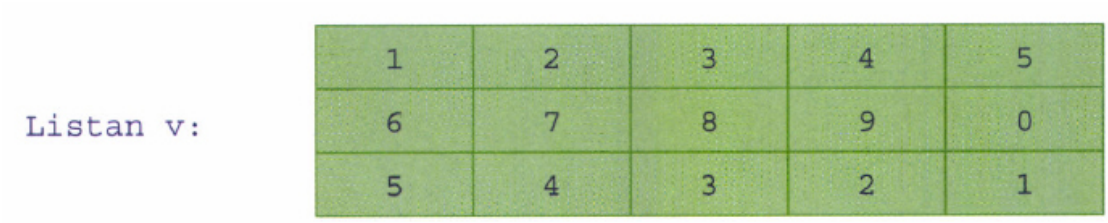
När det gäller tvådimensionella listor brukar man ofta uppfatta dem som en rektangulär uppställning med data. En sådan taluppställning brukar inom matematiken kallas en *matris.* Man säger att man har ett visst antal *rader* och ett viss antal *kolumner.* Antalet rader och kolumner behöver inte vara lika. Listan `m` har tre rader och två kolumner.

Här kommer ett exempel till, där vi skrivit satsen så att raderna och kolumnerna framgår tydligare.

```
v = [[ 1, 2, 3, 4, 5],
     [6, 7, 8, 9, 0]
     [5, 4, 3, 2, 1]]
```

Från avsnitt 3.4 minns du säkert att man måste skriva tecknet `\` sist på raden om man vill att en sats ska fortsätta på nästa rad. Men det finns ett praktiskt undantag från denna regel och det är när man har uttryck som omges av någon form av parenteser, t.ex. `[]`. Därför är det tillåtet att skriva såsom vi gjort här. Man kan tänka sig att variabeln `v` efter detta ser ut som i figur 6.7.

Figur 6.7 En tvådimensionell lista.



Bilden visar listan v som en tabell med tre rader och fem kolumner. Första raden visar siffrorna 1–5, andra raden visar 6–9 samt 0, och sista raden visar 5–1.

För att skapa en tvådimensionell lista med namnet `t` som innehåller informationen i avståndstabellen på sidan 115 kan man skriva:

```
t = [[ 0, 64, 1228, 284, 253, 411, 962],
       [64, 0, 1249, 273, 317, 475, 983],
       [1228, 1249, 0, 1476, 1022, 906, 266],
       [284, 273, 1476, 0, 458, 615, 1210],
       [253, 317, 1022, 458, 0, 161, 756],
       [411, 475, 906, 615, 161, 0, 640],
       [962, 983, 266, 1210, 756, 640, 0]]
```

I praktiken använder man sällan listor med fler än två dimensioner, men vill man ha listor med flera dimensioner är det bara att generalisera det du just sett. Vill man t.ex. ha en tredimensionell lista kan man låta varje listelement i kolumnerna i sin tur också vara listor.

Precis som i vanliga listor använder man *index* för att ange vilket av elementen man vill komma åt. Man anger då normalt lika många index som antalet dimensioner. I en tvådimensionell lista anger det första indexet radens nummer och det andra kolumnnumret. Även här sker indexeringen från noll, vilket betyder att den översta raden är rad nr 0 och att den vänstra kolumnen är kolumn nr 0. Här kommer några exempel på hur man kan tilldela värden till olika element i den tvådimensionella listan `v` i figur 6.7:

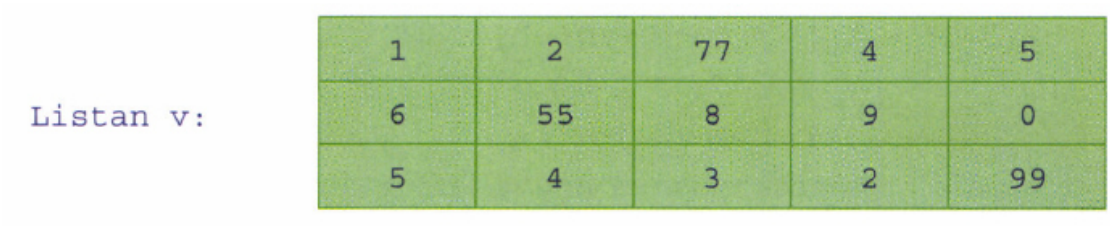
$$\mathrm{v}[0][2] = 77$$

$$\mathrm{v}[1][1] = 55$$

$$\mathrm{v}[2][4] = 99$$

Hur listan `v` ser ut efter att man utfört dessa satser visas i figur 6.8.

Figur 6.8 En tvådimensionell lista efter tilldelningar.



Bilden visar listan v som en tabell med tre rader och fem kolumner. Första raden visar siffrorna 1, 2, 77, 4, och 5; andra raden visar 6, 55, 8, 9, och 0, och sista raden visar 5, 4, 3, 2, och 99.

Naturligtvis behöver inte indexen vara konstanta värden. Man hade t.ex. fått samma resultat om man hade skrivit:

```
i, j, k = 0, 1, 4 # multipel tilldelning
V[i] [2*j] = 77
v[j] [j] = 55
v[int(k/2)] [k] = 99
```

Observera att indexen måste vara heltal.

Precis som när det gäller vanliga listor får man vara försiktig så att man inte indexerar utanför listornas gränser och t.ex. försöker ändra på rad nr 3 eller kolumn nr 5 i listan `v`.

Det är enkelt att skriva ut en tvådimensionell lista. Satsen `print (v)` ger t.ex. utskriften

```
[[1, 2, 77, 4, 5], [6, 55, 8, 9, 0], [5, 4, 3, 2, 99]]
```

Vill man ha en snyggare utskrift där man ser rader och kolumner tydligare kan man löpa igenom alla elementen och skriva ut dem ett och ett. Då är det naturligt att använda två nästlade `for`-satser. Följande programrader skriver t.ex. ut alla elementen i listan `v`:

```
for i in range(0, len(v)):
        for j in range(0, len(v [i])):
                print(f'{v[i] [j] : 3} ', end='') #
inget radbyte här
        print() # ny rad
```

Utskriften blir:

```
1  2 77  4  5
6 55  8  9  0
5  4  3  2 99
```

Den första `for`-satsen (den yttersta) löper tre varv, ett varv för varje rad. På första varvet har räknaren `i` värdet 0. Det första som händer på det första varvet är att den inre `for`-satsen utförs. Denna löper då totalt fem varv, ett varv för varje kolumn, och räknaren `j` kommer att löpa från 0 till 4. På varje inre varv skrivs värdet av `v[i][j]` ut. På det första inre varvet skrivs alltså `v[0][0]` ut, på det andra `v[0][1]` osv. På det sista inre varvet skrivs `v[0][4]` ut. När den inre `for`-satsen är klar anropas `print` utan argument. Då kommer en ny rad att påbörjas i utskriften.

På det andra varvet i den yttre `for`-satsen upprepas allt, fast räknaren `i` har nu värdet 1. Den inre `for`-satsen kommer att löpa fem varv och elementen på den andra raden i `v` kommer att skrivas ut. På det tredje och sista varvet i den yttre `for`-satsen upprepas allt ytterligare en gång. Den inre `for`-satsen löper fem varv och elementen på den sista raden i listan `v` kommer att skrivas ut.

I stället för att använda `range` och indexera kan man löpa igenom listan `v` på följande sätt, där variabeln `r` löper igenom raderna och variabeln `k` kolumnerna:

```
for r in v:
        for k in r:
                print(f'{k:3}', end=' ')
        print()
```

Det går förstås att skapa en tvådimensionell lista dynamiskt, utan att räkna upp alla elementen. Som exempel visas här ett program som skapar och skriver ut en multiplikationstabell för tal upp till tio gånger tio. För varje värde på `i` och `j` ska elementen på rad nummer `i` och kolumn nummer `j` i tabellen innehålla värdet

$$(i + 1) * (j + 1)$$

. (Indexering sker ju från noll.)

[fullständigt program]

```
# Skapa en 10 X 10 multiplikationstabell
a = []
for i in range(0, 10):
        a.append([]) # skapa en ny tom rad
        for j in range(0, 10):
```

```
                a[i] .append((i+1)*(j+1)) # lägg till
elementen
print(a)
```

Först skapas en tom lista med namnet `a`. På varje varv i den yttre `for`-satsen börjar man med att lägga till en ny tom lista som sista element i `a`. Eftersom den yttre `for`-satsen löper tio varv kommer alltså listan a att få tio element. I den inre `for`-satsen lägger man till de element som ska ligga i den lista som motsvarar rad nummer `i`.

Uppgift 6.6

Utskriften från programmet som skapar en multiplikationstabell blir inte så snygg. Dessutom vill man kanske inte alltid ha en tabell för tal upp till tio. Skriv en ny version av programmet där man låter den som kör programmet bestämma vilket som ska vara den övre gränsen i tabellen. Ändra också utskriften så att varje rad i tabellen hamnar på en egen rad i utskriften.

Vill man låta användaren av programmet läsa in de värden som ska finnas i en tvådimensionell lista kan man köra följande programrader.

```
# Inläsning till en matris
print('Skriv en matris rad för rad.'
        'Avsluta med en tom rad.')
m = []
while True:
        s = input('? ')
        if s == '':
                break
        ls = s.split() # lista med texter
```

```
        rad = [float(e) for e in ls] # lista med float
        m.append(rad) # ny rad i matrisen
```

Först skapas en tom lista. För varje rad användaren skrivit in anropar vi sedan funktionen `split` för att skapa en lista `ls` av elementen på den inlästa raden. Listan `ls` kommer att innehålla elementen som text, men här vill vi att matrisen ska innehålla numeriska värden. Därför skapar vi en ny lista med namnet `rad` som innehåller element av typen `float`. Sist på varje varv läggs sedan listan `rad` in som nytt sista element i listan `m`. De tre sista programraderna hade kunnat skrivas mer kompakt, men det hade troligen blivit mer svårläsligt.

```
m. append ( [float (e) for e in s.split()])
```

Uppgift 6.7

I programraderna du nyss sett förutsätts att användaren skriver in värdena till matrisen med blanka tecken mellan. Ändra i programmet så att det fungerar om användaren i stället skriver kommatecken mellan värdena.

Alla raderna i en tvådimensionell lista behöver inte vara lika långa. I avståndstabellen på sidan 117 är det t.ex. onödigt att ha med samma avstånd två gånger. Vi kan i stället skriva:

```
t = [[0],
[64, 0],
[1228, 1249, 0],
[ 284, 273, 1476, 0],
[ 253, 317, 1022, 458, 0],
[ 411, 475, 906, 615, 161, 0],
[ 962, 983, 266, 1210, 756, 640, 0]]
```

Uppgift 6.8

Skriv en ny version av programmet i övning 6.6. I den nya versionen ska du låta multiplikationstabellen vara triangulär, så att den första raden har längden 1, den andra längden 2 osv. Då behöver de olika värdena inte upprepas i tabellen. Du ska fortfarande låta användaren ange hur stor tabellen ska vara, dvs. hur många rader den ska ha.

6.7 Exempel – Mandatfördelning

Här kommer ett lite större exempel som illustrerar hur mandatfördelning går till i svenska val. När ett val har skett ska platserna, de s.k. mandaten, i riksdagen, i ett landsting eller i en kommun fördelas på ett så rättvist sätt som möjligt. Man utgår då förstås från hur många röster de olika partierna har fått i valet. Man börjar med att räkna bort röster från små partier som inte uppnått den undre gränsen. Denna gräns är olika för val på olika nivåer. (För riksdagen är den t.ex. 4 %.) När detta har skett tillämpas en algoritm som kallas den *jämkade uddatalsmetoden* för att bestämma hur många mandat de olika partierna ska få.

Du ska nu få se ett program som använder denna metod. Programmet läser först in hur många mandat som ska fördelas och sedan hur många röster de olika partierna har fått. Som resultat visar programmet hur många mandat de olika partierna får. Så här kan det se ut om man kör programmet och ger antalet röster i riksdagsvalet 2018 som indata. (Du kan själva lista ut vilka de olika parterna är.)

```
Antal mandat? 349
Spärr för småpartier. Procent? 4
Antal röster för partierna? 1284698 557500 355546
```

```
4094781830386 518454 285899 1135627 29665 69472
Mandatfördelning:
[70, 31, 20, 22, 100, 28, 16, 62, 0, 0]
```

(Antalet röster skrivs egentligen på en rad när man kör programmet, men den får inte plats här i boken.)

Beräkningen går till på följande sätt: Först divideras antalet röster med 1.2 för alla partier. De tal man då får fram kallas jämförelsetal. Man fördelar sedan mandaten ett och ett till det parti som för tillfället har det

---

största jämförelsetalet. Detta upprepas tills alla mandaten har fördelats. Varje gång ett parti tilldelats ett mandat beräknas ett nytt, lägre, jämförelsetal för detta parti. Första gången ett parti får ett mandat får man fram det nya jämförelsetalet genom att dividera antalet röster partiet fått med 3, andra gången med 5, tredje gången med 7 osv. Man kan alltså använda följande formel, där *t* är det nya jämförelsetalet, *r* antalet röster och *m* det antal mandat partiet hittills fått:

$$t = {}^{r}\!\Big/\!_{(2m+1)}$$

I programmet används tre listor: `röster` för att hålla reda på antalet röster de olika partierna fått, `jfrtal` för att lägga jämförelsetalen i och `mandat` för att hålla reda på hur många mandat de olika partierna fått.

[fullständigt program]

```
# Mandatfördelning
antalMandat = int(input('Antal mandat? '))
spärr = int(input('Spärr för småpartier. Procent? '))
spärr /= 100 # räkna om från procent
s = input('Antal röster för partierna? ')
röster = [int(x) for x in s.split()) # antal röster
# Beräkna totala antalet röster
```

```
tot = sum(röster)
# Initiera listorna mandat och jfrtal
mandat = [0] * len(röster) # antal mandat hittills
jfrtal = []
for i in range(0, len(röster)):
        if röster [i] / tot < spärr:
                röster [i] =0 # ta bort småpartier
        jfrtal.append(röster[i] / 1.2) # första
jämförelsetal
# Dela ut mandaten
for i in range(0, antalMandat):
        m = max(jfrtal) # det största jämförelsetalet
        p = jfrtal.index(m) # har parti nr p
        mandat[p] += 1 # ge mandatet till parti nr p
        # beräkna ett nytt jämförlsetal för parti nr p
        jfrtal[p] = röster [p] / (2*mandat[p]+1)
print('Mandatfördelning:')
print(mandat)
```

Summan av alla rösterna beräknas med standardfunktionen `sum`. Summan används sedan i den andra `for`-satsen när man testar om några partier har fått så få röster att de hamnar under spärren. I så fall sätts

---

deras röster till noll så, att de inte kan få några mandat i de följande beräkningarna.

I den sista `for`-satsen fördelas ett mandat per varv. Detta pågår tills alla mandaten är slut. På varje varv söker man efter det parti som har störst jämförelsetal. När detta parti fått ett nytt mandat räknar man om partiets jämförelsetal.

6.8 Sammanfattning

Efter att ha läst detta kapitel bör du:

- kunna ange de yttre egenskaperna för listor, köer och stackar,
- känna till hur man skapar listor och tupler i Python,
- kunna använda de generella operationerna för sekvenser, t.ex. indexering och skivor, på listor,
- kunna använda funktioner som är speciella för listor,
- kunna löpa igenom alla elementen i en lista,
- veta hur man kan läsa in värden till en lista,
- veta hur man kan skriva ut en lista,
- kunna skapa och använda tvådimensionella listor.

6.9 Övningar

6.1 Skriv ett program som läser in ett antal heltal och som skriver ut dem i samma ordning som de lästes in. Vid utskriften ska ett visst tal skrivas ut bara en gång. Om talet skrivits ut tidigare, ska det alltså inte skrivas igen. Om t.ex. följande tal läses in:

```
45 77 -22 3 45 0 21 -1 3
```

så ska programmet ge utskriften:

```
45 77 -22 3 0 21 -22
```

6.2 Skriv ett program som läser in ett antal temperaturer som uppmätts vid samma tidpunkt på olika platser. Mätstationerna är

numrerade från 1 och uppåt och programmet läser in temperaturerna i nummerordning (mätstation 1 först osv.). Programmet ska först skriva ut medelvärdet av alla temperaturerna. Det ska sedan skriva ut alla temperaturer som är högre än medelvärdet. För varje sådan temperatur ska programmet också skriva ut mätstationens nummer.

6.3 Om man använder operatorn `==` för att jämföra en variabel av typen `list` med en variabel av typen `tuple` blir resultatet alltid `False`, oberoende av vad listan och tupeln innehåller. Skriv ett program som jämför en lista och en tupel på ett annat sätt. Listan och tupeln ska läsas in till programmet innan jämförelsen görs. Programmet ska ange om listan och tupeln är lika eller inte. Ditt program ska uppfatta dem som lika om de är lika långa och motsvarande element är lika.

6.4 Skriv ett program med som beräknar den s.k. *medianen* för ett antal uppmätta värden. Programmet ska läsa in de uppmätta värdena till en lista. Det ska finnas lika många värden som är större än medianen som det finns värden som är mindre än medianen. Om värdena är sorterade och det finns ett ojämnt antal tal är medianen talet i mitten. Om det finns en jämnt antal tal är medianen medelvärdet av de två mittersta talen.

[överkurs]
6.5 Skriv ett program med som beräknar den s.k. *standardavvikelsen* för ett antal tal. Programmet ska läsa in talen till en lista. Formeln för standardavvikelse är följande (x_i betecknar talen, *n* antalet tal och μ medelvärdet av talen):

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

6.6 Man brukar säga att en matris `a` är *symmetrisk,* om den har lika många rader som kolumner och om det för varje `i` och `j` gäller att `a[i][j]==a[j][i]`. Skriv ett program som läser in en matris till en tvådimensionell lista och som undersöker om matrisen är symmetrisk eller inte.

6.7 En magisk fyrkant är en taluppställning med lika många rader som kolumner. Summorna av värdena i varje rad, kolumn och diagonal är lika. Ett exempel är uppställningen:


```
16  9  2  7
6  3 12 13
11 14  5  4
1  8 15 10
```

Skriv ett program som avgör om en taluppställning är en magisk fyrkant eller inte. Funktionen ska läsa in taluppställningen till en tvådimensionell lista.

6.8 En lärare brukar ha ett antal prov varje termin. Vid läsårets slut vill han, för att kunna sätta betyg, göra en sammanställning av elevernas resultat. Skriv ett program som läser den information läraren skriver in. För varje elev ska programmet läsa in två rader. På den första raden står elevens namn och på den andra elevens resultat på vart och ett av de olika proven.
Elevernas namn ska sparas i en vanlig, enkel lista inne i programmet. Provresultaten ska sparas i en tvådimensionell lista. Programmet ska sedan, för var och en av eleverna skriva ut elevens namn, följt av hans eller hennes genomsnittspoäng på de olika proven. Programmet ska slutligen, för vart och ett av proven, skriva ut genomsnittspoängen för alla elever.

6.9 Inom matematiken arbetar man med begreppet vektorer. Med längden av en vektor $(v_1, v_2, \ldots v_n)$ menas emellertid där inte antalet element i vektorn. Längden av en vektor definieras i stället av formeln:

$$l = \sqrt{v_1^2 + v_2^2 + \ldots + v_n^2}$$

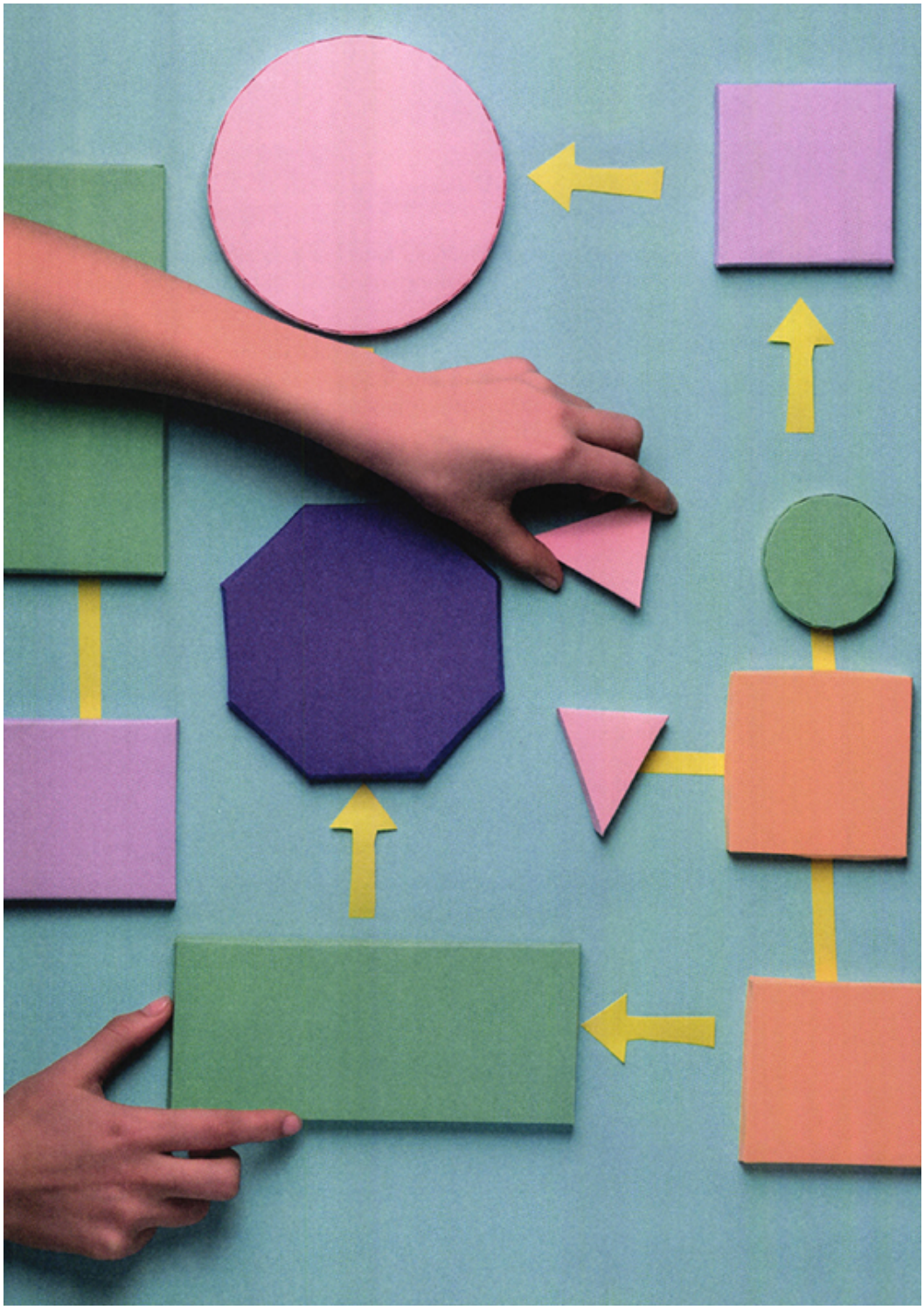
Skriv ett program som läser in en vektor till en lista och beräknar den matematiska längden av vektorn.

6.10 Inom matematiken säger man att två vektorer $(u_1, u_2, \ldots u_n)$ och $(v_1, v_2, \ldots v_n)$ är *ortogonala* om följande summa är lika med noll.

$$\sum_{k=1}^{n} u_k v_k$$

Skriv ett program som läser in två vektorer till två listor och avgör om vektorerna är ortogonala.

7 Algoritmer



Detta kapitel handlar om algoritmer, hur de kan uttryckas, hur de kan byggas upp och hur de kan utnyttjas vid programmering. Du får bl.a. se ett exempel där vi också visar hur en algoritm kan förbättras.

7.1 Vad är en algoritm?

En algoritm är en exakt beskrivning av hur man löser ett visst problem. Den innehåller ett antal instruktioner som steg för steg talar om vad man ska göra för att nå målet. Man tänker kanske i första hand på algoritmer för att lösa matematiska problem. Men algoritmer kan användas i alla möjliga sammanhang; i själva verket stöter man ofta på algoritmer. Ett exempel är matlagningsrecept. Där är problemet att laga en viss maträtt och algoritmen ger lösningen på problemet. Ett annat exempel är monteringsanvisningar och bruksanvisningar av olika slag.

När man ska skriva ett program för att lösa ett problem är det bra om man i förväg tänkt igenom hur problemet ska lösas. I programkoden är det så många detaljer som måste vara rätt (parenteser, apostrofer, indenteringar, etc.). Man kan då lätt fastna i detaljerna och tappa bort själva problemet. Det kan därför vara bra att skriva ner hur programmet ska fungera innan man börjar programmera. Man kan med andra ord skriva en algoritm.

7.2 Pseudokod och strukturdiagram

Algoritmer kan uttryckas på många olika sätt. Ett sätt är att använda vanligt naturligt språk. Då får man s.k. *pseudokod.* Man kan också använda bilder och symboler. Du har kanske sett monteringsanvisningar för möbler från IKEA. Ett annat vanligt sätt är att använda s.k. *strukturdiagram,* eller *flödesscheman* som de också kallas.

Algoritm

Detaljerad beskrivning av hur ett visst problem ska lösas, steg för steg.

Här kommer ett exempel. På sidan 63 läste vi in ett positivt heltal `n` och beräknade summan

$$1 + 2 + 3 \ldots + \mathrm{n}$$

. Om man skriver algoritmen i pseudokod kan den t.ex. se ut så här:

Läs in ett värde till `n`*.*

Använd en variabel `summa` *som sätts till 0 och en räknare* `k` *som sätts till 1.*

Upprepa följande två rader så länge som `k` *är mindre än eller lika med* `n`*:*

Öka `summa` *med* `k`*.*

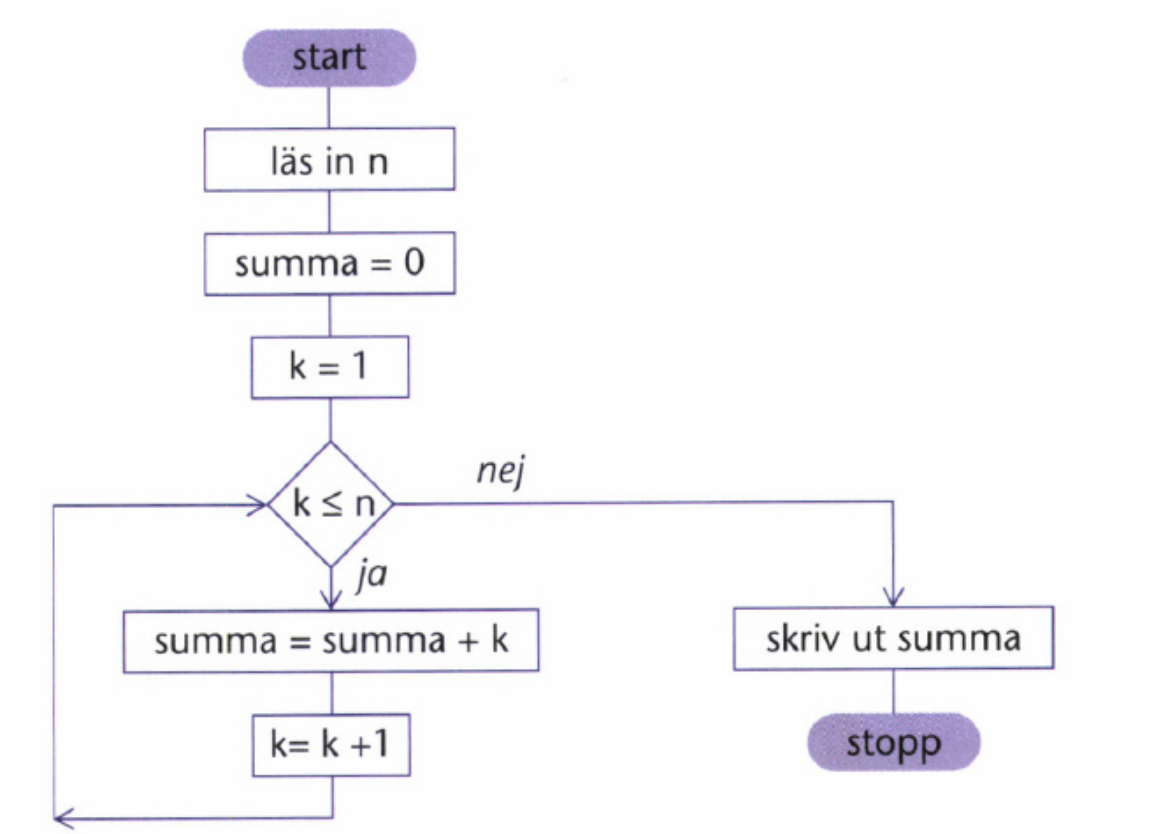
Öka `k` *med ett.*

Skriv ut `summa` *som resultat.*

Man kan ganska fritt uttrycka hur beräkningen ska gå till. Det finns inga speciella regler, men man måste vara tydlig så att det inte blir några missförstånd. Fördelen med att använda pseudokod är att man kan strunta i detaljerna och koncentrera sig på hur själva problemet ska lösas. Man kan senare översätta pseudokoden till programkod.

I stället för att använda pseudokod kan man rita ett strukturdiagram. I figur 7.1 visas hur algoritmen kan beskrivas med hjälp av ett sådant.

Figur 7.1 Ett strukturdiagram för beräkning av summan $1 + 2 + 3 + \ldots + n$



Bilden visar ett strukturdiagram med två vägar, ja eller nej, och som mest 10 steg.

1. Start
2. Läs in n
3. Summan är lika med 0
4. k är lika med 1
5. Är k mindre än n?
6. Vid ja, summan är lika med summa plus k
7. k är lika med k plus 1
8. Tillbaka till k, är k fortfarande mindre än n?

9. Vid nej, skriv ut summa
10. Stopp

I ett strukturdiagram använder man rektanglar för att beskriva de olika stegen. Romboider innehåller ett villkor, och beroende på om villkoret är sant eller falskt väljer man olika vägar.

Uppgift 7.1

Beskriv i pseudokod en algoritm som läser 100 tal från en lista och som beräknar och visar medelvärdet av de lästa talen.

När man ska lösa ett problem behöver man för det mesta kunna utföra ett steg i taget, välja mellan olika alternativ och upprepa ett eller flera steg. Dessa tre ting måste man kunna uttrycka i algoritmen. Hur man gör det i Python vet du redan: Ett steg i taget får man om man helt enkelt skriver ett antal satser efter varandra. För att välja kan man använda `if`-satser, och för att upprepa kan man skriva `while`-satser eller `for`-satser.

7.3 Stegvis förfining

När man ska lösa ett komplicerat problem är det till stor hjälp om man kan dela in problemet i mindre delproblem som kan lösas vart och ett för sig. Delproblemen kan i sin tur delas upp i fler delproblem. Detta är en viktig teknik vid programkonstruktion. Tekniken kallas *stegvis förfining.* Som

exempel visas här en "vardagsalgoritm" som beskriver hur man kan tvätta bilen. En första grov algoritm kan helt enkelt vara:

Tvätta bilen

Denna kan snabbt förfinas till:

1 Om du känner dig lat

1.1 Tvätta i automattvätt

2 annars

2.1 Tvätta för hand

Steg 1.1 kan förfinas till:

1.1.1 Kör till närmaste bensinstation

1.1.2 Köp tvättbiljett

1.1.3 Vänta tills det blir ledigt

1.1.4 Tvätta i automaten

Steg 1.1.4 kan ytterligare förfinas

1.1.4.1 Kör in bilen

1.1.4.2 Kontrollera att allt är stängt

1.1.4.3 Gå ur bilen

1.1.4.4 Sätt in tvättbiljett i automaten

1.1.4.5 Vänta tills det är klart

1.1.4.6 Sätt dig i bilen

1.1.4.7 Kör ut

Stegvis förfining

Dela in i delproblem. Lös delproblemen vart och ett för sig.

Dela in delproblemen i ytterligare delproblem.

Fortsätt på detta sätt tills alla delproblem är enkelt lösbara.

7.4 Ett exempel

Nu ska vi demonstrera hur man kan utveckla ett program med hjälp av pseudokod. Problemet som ska lösas är följande:

> *Skriv ett program som läser in ett godtyckligt antal heltal. För varje tal ska programmet ange om talet är ett primtal eller inte. Med primtal menas alla tal som endast är jämnt delbara med talet 1 och med sig själva.*

Exempel på primtal enligt denna definition är 1, 2, 3, 5, 7 osv. Man kan börja med en grov algoritm:

> *Upprepa följande ett godtyckligt antal gånger*
>
> *1 Läs in ett tal och avbryt om användaren vill sluta*
>
> *2 Undersök om talet är ett primtal*
>
> *3 Skriv ut resultatet*

Den första raden är lätt att översätta till programkod. Det är bara att skriva en `while`-sats. Steg 1 och 3 är också enkla. Du har sett motsvarande kod flera gånger. Man kan därför göra följande översättning:

```
while True:
        # Läs in ett tal och avbryt om användaren vill
sluta
        s = input('Talet? ')
        if s == '':
                break
        talet = int(s)
        # Undersök om talet är ett primtal
        ...
        # Skriv ut resultatet
        if är_primtal:
                print('Primtal')
        else :
                print('Ej primtal')
```

Här har man låtit de olika stegen i algoritmen stå kvar som kommentarer. Användaren markerar att han eller hon vill sluta genom att bara trycka på Enter. Då kommer `s` att innehålla en tom text. I det sista steget används en variabel av typen `bool` med namnet `är_primtal`. Denna variabel ska beräknas i steg 2 som ännu inte är klart.

Steg 2 är inte färdigt. Det kan förfinas ytterligare till:

> 2.1 Sätt variabeln `är_pirimtal` till `True`
>
> 2.2 Försök dividera det inlästa talet med alla tal som är mindre än talet och större än 1. Om någon division går jämnt ut så ändra `är_primtal` till `False`

Steg 2.1 är enkelt:

```
är_primtal = True
```

Steg 2.2 kan förfinas till:

> *Upprepa följande för alla tal k i intervallet 2 till talet-1*
>
> *Om resten blir 0 när man dividerar talet med k så sätt* `är_primtal` *till* `False`

Detta kan man lätt översätta till programtext:

```
for k in range(2, talet):
        if talet % k == 0:
                är_primtal = False
```

Nu återstår bara att sätta samman allt till ett fullständigt program:

[fullständigt program]

```
# Primtal
while True:
        # Läs in ett tal och avbryt om användaren vill
sluta
        s = input('Talet? ')
        if s == ' ' :
                break
        talet = int(s)
        # Undersök om talet är ett primtal
        är_primtal = True
        for k in range(2, talet):
                if talet % k == 0:
                        är_primtal = False
```

```
# Skriv ut resultatet
if är_primtal:
        print('Primtal')
else :
        print('Ej primtal')
```

När man har skrivit ett program ser man ofta att det går att göra en del förändringar så att det blir mer effektivt. Men man ska inte börja tänka på sådana förändringar förrän man fått den första versionen att fungera. Om du studerar programmet, ser du att det finns en enkel förbättring vi kan göra. Om vi i `for`-satsen upptäcker att talet är jämnt dividerbart med ett visst tal `k`, så är det onödigt att fortsätta och testa med fler värden på `k`. Vi kan därför avbryta testerna med en `break`-sats:

```
if talet % k == 0
        är_primtal = False
        break
```

En annan förbättring är att inte testa med onödigt stora värden på `k`. För att förstå detta behöver vi lite matematik. Om talet *t* inte är ett primtal kan vi dela upp det i två faktorer $t = x \times y$. Eftersom $t = \sqrt{t} \times \sqrt{t}$ måste det gälla att $x \leq \sqrt{t}$ eller att $y \leq \sqrt{t}$. Annars skulle produkten $x \times y$ bli större än talet *t.* Om talet inte är ett primtal finns det alltså en faktor som är mindre än eller lika med kvadratroten ur talet. Det räcker därför att vi undersöker om det finns en sådan faktor. Vi ändrar därför i `for`-satsen så att vi bara undersöker `k`-värden upp till och med roten ur `tal`:

```
for k in range(2, math.floor(math.sqrt(talet))+1) :
```

Det finns andra effektivare algoritmer för att avgöra om ett tal är ett primtal. Se t.ex. övning 7.5.

7.5 Sammanfattning

Efter att ha läst detta kapitel bör du:

- veta vad en algoritm är,
- kunna skriva pseudokod för att beskriva enkla algoritmer,
- veta vad ett strukturdiagram är,
- känna till hur man kan använda sig av stegvis förfining.

7.6 Övningar

7.1 Beskriv i pseudokod en algoritm som läser in ett antal tal och som undersöker om talen skrivits in i storleksordning eller inte.

7.2 Ett "perfekt tal" är ett tal där summan av talets faktorer, inklusive 1, men exklusive talet själv, blir lika med talet. Ett par exempel är talen $6 (= 1 + 2 + 3)$ och $28 (= 1 + 2 + 4 + 7 + 14)$. Beskriv i pseudokod en algoritm som avgör om ett tal är perfekt.

7.3 *Euklides algoritm* för att beräkna den största gemensamma delaren till två positiva heltal *m* och *n* (dvs. det största heltal som både *m* och *n* är jämnt delbara med) kan beskrivas med följande pseudokod:

- Dividera m med *n* och beteckna resten vid divisionen med *r.*

- Om $r = 0$, är beräkningen klar och resultatet finns i *n.*

- Sätt annars *m* till *n* och *n* till *r* och gå tillbaka till steg 1.

Använd denna algoritm för att skriva ett program som läser in två heltal och beräknar deras största gemensamma delare.

7.4 Programmet `Primtal` på sidan 131 läste in ett tal och undersökte om talet var ett primtal. Använd detta program som förebild och skriv ett nytt program som läser in ett positivt heltal `n` och som skriver ut alla primtal som är mindre än eller lika med `n`. Tips: Prova alla tal i intervallet 1 till `n` och se om de är primtal.

[överkurs]
7.5 En algoritm för att finna primtal går under namnet *Eratosthenes såll.* Den går till på följande sätt: Anta att man vill finna alla primtal som är mindre än eller lika med `n`. Man skapar då en lista med längden

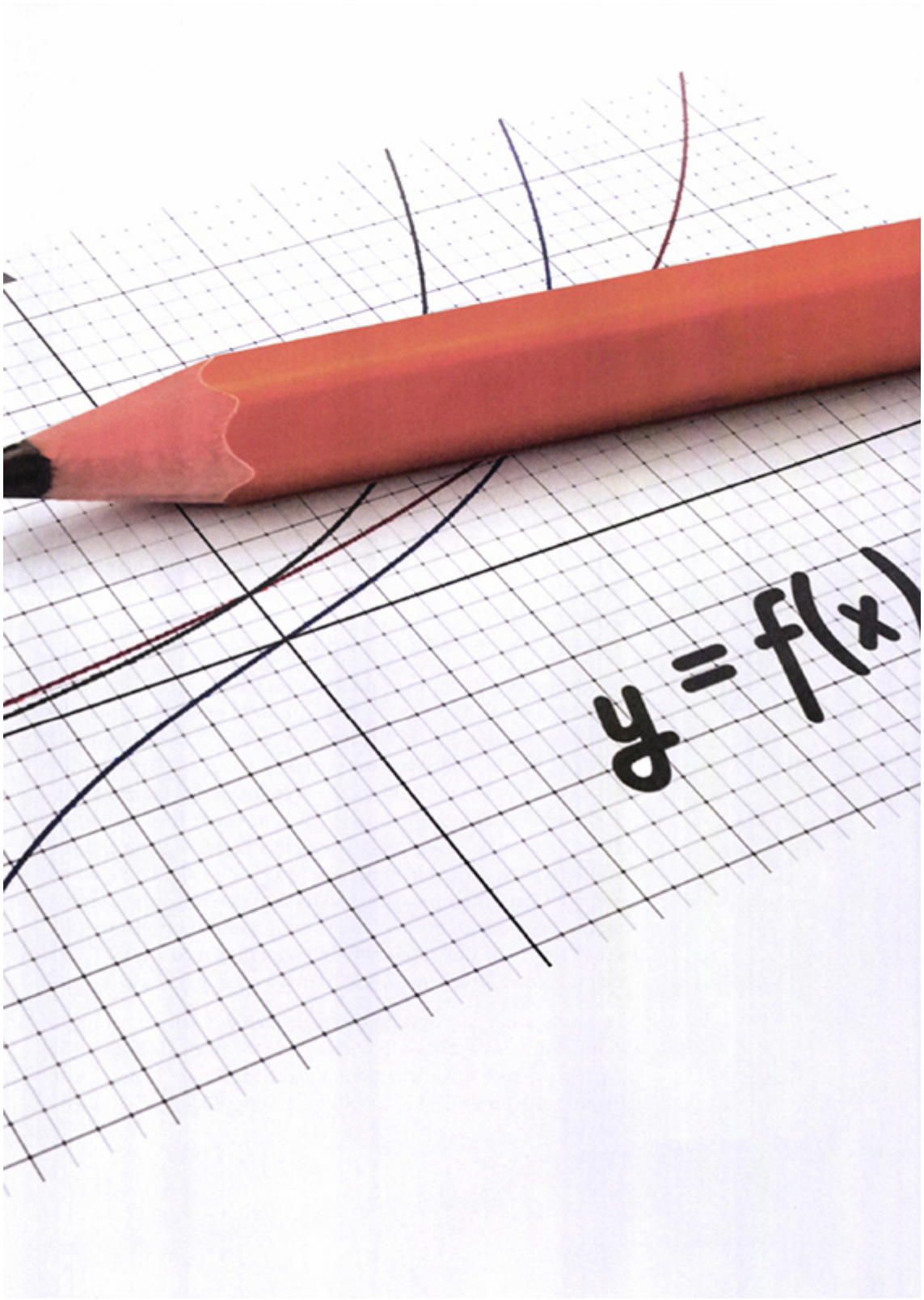
$$\mathrm{n}+1$$

där elementen har typen `bool`. Från början sätts element nummer 0 till `False` och alla övriga element i listan till `True`. Därefter löper man igenom listan med början på index 2. När man stöter på ett element (låt oss säga att det har nummer `i`) som har värdet `True` gör man följande: Löp igenom resten av listan (fr.o.m.

$$\mathrm{i}+1$$

till listans slut) och sätt varje element som har egenskapen att dess index är en jämn multipel av `i` till `False`. När man t.ex. kommer till elementet med index 5, ska elementen med indexen 10, 15, 20 osv. i listan ges värdet `False`.
När denna process är avslutad visar de element i listan som fortfarande har värdet `True` vilka index som är primtal. Konstruera ett program som använder denna algoritm för att beräkna och skriva ut alla primtal som är mindre än eller lika med `n`. Talet `n` ska läsas in.

8 Funktioner


y = f(x)

Nu har du lärt dig grunderna när det gäller programmering. Du vet hur man läser in och skriver ut data, du kan hantera numeriska värden, du kan skriva enkla satser och satser som utför val och repetition och du vet hur man kan hantera text och utnyttja listor. Nästa steg är att skriva lite större, mer komplicerade program. En förutsättning för detta är att du använder funktioner. Alla de program du sett hittills har bara bestått av en enda följd av satser. I detta kapitel ska du få se hur man kan skapa program som består av flera funktioner.

8.1 Definitioner av funktioner

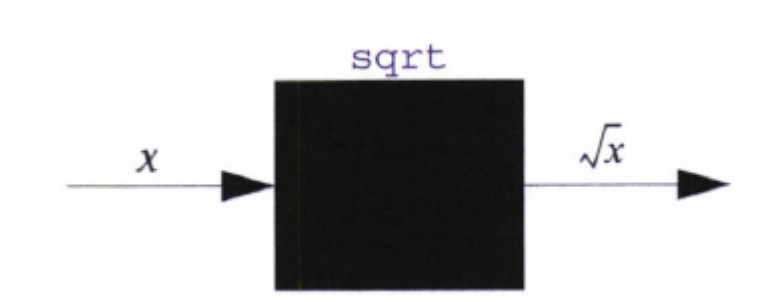
En sats i ett program utför oftast något relativt enkelt. När man beskriver hur en beräkning ska gå till uttrycker man sig ofta på en "högre nivå" än med Pythons grundläggande satser. Man kan t.ex. säga: "beräkna medelvärdet av mätningarna" eller "skriv ut resultaten". Fördelen med att uttrycka sig på denna högre nivå är att man kan bortse från detaljer som för tillfället är oväsentliga. I Python kan man använda funktioner för att sätta samman flera satser till ett steg på en sådan högre nivå.

Varje funktion i ett program ska ha en viss bestämd uppgift. Den ska räkna ut något eller utföra vissa bestämda handlingar. Många funktioner kan göras så generella att de kan vara användbara på flera olika ställen i ett program eller till och med i flera olika program. Man säger då att de kan återanvändas. Ett exempel på detta är funktionerna i modulen `math`. För att t.ex. beräkna roten ur det tal som finns i variabeln `x` och tilldela en annan variabel `y` resultatet kan man skriva:

```
y = math.sqrt(x)
```

Detta är exempel på ett *anrop* av en funktion. Många funktioner har, liksom `sqrt`, till uppgift att beräkna ett värde. En sådan funktion kan uppfattas som en "svart låda" i vilken man stoppar in ett eller flera utgångsvärden. Ut ur lådan kommer då ett resultat som är beroende av dessa utgångsvärden. Funktionen `sqrt` kan illustreras som i figur 8.1.

Figur 8.1 Funktionen `sqrt`.



Figuren visar att variabeln x via funktionen sqrt blir till roten ur x.

Vi ska nu konstruera en funktion `medelv`, som beräknar medelvärdet av två tal. Man ska kunna anropa funktionen genom att t.ex. skriva

```
z = medelv(x, y)
```

Då ska medelvärdet av de tal som finns i variablerna `x` och `y` beräknas och tilldelas variabeln `z`. Se figur 8.2.

Figur 8.2 Funktionen `medelv`*.*

Figuren visar att x och y via funktionen medelv blir till medelvärdet av x och y.

När man anropar en funktion hoppar programmet till funktionen och utför de satser som finns i den. Därefter återvänder programmet till det ställe från vilket anropet skedde. Vi beskriver först hur man konstruerar själva funktionen. Den ser ut på följande sätt:

[fullständigt program]

```
# Medelvärde
def medelv(a, b) :
        return (a + b) / 2
```

Detta kallas en *funktionsdefinition.* Den består av två delar: ett *funktionshuvud* och en *funktionskropp.* Funktionshuvudet står på den första raden och ska inledas med det reserverade ordet `def`. I ett

funktionshuvud anges funktionens namn, som här är `medelv`. Efter namnet anger man vad som ska stoppas in i funktionen genom att ge en lista med namnen på funktionens *parametrar.* Här anges att funktionen har två parametrar som heter `a` och `b`. Inne i funktionen `medelv` betraktas `a` och `b` som variabler. När funktionen `medelv` anropas kommer `a` och `b` att innehålla de värden som stoppas in i funktionen.

I funktionskroppen, som skrivs på raderna efter funktionshuvudet och som måste vara indenterade, beskrivs hur det ser ut inne i den "svarta lådan".

Definition av funktion

```
def namn (p1, p2, etc.):
    en eller flera satser
```

`p1, p2, ...` är funktionens *parametrar.* Det kan finnas godtyckligt många. Alla satser inne i funktionen måste dras in.

En funktionskropp kan innehålla flera satser, men i funktionen `medelv` finns det bara en sats, en `return`-sats:

```
return (a+b)/2
```

En `return`-sats gör två ting: den anger vilket värde som ska ges som resultat från funktionen och den avslutar funktionen, vilket innebär att programmet hoppar tillbaka till det ställe där anropet gjordes. Man kan ha flera `return`-satser i en funktion. Som exempel kommer här en funktion som beräknar det största av två tal.

[fullständigt program]

```
# Största talet
def störst(a, b):
        if a > b:
                return a
        else :
                return b
```

Funktionerna `medelv` och `störst` ger ett numeriskt värde som resultat, men resultatet från en funktion kan vara av vilken typ som helst. Här kommer ett exempel på en funktion som undersöker om ett visst år är ett skottår. (Du kommer väl ihåg hur man testade detta? Se annars på sidan 55.)

[fullständigt program]

```
# Skottår
def är_skottår(år):
        return (år % 4 == 0 and år % 100! = 0) or år %
400 == 0
```

Uttrycket efter `return` är ett sanningsvärde och får typen `bool`.

Det finns funktioner som bara gör något, utan att lämna något särskilt resultat. Här kommer som exempel en funktion som skriver ut dagens

datum. (Hur man får fram datumet beskrevs i avsnitt 5.5 på sidan 96.)

138

[fullständigt program]

```
# Skriv ut dagens datum
import datetime # måste vara med
def skriv_datum():
        dt = datetime.datetime.now() # datum och
tid just nu
        d = dt. date () # datum i dag
        print(str(d)) # skriver ut datum som
'åååå-mm-dd'
```

Det finns inte någon `return`-sats i denna funktion. Den avslutas i stället automatiskt när dess sista sats har utförts. Funktionen ger då det speciella värdet `None` som värde. Man kan också avsluta en funktion av detta slag genom att skriva en `return`-sats utan något uttryck efter ordet `return`. Då avslutas funktionen och resultatet blir värdet `None`.

return-sats

`return` *Uttryck*

Avslutar funktionen och ger värdet *uttryck* som resultat.

Om *uttryck* utelämnas, ges värdet `None` som resultat.

8.2 Anrop av funktioner

En funktionsdefinition är bara en beskrivning av hur en viss beräkning går till. För att verkligen få en beräkning utförd måste man *anropa* funktionen. I följande exempel har funktionen `medelv` lagts in i ett fullständigt program som läser in två tal och som beräknar och skriver ut deras medelvärde. Funktionen `medelv` har placerats i först i filen. Anledningen till detta är att den då är känd av interpretatorn när den kommer till den rad där `medelv` anropas. Här följer programmet:

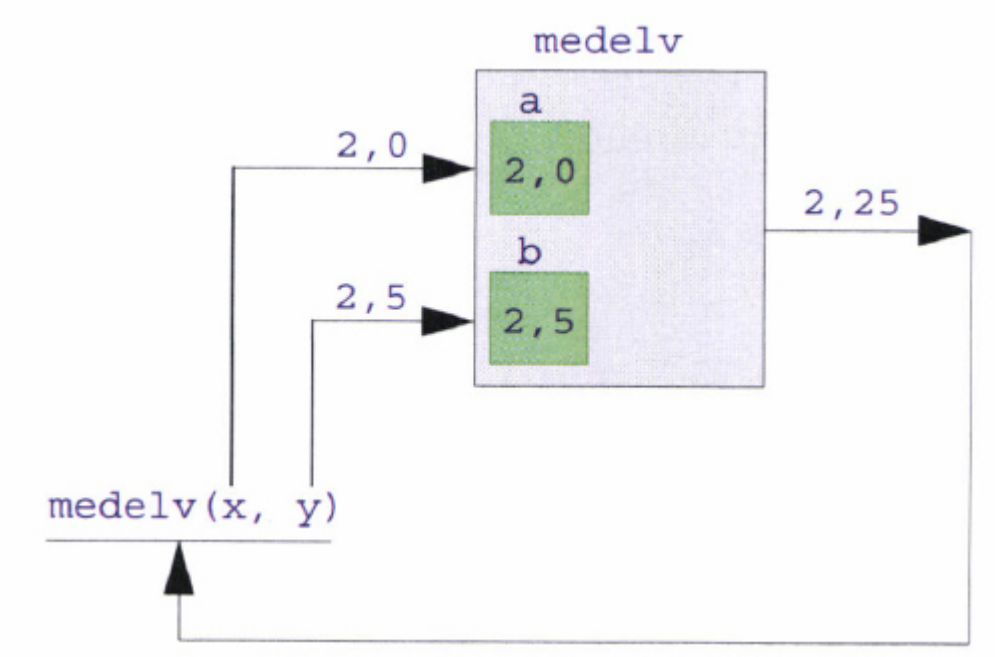
[fullständigt program]

```
# Medelvärde, fullständigt program
def medelv (a, b):
        return (a+b)/2

# Här startar exekveringen
x = float(input('Det första talet? '))
y = float(input('Det andra talet? '))
mv = medelv (x, y)
print(f'Medelvärde: {mv:.2f}')
```


Anropet av funktionen `medelv` har markerats med rött. Först i anropet står namnet på funktionen som ska anropas och sedan följer inom parenteser en lista med *argument* till funktionen. När anropet exekveras beräknas först argumentens värden. (I detta exempel

behövs inga beräkningar eftersom värdena redan finns färdiga i variablerna `x` och `y`.) Därefter stoppas argumentens värden in i funktionen och deras värden *kopieras* till motsvarande parametrar. Anta t.ex. att den som kör programmet har skrivit in de två talen 2,0 och 2,5. Variabeln `x` innehåller då 2,0 och `y` 2,5. Vad som händer demonstreras i figur 8.3.

Figur 8.3 Anrop av funktionen `medelv`*.*



Figuren visar medelv (x, y), där x är 2,0 och y är 2,5, och som via funktionen medelv blir till medelvärdet av a 2,0 och b 2,5. Resultatet blir lika med 2,25.

Vid anropet läggs värdet av det första argumentet i parametern `a` och värdet av det andra argumentet i parametern `b`. Observera att varken `x` eller `y` påverkas av detta eller av vad som senare sker inne i funktionen. Denna teknik att överföra argumentens värden till en

funktion brukar kallas *värdeanrop* (på engelska *call by value)* och innebär alltså att argumentens värden *kopieras* till lokala *kopior* inne i funktionen.

När argumentens värden kopierats till parametrarna `a` och `b` beräknas uttrycket i `return`-satsen. Resultatet av funktionsanropet, alltså av hela uttrycket `medelv (x, y)`, kommer alltså att bli 2,25. När funktionsanropet är klart fortsätter exekveringen och uttryckets värde 2,25 tilldelas variabeln `mv`. Lägg märke till att när funktionsanropet är avslutat, existerar inte de båda parametrarna `a` och `b` längre.

Argumenten till en funktion kan vara uttryck. De behöver inte vara enkla variabler. Man kan t.ex. skriva uttrycket

```
medelv(x*y, x+10)
```

I detta exempel skulle `a` få värdet 5,0 och `b` värdet 12,0 (under förutsättning att `x` och `y` har samma värden som tidigare).

Om en funktion ger ett värde som resultat bör man oftast ta hand om detta resultat vid anropet. I programmet finns t.ex. raden

```
mv = medelv(x, y)
```

Där sparas resultatvärdet i variabeln `mv`. Ett annat sätt att ta vara på resultatvärdet är att skriva ut det. Man kunde t.ex. skrivit

```
print(f'Medelvärde: {medelv(x, y):.2f}')
```

Om man kör i *interactive mode* och bara skriver anropet, kommer resultatvärdet att skrivas ut direkt:

```
>>> medelv(2.0, 2.5)
2.25
```

Att Python använder *värdeanrop (call by value)* när argumenten överförs till parametrarna innebär att argumenten inte kan ändras av funktionsanropet. Anta t.ex. att man vill skriva en funktion som byter värden på två variabler. Man försöker med följande:

```
def byt_fel(x, y): # Felaktig!!
        temp = x
        x = y
        y = temp
```

Man testar funktionen med följande programrader:

```
a = 5
b = 1
byt_fel(a, b)
print(f'a = {a} b = {b}'
```

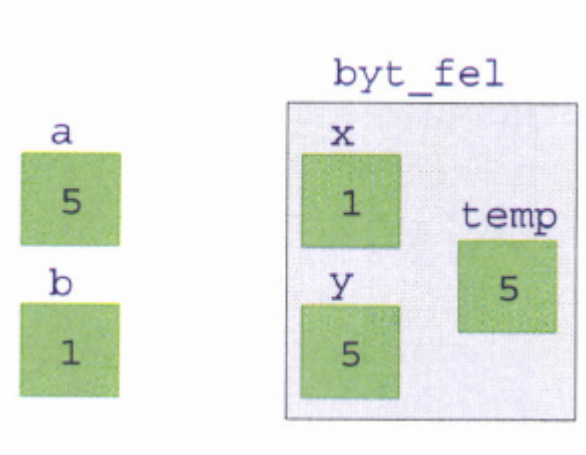
Avsikten är att `a` och `b` ska byta värden så att `a` får värdet 1 och `b` värdet 5. Men när man kör programraderna får man utskriften:

$$a = 5$$

$$b = 1$$

Variablerna har alltså *inte* bytt värde. Att det blir så beror på att parametrarna `x` och `y` är *kopior* av argumenten `a` och `b`. Vad som händer är att värdena ändras i kopiorna, men originalen ändras inte. I figur 8.3 visas hur det ser ut precis innan funktionen `byt_fel` avslutas.

Figur 8.4 Anrop av funktionen `byt_fel`.



Figuren visar att a är 5 och b är 1. Vid anrop av funktionen byt_fel är x 1 och y 5, vilket blir till temp 5.

En funktion kan ha ett godtyckligt antal parametrar. Den kan t.o.m. helt sakna parametrar. Ett exempel på en sådan funktion är funktionen `random` i modulen `random`, se sidan 38. Denna funktion ger som resultat ett slumpmässigt tal i intervallet `0` till `1`.

```
import random
x = random.random() # 0 ≤ x < 1
```

Lägg märke till att man måste ha tomma parenteser med i anropet fast funktionen saknar parametrar.

Om en funktion ger ett värde av typen `bool` som resultat kan man använda anropet som villkor t.ex. i en `if`-sats. Vi kan t.ex. anropa funktionen `är_skottår` så här:

```
n = int(input('Vilket år? '))
if är_skottår(n):
        print('Skottår')
else :
        print('Inte skottår')
```

När man anropar en funktion som inte ger något resultat skriver man helt enkelt funktionens namn och argumenten. Det finns ju inget värde att ta hand om. Funktionen `skriv_datum` anropas så här:

```
skriv_datum()
```

Uppgift 8.1

I uppgift 2.7 på sidan 38 skulle du skriva ett program som beräknade en cirkels omkrets och area. Gör en ny version av detta program, i vilket beräkningen av omkretsen och arean görs i två separata funktioner.

Du kanske undrar om inte en funktion kan ge flera resultatvärden. Kan man t.ex. inte konstruera en

---

funktion som returnerar både omkretsen och arean av en cirkel? Svaret på frågan är egentligen nej eftersom en funktion kan returnera högst ett värde. Men vill man ge flera resultat kan man som resultat ge tupel som innehåller de värden man vill returnera. Här kommer som exempel en funktion som räknar ut både summan och produkten av två tal:

[fullständigt program]

```
# Summa och produkt, version 1
def sum_prod(a, b):
        return (a+b, a*b)
```

Som resultat ger `sum_prod` en tupel där det första elementen innehåller summan och det andra produkten. Men man kan utelämna parenteserna runt en tupel. Därför kan man låta funktionen se ut på följande sätt, och då ser det ut som om den returnerar två värden:

[fullständigt program]

```
# Summa och produkt, version 2
def sum_prod(a, b):
        return a+b, a*b
```

Funktionen kan anropas så här

```
t = sum_prod(2.0, 2.5)
print(f'Summa: {t[0]:.2f}')
print(f'Produkt: {t[l]:.2f}')
```

Här blir variabeln `t` en tupel. Vi har använt indexering för att få ut de två värdena. Ett elegantare sätt är att använda multipel tilldelning:

```
s, p = sum_prod(2.0, 2.5)
print(f'Summa: {s:.2f}')
print(f'Produkt: {p:.2f}')
```

Uppgift 8.2

Gör ytterligare en version av det program som beräknar en cirkels omkrets och area. Låt beräkningarna av omkretsen och arean göras i en enda funktion som ger två resultat med hjälp av en tupel.

8.3 Lokala variabler

De funktioner som vi diskuterat hittills i detta kapitel har varit ganska enkla. Men funktioner kan vara mer avancerade. De kan innehålla flera

---

satser och använda sig av variabler. Variabler som man deklarerar inne i en funktion kallas *lokala variabler.*

Som exempel kommer här en funktion som får ett heltal som parameter och som beräknar summan av siffrorna i talet.

[fullständigt program]

```
# Siffersumman
def siff_sum(n):
        sum = 0
        while n > 0:
                rest = n % 10
                sum = sum + rest # addera sista
siffran
                n = n // 10 # ta bort den sista
siffran
        return sum
```

Om vi t.ex. gör anropet

```
z = siff_sum(52073);
```

får `z` värdet `17`.

I funktionen `siff_sum` ska vi beräkna summan av siffrorna i det tal som finns i parametern `n`. Det finns två lokala variabler: `sum` och `rest`. Variabeln `sum` använder vi för att beräkna siffersumman. Den sätts till noll från början, `while`-satsen snurrar ett varv för varje siffra i talet `n`. Uttrycket `n % 10` ger den rest som man får när man dividerar `n` med `10`. Det blir helt enkelt värdet av den sista siffran. Om talet `n` har värdet `52073` som i exemplet, beräknas uttrycket `52073 % 10` som blir `3`. Vi lägger den rest vi får i den lokala variabeln `rest`. På den andra raden i `while`-satsen adderar vi denna rest till summan. På den sista raden i `while`-satsen utför vi heltalsdivisionen `n // 10`. Då divideras `n` med 10 och decimalerna kapas av. Då blir resultatet ett värde som består av alla siffrorna, utom den sista. Om `n` har värdet `52073`, beräknas `52073//10` som blir `5207`. Efter första varvet i `while`-satsen har alltså variabeln `sum` ändrats till `3` och variabeln `n` till `5207`. På andra varvet adderas resten `7` till variabeln `sum` och `n` ändras så att den blir `520`. `while`-satsen snurrar sedan tre varv till. På det sista varvet har `n` från början värdet `5`. Då blir uttrycket `n // 10` lika med `0`. Variabeln `n` tilldelas då värdet `0`, vilket betyder att villkoret i `while`-satsen blir falskt, och inga fler varv görs.

Uppgift 8.3

Skriv en funktion som beräknar *antalet* siffror i ett heltal. För enkelhets skull får du förutsätta att talet är större än noll. *Tips:* Gör på ungefär samma sätt som i programmet `siff_sum`, men räkna antalet varv i stället för att beräkna siffersumman.

Nu ska vi titta närmare på var de två variablerna `sum` och `rest` har skapats. Om man tilldelar ett värde till en variabel inne i en funktion skapas en ny variabel, och denna variabeln blir bara synlig *inne* i denna funktion. Detta gäller för variablerna `sum` och `rest`. Skulle vi senare i programmet, utanför funktionen `siff_sum`, försöka avläsa värdet av någon av dessa variabler skulle vi få meddelandet att variabeln inte är definierad.

Eftersom man aldrig kan komma åt en lokal variabel utifrån, t.ex. från en annan funktion, är det tillåtet att skapa flera variabler med samma namn om variablerna inte ligger i *samma* funktion. Det blir i så fall fråga om *olika* variabler som råkar ha samma namn.

Lokala variabler har en begränsad livslängd. De skapas i det ögonblick funktionen anropas och existerar sedan bara under den tid som funktionsanropet pågår. När funktionen kört klart upphör de lokala variablerna att finnas till. Parametrarna till en funktion är också ett slags lokala variabler. De existerar alltså också bara medan funktionen körs.

Lokala variabler och parametrar

Deklareras inne i en funktion.

Är kända bara inne i den funktion i vilken de skapas. Existerar bara medan funktionsanropet pågår.

Man får också skapa variabler utanför funktioner. Sådana variabler kalllas *globala variabler* och de är synliga i hela filen.

Globala variabler

Deklareras utanför funktioner. Är kända i hela filen (modulen).

Observera att här finns en sak som är lite underlig i Python. Om man inne i en funktion tilldelar ett värde till en variabel med samma namn som en global variabel så skapas en *ny* variabel som blir lokal, men om man bara avläser en variabel som är global, skapas ingen ny variabel. Detta kan demonstreras i följande program:

[fullständigt program]

```
# Demo. Lokala och globala variabler. Version 1
x, y, z = 0, 0, 0 # skapar tre globala variabler

def f(x):
        x = 1 # ändrar parametern x
        y = 2 # skapar en lokal variabel y
        print(x, y, z)

f(x) # anropar funktionen f
print(x, y, z)
```

När man kör programmet skapas först de tre globala variablerna `x`, `y` och `z`. De tilldelas alla värdet 0. Därefter följer definitionen av en funktion med namnet `f`. Denna funktion anropas på den nästa sista raden i programmet. Som argument ges den globlala variabeln `x`.

Den innehåller värdet 0. Det betyder att värdet 0 *kopieras* till funktionens parameter som också råkar heta `x`. Observera att den globala variabeln `x` och parametern `x` inte är samma variabel; de har bara samma namn. Inne i funktionen `f` tilldelas parametern `x` värdet 1, men den globala variabeln `x` påverkas inte av detta. På nästa rad i funktionen tilldelar man värdet 2 till en variabel med namnet `y`. Då skapas en ny lokal variabel. Den globala variabeln `y` påverkas inte. På den sista raden i funktionen skrivs värdena av variablerna `x, y` och `z` ut. Då får man utskriften:

```
1 2 0
```

Det som skrivs ut är parametern `x`, den lokala variabeln `y` och den globala variabeln `z`.

På den sista raden i programmet, efter anropet av `f`, skrivs variablerna `x, y` och `z` ut igen. Men denna gång skriver man ut värdena av de tre globala variablerna. (Parametern `x` och den lokala variabeln `y` är ju inte kända utanför funktionen.) Utskriften blir:

```
0 0 0
```

Dessa regler finns i Python för att man inte inne i en funktion av misstag ska kunna ändra en global variabel. Men om man verkligen vill

göra detta är det möjligt. Man måste då uttryckligen ange att man menar en global variabel. Här kommer en ny version av demoprogrammet där vi gjort detta. Vi har lagt till den röda raden.

[fullständigt program]

```
# Demo. Lokala och globala variabler. Version 2
x, y, z = 0, 0, 0 # skapar tre globala variabler

def f(x):
        x = 1 # ändrar parametern x
        global y # anger att y är den globala
variabeln
        y = 2 # ändrar den globala variabeln y
        print(x, y, z)

f(x) # anropar funktionen f
print(x, y, z)
```

Här skapas inte någon ny lokal variabel `y` inne i funktionen. I stället kommer den globala variabeln `y` att ändras. Om man kör detta program får man utskrifterna:

```
1 2 0
0 2 0
```

Uppgift 8.4

Produkten $1 \times 2 \times 3 \times \ldots \times n$ kallas inom matematiken för *fakulteten* för *n.* Den skrivs som *n!* Fakulteten för talet 0, dvs. 0! är definierad som 1. (För negativa tal är fakulteten inte definierad.) Skriv en funktion `nfak` som får ett heltal *n* som

parameter och som beräknar och returnerar värdet av *n!.* Du får förutsätta att *n* inte är negativ. Lägg också sist i filen in några programrader som anropar funktionen och som skriver ut resultatet från den.

Uppgift 8.5

I uppgift 7.1 på sidan 129 skulle du skriva pseudokod för en algoritm som avgjorde om ett tal var "perfekt". Använd denna algoritm för att skriva en funktion med namnet `är_perfekt`. Funktionen ska få talet som parameter och ge ett värde av typen `bool` som resultat. Lägg in funktionen i ett fullständigt program som läser in ett tal och testar om det är perfekt.

8.4 Referenser som parametrar

De parametrar du hittills har sett har varit av enkla. I detta avsnitt ska du få se att funktioner också kan ha texter, tupler och listor som parametrar. Som första exempel visas en funktion som beräknar summan av alla element som är större än eller lika med `x` i en lista eller tupel:

[fullständigt program]

```
def ref_demo(a, x):
        sum = 0
        for e in a:
                if e > = x:
                        sum += e
return sum
```

Om vi t.ex. gör följande

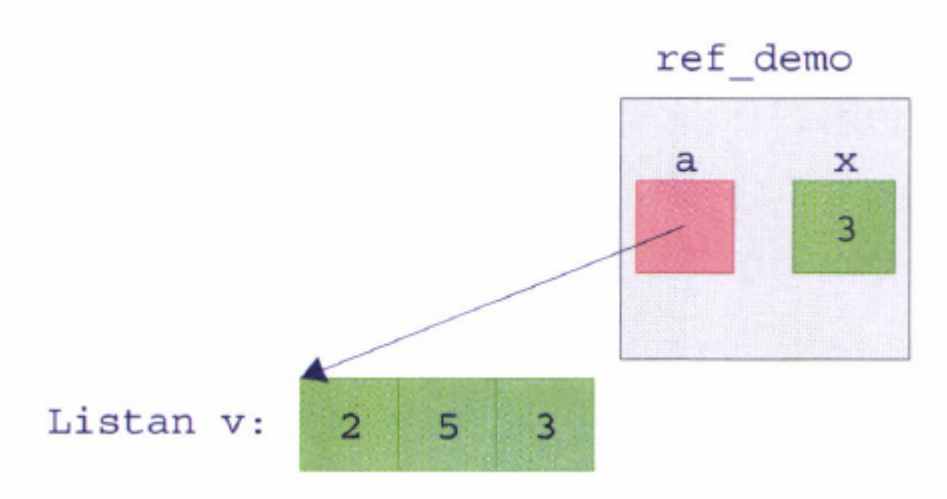
```
v = [2, 5, 3]
print(ref_demo(v, 3))
```

så får vi utskriften

```
8
```

I avsnitt 6.5 konstaterade vi att en listvariabel i själva verket är en referens till själva listan. Det betyder att variabeln `v` är en referens. När man anropar funktionen `ref_demo` kommer därför *referensen* att kopieras till parametern `a`. Själva listan kopieras inte. Detta illustreras figur 8.5.

Figur 8.5 Anrop av funktionen `ref_demo`.



Figuren visar att listan v är 2, 5 och 3. Vid anrop av funktionen ref_demo refererar a till v, och x är 3.

Här visas hur det ser ut när man gjort anropet `ref_demo (v, 3)`. I den skuggade delen finns funktionen `ref_demo` med sina två parametrar `a` och `x`. Parametern `x` innehåller värdet `3` som är en kopia av argumentet `3`. Det är inget konstigt med det. Men parametern `a`, innehåller inte någon lista. I stället innehåller den en *referens* till listan `v`.

Detta händer alltid när man anropar en funktion som har en sekvens, t.ex. en lista som parameter. Det som skickas till funktionen är en *referens* till sekvensen, inte själva sekvensen.

Uppgift 8.6

Skriv en ny version av funktionen `ref_demo`, i vilken du använder dig av *list comprehension* (se sidan 104) och standardfunktionen `sum`.

Uppgift 8.7

Skriv en funktion som beräknar medelvärdet av elementen i en sekvens med numeriska element. Anropa den sedan två gånger i slutet av programmet, en gång med en lista och en gång med en tupel som parameter.

På sidan 140 diskuterade vi en funktion `byt_fel` som försökte byta värden på två enkla paramerar. Den funktionen fungerade inte eftersom Python använder värdeanrop. Studera nu följande funktion som är mycket lik funktionen `byt_fel`:

[fullständigt program]

```
def byt_elem(v, i, j):
        temp = v[i]
        v[i] = v[j]
        v[j] = temp
```

Meningen är att man ska kunna anropa denna för att byta innehåll mellan två element i en lista. Den första parametern är listan och de två andra är indexen för de element som ska byta värde. Man kan t.ex. skriva programraderna:

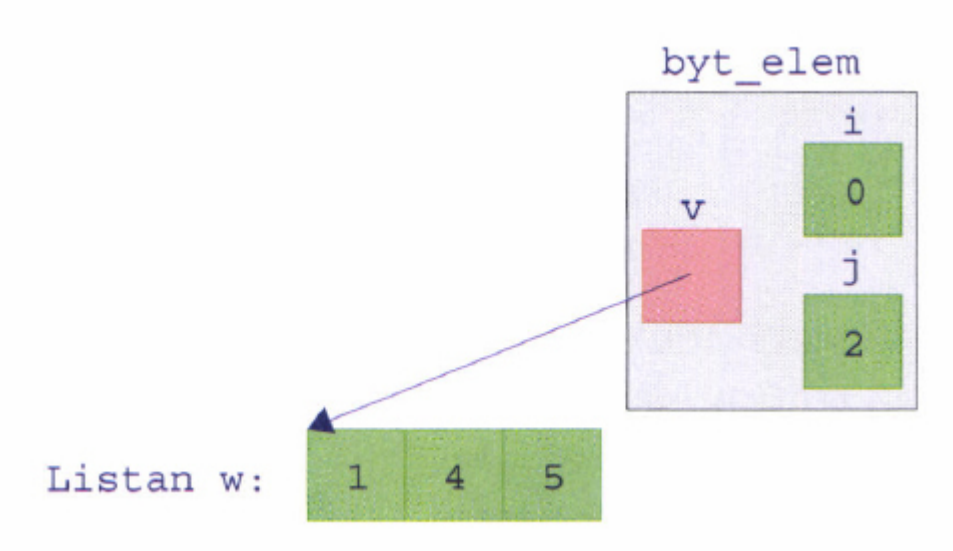
```
w = [5, 4, 1]
byt_elem(w, 0, 2) # byt elementen 0 och 2
print(w)
```

Om man resonerar på samma sätt som när det gällde funktionen `byt_fel`, borde inget ändras i listan `w`. Men utskriften blir:

```
[1, 4, 5]
```

Här har alltså elementen 0 och 2 bytt värde med varandra! För att få svar på varför detta sker kan vi studera figur 8.6. Figuren visar situationen när funktionen `byt_elem` har exekverat färdigt och precis ska avslutas. När man anropar funktionen `byt_elem` kopieras argumenten till lokala kopior inne i funktionen.

Figur 8.6 Anrop av funktionen `byt_elem`.



Figuren visar att listan w är 1, 4, och 5. Vid anrop av funktionen byt_elem refererar v till w, i är 0, och j är 2.

Men eftersom argumentet `w` är en lista är det en referens till listan som kopieras till parametern `v`. Själva listan kopieras inte. Detta betyder, som framgår av figuren, att parametern `v` pekar på listan `w`. Om man gör ändringar i listan via parametern `v`, kommer man därför att ändra i själva listan.

Det skulle inte gå att anropa funktionen `byt_elem` med en tupel som parameter, eftersom man inte får ändra i en sådan.

Listor som parametrar till funktioner

När man anropar en funktion och ger en lista som argument, kopieras en *referens* till listan till motsvarande parameter i funktionen. Parametern pekar alltså på *samma* lista som argumentet.

Om man gör en ändring i listan inne i funktionen, kommer därför denna ändring att påverka listan.

Uppgift 8.8

Skriv en funktion med namnet `fyll` som tilldelar ett (och samma) värde till alla element inom ett visst intervall i en lista. Funktionen ska ha fyra parametrar: listan, det undre indexet i intervallet, det övre indexet och det värde som ska tilldelas. För att t.ex. lägga värdet 2.3 i elementen nummer 5 till 9 i listan `a` ska man kunna göra anropet:

```
fyll(a, 5, 9, 2.3)
```

Vi ska också i detta avsnitt ge ett exempel på en funktion som har en text som parameter. Denna funktion löser ett problem som vi stötte på i avsnitt 5.3.3 när vi skulle jämföra texter alfabetiskt. Som vi såg i detta avsnitt uppfattades stora och små bokstäver som olika när man jämförde.

Dessutom kom de svenska bokstäverna å och ä i fel ordning. Vi såg t.ex. följande:

```
'abc' == 'Abc' # ger värdet False
```

```
'å' < 'ä' # ger värdet False
```

Vi ska nu skriva en funktion med namnet `alfa`. Meningen är att när man ska göra en jämförelse, ska man inte jämföra texterna direkt, utan anropa funktionen `alfa` för båda texterna. Man jämför sedan de resultat man får. Man ska alltså skriva

```
alfa('abc') == alfa('Abc') # ger värdet True
alfa('å') < alfa('ä') # ger värdet True
```

Så här ser funktionen `alfa` ut:

[fullständigt program]

```
def alfa(s):
        s = s.lower()
        v = list(s)
        for i in range(0, len(v)):
                if v[i] == 'ä':
                        v[i] = 'å'
                elif v[i] == 'å':
                        v[i] = 'ä'
        return v
```

Funktionen har en parameter `s` av typen `str`. Eftersom `str` är en sekvenstyp är parametern en referens till själva texten. På andra raden anropas funktionen `lower` som ersätter alla eventuella stora bokstäver med små. Är inte detta farligt? Kommer inte då texten man anropade med att ändras? Nej, det gör den inte. Vad som händer är att funktionen `lower` skapar en *ny* text och att parametern `s` ändras så att den pekar på den nya texten. Den referens man gav som argument när man anropade funktionen kommer inte att ändras.

Parametern `s` var ju en kopia av argumentet. Slutsatsen är att det aldrig är "farligt" att anropa en funktion med en text som argument. Texten kan aldrig ändras.

Vad som sedan sker i funktionen är följande: Vi skapar en lista `v` som innehåller alla tecknen i texten `s`. Anledningen till att vi använder en lista är förstås att det går att ändra i den. Det går inte i en text. Sedan löper vi igenom hela listan och ändrar alla ä till å och vice versa. Som resultat ger vi listan.

8.5 Mer om parametrar

8.5.1 Anrop med namn

För det mesta anger man argumenten i samma ordning som parametrarna när man anropar en funktion, men man kan alternativt ange parametrarnas namn i anropet. Vi demonstrerar detta med hjälp av följande funktion som beräknar ränta på ränta. Om man sätter in beloppet *b* kr på banken, räntan är *r* % och man låter beloppet stå inne *n* år, växer kapitalet, med ränta på ränta, till $b \times (1 + 0,01r)^n$ kr. En funktion som utför denna beräkning kan se ut så här:

[fullständigt program]

```
# Ränta på ränta, version 1
```

```
def saldo(ränta, kapital, år):
        return kapital*(1+0.01*ränta)**år
```

Funktionen har tre parametrar. Vi kan förstås anropa funktionen som vanligt genom att räkna upp argumenten:

```
s = saldo(1.2, 100, 10)
```

Men man kan i stället ange parametrarnas namn. Vi kan t.ex. skriva:

```
s = saldo(kapital = 100, ränta = 1.2, år = 10)
```

Man får då ge argumenten i vilken ordning man vill. Det går också att kombinera de båda sätten och ge några parametrar i ordning och några med namn, t.ex.

```
s = saldo(1.2, år = 10, kapital = 100)
```

Men observera att de argument som ska ges i ordning måste komma först och de som anges med namn sist.

8.5.2 Defaultvärden

När man definierar en funktion kan man låta en eller flera parametrar ha s.k. defaultvärden. Det kan t.ex. se ut så här:

[fullständigt program]

```
# Ränta på ränta, version 2
def saldo(ränta, kapital = 100, år = 1):
        return kapital*(1+0.01*ränta)**år
```

Här har parametern `kapital` defaultvärdet `100` och `år` defaultvärdet `1`. Om en parameter har ett defaultvärde behöver man inte ge motsvarande argument. I så fall får parametern defaultvärdet. Om vi t.ex. gör anropet:

```
s = saldo(0.9)
```

så är det samma sak som om vi hade skrivit

```
s = saldo(0.9, 100, 1)
```

Om en parameter har ett defaultvärde kan man ändå ge ett argument. I så fall är det detta som gäller. Vi kan t.ex. skriva

```
s = saldo(0.9, 1000)
```

Då får parametern `kapital` värdet `1000` och `år` värdet `1`.

Om man vill ge ett värde till en parameter som ligger efter en parameter som fått defaultvärde, måste man ange parameterns namn, t.ex.

```
s = saldo(0.9, år = 5)
```

Här får `kapital` värdet `100`.

När man definierar en funktion där några parametrar har defaultvärden, måste alltid de parametrar som saknar defaultvärden komma först. Det hade t.ex. inte gått att lägga parametern `ränta` efter parametern `kapital` eller parametern `år`.

8.5.3 Variabelt antal parametrar

Det går att definiera funktioner som har ett variabelt antal parametrar. Standardfunktionen `print` som vi använt så många gånger är en sådan funktion. Man kan ju räkna upp ett eller flera värden när man anropar denna funktion, t.ex.

```
print()
print(x)
print(x, y)
```

Om vi tittar på hur funktionen `print` är definierad ser vi att den första raden ser ut så här:

```
print(*objects, sep=' ', end='\n',
```

```
        file=sys.stdout, flush=False)
```


Det är den första parametern som är intressant just nu. Tecknet * som står framför parameternamnet anger att man här kan ge ett godtyckligt antal argument. Dessa argument kommer att överföras till funktionen som en tupel.

Hur det kan se ut inne i en funktion med ett variabelt antal parametrar kan vi illustrera genom att skriva en ny version av funktionen `medelv`. Den version av funktionen som definierades på sidan 136 hade alltid två parametrar, men nu ska vi göra en ny version där man kan ha ett godtyckligt antal parametrar:

[fullständigt program]

```
# Medelvärde, version 2
def medelv(*param):
        sum = 0
        for tal in param:
                sum += tal
        return sum / len(param)
```

Om vi t.ex. gör anropet

```
m = medelv(1.5, 2.3, 3.5)
```

kommer `param` att bli en tupel med innehållet `(1.5, 2.3, 3.5)`. Inne i funktionen kan man sedan löpa igenom denna på vanligt sätt.

Om en funktion har några parametrar som kommer efter den som markerats med tecknet `*`, måste dessa parametrar anropas med namn.

8.6 Referenser till funktioner

[överkurs]

Vi har sett att om man tilldelar en sekvens (lista, tupel eller text) till en variabel kommer denna variabel att innehålla en *referens* till sekvensen. Själva sekvensen finns inte i variabeln. Det är faktiskt samma sak med funktioner. En funktions namn är i själva verket en referens till själva funktionskoden. För att illustrera detta kan vi definiera följande två funktioner:

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

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


Om vi nu gör tilldelningen

```
f = kvad # Obs! inga parenteser här
```

kommer en variabel med namnet `f` att skapas. Denna tilldelas det värde variabeln `kvad` innehåller. Men detta värde är en referens till den första av funktionerna ovan. Det betyder att variabeln `f` kommer att referera till samma funktion som variabeln `kvad`. Detta betyder att vi kan anropa denna funktion med hjälp av variabeln `f`:

```
a = f(3) # a får värdet 9
```

Det blir exakt samma sak som om vi hade skrivit

```
a = kvad(3) # a får värdet 9
```

Men en variabel kan ändras. Om vi nu skriver:

```
f = kub
b = f(3) # a får värdet 27
```

kommer variabeln f att peka på den andra av funktionerna.

Referenser till funktioner kan användas som parametrar. Detta är användbart i vissa sammanhang. En standardfunktion som använder funktionsreferenser är funktionen `map`. Den ska som första parameter ha namnet på en funktion, alltså en referens till en funktion, och som andra parameter en sekvens. Inne i funktionen `map` anropas den funktion parametern refererar till för alla element i den andra parametern. Vi kan t.ex. anropa `map` med funktionen `kvad` som

parameter. Då anropas funktionen `kvad` för varje element i den andra parametern.

```
t = (1, 2, 3)
m = tuple(map(kvad, t)) # m får värdet (1, 4, 9)
```

En annan liknande standardfunktion är `filter`. Den väljer ut vissa element ur en sekvens. Vilka element som ska väljas bestäms av den första parametern som är en referens till en funktion som ger en `bool` som resultat. Om denna funktion anropas med ett element som parameter och resultatet blir `True` kommer elementet att väljas, annars inte.

Här kommer ett exempel. Anta att vi vill plocka ut att värden som är större än noll ur en sekvens med numeriska tal. Vi börjar med att definiera en funktion som testar om ett tal är positivt:

```
def positiv(x):
        return x > 0
```


Sedan utför vi satserna.

```
v = [1, -2, 3, 0]
w = list(filter(positiv, v)) # w får värdet [1, 3]
```

Endast de värden som är större än noll kommer nu att väljas ut.

När man definierar en funktion med hjälp av nyckelordet `def` sker egentligen två ting: själva funktionen byggs upp och man får en referens till funktionen. Själva funktionen är egentligen namnlös; man når den via referensen. Om man vill konstruera en liten enkel funktion som bara innehåller en `return`-sats, kan man skapa en anonym funktion med hjälp av ett s.k. *lambda-uttryck.* Låt oss som exempel ta funktionen `kvad` som vi definierade tidigare. Den såg ut så här:

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

Vi kan få samma resultat genom att skriva:

```
kvad = lambda x :x*x
```

Lambda-uttrycket har markerats med rött. Efter ordet `lambda` räknar man upp parametrarna och efter kolontecknet skriver man det uttryck som funktionen ska returnera. Här har vi tilldelat resultatet av lambdauttrycket till en variabel med namnet `kvad`. Denna refererar alltså till den anonyma funktionen.

Ofta använder man lambda-uttryck när man inte är intresserad av att använda funktionen på mer än ett ställe, t.ex. då man ska ge den som argument i ett funktionsanrop. Vi kan t.ex. skriva

```
m = tuple(map(lambda x :x*x, t))
```

Då behöves inte variabeln `kvad`.

Uppgift 8.9

Skriv ett program som läser in en lista med heltal och som skapar och skriver ut en ny lista som bara innehåller de tal som är udda. Använd funktionen `filter`. Prova också att anropa den med ett lambda-uttryck.

Två ytterligare standardfunktioner som kan ha en referens till en funktion som parameter är `sorted` och `sort`. Båda utför sortering. Funktionen `sort` ska ha en lista som parameter och den ändrar i listan, sorterar

---

den "på plats". Funktionen `sorted` kan ha vilken typ av sekvens som helst som parameter och den skapar och returnerar en ny lista som är sorterad. Båda funktionerna behöver jämföra elementen två och två för att kunna placera dem i rätt ordning. I normala fall används då elementens värden direkt, t.ex.

$t = (7, 0, 3, 2)$
```
u = sorted(t) # u får värdet [0, 2, 3, 7]
```
$v = [-1, 8, 4, 0]$
```
v.sort() # v får värdet [-1, 0, 4, 8]
```

Men när man har mer komplicerade element som ska sorteras har man möjlighet att själv bestämma hur elementen ska jämföras. Det är möjligt eftersom funktionerna `sort` och `sorted` båda har en extra parameter med namnet `key`. Denna ska vara en referens till en funktion som anger vad som ska jämföras. Anta t.ex. att vi har listan

```
a = ['kärna', 'kål', 'Körsbär']
```

och försöker sortera den på vanligt sätt:

```
a.sort() # a blir: ['Körsbär', 'kärna', 'kål']
```

Då hamnar elementen i helt fel ordning. Om vi i stället anger att funktionen `alfa` som vi skrev på sidan 150 ska användas när man jämför, blir den alfabetiska ordningen rätt:

```
a.sort(key=alfa) # a blir: ['kål', 'kärna',
'Körsbär']
```

Funktionen `sorted` fungerar på samma sätt:

```
b = ('Nörd', 'näsa', 'nål')
c = sorted (b, key=alfa) # c blir: ['nål', 'näsa',
'Nörd']
```

Ett annat användningsområde för referenser till funktioner är numeriska beräkningar. En del numeriska funktioner är så utformade att de som en av sina parametrar har en referens till en matematisk funktion. En funktion som utför matematisk integrering har t.ex. en parameter som anger vilken funktion som ska integreras.

Som exempel ska vi konstruera en funktion som beräknar ett nollställe till matematiska funktioner. Funktionens första rad ser ut så här:

```
def nollställe(f, a, b, epsilon=1e-10):
```

Funktionen har fyra parametrar. Den första parametern, `f`, är en referens till den matematiska funktion man vill söka ett nollställe för. De

---

två parametrarna `a` och `b` anger inom vilket intervall nollstället ska sökas. Vi söker alltså ett värde x i intervallet (a, b) sådant att $f(x) = 0$. Vi antar att den funktion `f` pekar på har exakt ett nollställe inom det givna intervallet. Parametern `epsilon` anger vilket som är det största fel som får finnas i resultatet. Idén vi ska använda är att "ringa in" nollstället genom att flytta ändpunkterna `a` och `b` allt närmare varandra. Funktionen `nollställe` ger x-värdet för nollstället som resultat. Om det inte finns något nollställe i det givna intervallet, ges resultatet `None`. Hela definitionen av funktionen följer här:

[fullständigt program]

```
# Sök nollställe
def nollställe (f, a, b, epsilon=1e-10) :
        if f(a) > 0 and f(b) < 0:
                temp = a # byt a och b
                a, b = b, temp
if not (f(a) < 0 and f(b) > 0):
        return None # inget nollställe i
intervallet
while abs(a-b) > epsilon:
        xm = (a+b)/2
```

```
        fm = f (xm)
        if fm < 0:
                a = xm
        elif fm > 0:
                b = xm
        else :
                return xm # vi råkade finna
nollstället
return (a+b)/2 # returnera mittpunkten
```

I första delen av funktionen ser vi till att villkoret $f(a) < 0 < f(b)$ blir sant. Om det visar sig att $f(b) < 0 < f(a)$ byter vi helt enkelt `a` mot `b`. Om f(a) och f(b) båda är större än noll eller mindre än noll kan nollställe saknas i intervallet. Vi ger i så fall värdet `None` som resultat.

Annars beräknar vi mittpunkten `xm` i intervallet (a, b) samt funktionens värde i denna punkt. Detta görs i anropet `f(xm)`. Där anropas den funktion `f` pekar på. Om funktionsvärdet är mindre än noll måste nollstället ligga till höger om `xm`. Vi ändrar i så fall intervallet så att `a` sätts till `xm`. Skulle funktionsvärdet vara större än noll måste på motsvarande sätt nollstället ligga till vänster om `xm` och vi ändrar `b` till `xm`. Detta upprepas tills intervallet (a, b) blivit kortare än `epsilon` eller tills vi råkar finna nollstället.

Funktionen `nollställe` kan nu användas för att beräkna nollställen till olika funktioner. Om vi t.ex. har funktionen

$$f(x) = 2x^3 - 3x^2 - 18x - 8$$

och vill beräkna och dess nollställe i intervallet $(-1, 1)$, kan vi skriva

```
x0 = nollställe(f, −1, 1)
```

Funktionen `f` definierar vi på följande enkla sätt:

```
def f(x):
        return 2*x**3 + 3*x**2 − 18*x − 8
```

8.7 Rekursiva funktioner

[överkurs]

Du har sett flera exempel på att en funktion kan anropa andra funktioner. Det är dessutom faktiskt tillåtet för en funktion att anropa sig själv. En sådan funktion kallas en *rekursiv funktion.* Det vanligaste exemplet på en sådan funktion (det brukar förekomma i alla böcker om programmering) är en funktion som beräknar *n!,* den s.k. *fakulteten* för ett tal *n.* Denna kan definieras på följande sätt:

$$n! = \begin{cases} 1 & \text{om } n = 0 \\ 1 \times 2 \times 3 \times ... \times n & \text{om } n > 0 \end{cases}$$

I uppgift 8.4 använde du denna definition för att skriva en funktion som räknade ut fakulteten för ett tal. Ett annat sätt att skriva definitionen av den matematiska funktionen *n!* är:

$$n! = \begin{cases} 1 & \text{om } n = 0 \\ n(n-1)! & \text{om } n > 0 \end{cases}$$

Det finns ett fall för vilket värdet är givet $(n = 0)$ och ett fall där man använder induktion för att uttrycka lösningen med hjälp av redan definierade värden. Det senare sättet att skriva definitionen leder naturligt till följande Python-funktion:

[fullständigt program]

```
def nfak(n):
        if n < 0:
                return None
        elif n == 0:
                return 1
        else:
                return n * nfak(n-1)
```

Det är den sista raden som är den mest intressanta. Där anropar funktionen `nfak` sig själv. Många som inte stött på rekursion tidigare brukar tycka att det är lite underligt. Då ska man komma ihåg en grundläggande regel: rekursiva funktionsanrop fungerar på *precis samma sätt* som andra funktionsanrop. Det finns inga specialregler för rekursion. Detta betyder att när man anropar en funktion (sig själv eller någon annan) beräknas först argumenten. Sedan genereras minnesutrymme för den anropade funktionens parametrar och argumenten kopieras dit. Vid *varje* nytt anrop genereras nytt minnesutrymme för parametrarna. Om en viss funktion är anropad flera gånger betyder detta att *varje* upplaga av funktionen har sin *egen upplaga* av parametrarna. Därefter utförs satserna i den anropade funktionen. När dessa satser är klara återvänder man till

den punkt varifrån funktionen anropades. Om en funktion anropar sig själv återvänder man alltså till en punkt inne i den aktuella funktionen.

Rekursiva lösningar följer alltid samma idé. Om man har ett problem identifierar man först ett (eller flera) enkla specialfall där lösningen är trivial (`n==0` i `nfak`). Därefter försöker man formulera om problemet, så att det i någon mening blir enklare (`(n-1) !` är enklare än `n!`). Man kan då tänka sig att den funktion man håller på att skriva redan finns och klarar av att lösa det enklare problemet.

Som exempel på detta ska vi konstruera en rekursiv funktion som får en sekvens som parameter och som skriver ut elementen i sekvensen baklänges. Den ser ut så här:

```
def skriv_baklänges(sek):
        if len(sek) > 0:
                skriv_baklänges(sek[1:])
                print(sek [0], end=', ')
```

160

Det triviala specialfallet är att sekvensen har längden noll. I så fall gör vi inget alls. Om längden är större än noll, finns det minst ett element i sekvensen. Då kan man tänka så här: Vad är det sista som ska göras? Då ska sekvensens sista element skrivas ut. Vad ska ske innan dess? Då ska alla elementen, utom det första, skrivas ut baklänges. Det är precis så funktionen fungerar. På den tredje raden skapar man en skiva bestående av alla element utom det första. Dessa element skrivs ut baklänges med hjälp av funktionen. (Vi får ju anta den finns och fungerar.) Sist skrivs element 0, det första, ut.

Uppgift 8.10

Skriv en rekursiv funktion som ger det största elementet i en sekvens. Om sekvensen saknar element ska värdet `None` ges. Använd följande rekursiva idé: Om sekvensens längd är 1, är det enda elementet också det största. Annars kan man räkna ut det största elementet i en sekvens som är ett element kortare än det urspungliga. Sedan jämför man resultatet man får med det borttagna elementets värde. Man kan då returnera det största av dessa båda värden.

8.8 Sammanfattning

Efter att ha läst detta kapitel bör du:

- kunna skriva en korrekt definition av en funktion,
- veta hur man anropar funktioner och förstå hur argumenten överförs till parametrarna,
- kunna konstruera enkla funktioner och kunna bilda fullständiga program som består av flera funktioner,
- veta vad lokala och globala variabler är, var de deklareras, var de är synliga och vilken livslängd de har,
- förstå vad som händer när man har parametar som är referenser,
- veta hur man anropar parametrar med deras namn,
- kunna använda parametrar med defaultvärden,

- veta hur man gör för att ha ett variabelt antal parametrar,

- (om du läst avsnitt 8.6) förstå och kunna använda referenser till funktioner och kunna bilda enkla lambda-uttryck,
- (om du läst avsnitt 8.7) förstå hur rekursion fungerar.

8.9 Övningar

I uppgifterna i detta kapitel ska du skriva funktioner. För att kunna provköra funktionerna måste du förstås i varje uppgift också skriva några extra rader, där du anropar den funktion du skrivit.

8.1 Skriv en funktion som beräknar vad en vara kostar, inklusive moms. Som parametrar ska funktionen få dels priset exklusive moms och dels momssatsen uttryckt i procent.

8.2 Skriv en funktion som får ett heltal i intervallet 1 till 10 som parameter. Funktionen ska skriva ut en multiplikationstabell för det angivna talet. Om man t.ex. ger talet 6 som parameter ska funktionen skriva ut tio rader där det på första raden står talen 1 och 6, på andra raden 2 och 12, på tredje 3 och 18 osv.

8.3 Skriv en funktion som undersöker hur många små resp, stora bokstäver som finns i en text. Resultaten ska ges i en tupel.

8.4 Skriv en funktion med namnet primtal som får ett heltal (> 0) som parameter och som returnerar ett värde av typen bool. Tips: Programmet på sidan 131 undersökte om ett tal var ett primtal.

8.5 Två positiva heltal *i* och *j* kallas för *relativa primtal* om det inte finns något heltal större än 1 som de båda är delbara med. (Talen 16 och 21 är t.ex. relativa primtal, men däremot inte talen 18 och 21 eftersom de båda är delbara med 3.) Skriv en funktion som undersöker om två positiva heltal är relativa primtal.

8.6 Skriv en funktion som roterar elementen i en lista ett steg åt höger. Element nr 0 ska flyttas till plats nr 1, element nr 1 till plats nr 2 osv. De sista elementen ska flyttas till plats nr 0.

8.7 De s.k. binomialkoefficienterna kan definieras för icke-negativa heltal *n* och *k* på följande sätt:
$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$

Skriv med hjälp av funktionen `nfak` från uppgift 8.4 en funktion som beräknar binomialkoefficienten för två tal. De två talen (*n* och *k*) ska ges som parametrar till funktionen.

8.8 Skriv en funktion som använder nedanstående Maclaurin-serie för att beräkna värdet av e^x. Tag inte med sådana termer i summan som är mindre än 10^{-7}.

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

8.9 För att beräkna kvadratroten till ett tal *x* kan man använda *Newtons metod* som fungerar på följande sätt:
Man börjar med att gissa ett tal$g >= 0$. När man har gissat *g* vet man att det måste finnas ett annat tal *h* sådant att $g \cdot h = x$. (Talet *h* kan alltså skrivas som $h = {}^{x}/_{g}$.) Om man har riktig tur och gissat bra så gäller att *g* och *h* är ungefär lika och då har man funnit lösningen. I normala fall gör man nog inte en så god gissning. En ny bättre gissning g_{ny} kan man då göra genom att bilda medelvärdet av *g* och *h*:

$$g_{ny} = \frac{g + \frac{x}{g}}{2}$$

Man kan nu ersätta *g* med g_{ny} och beräkna ett nytt tal *h.* Genom att bilda medelvärdet av de nya värdena *g* och *h* kan man få en ännu bättre gissning o.s.v. Använd denna metod för att konstruera en funktion som beräknar $\sqrt{x}$. Använd som första gissning värdet ${}^{x}/_{2}$ och låt gissningarna fortsätta tills skillnaden mellan två gissningar är mindre än 10^{-6}.

8.10 I Sudoku använder man en uppställning med 9×9 rutor. Spelplanen är dessutom indelad i 9 regioner, där varje region består av tre rader och tre kolumner. I en korrekt ifylld lösning får varje siffra 1 till 9 bara finnas en gång på varje rad, i varje kolumn och i varje region. Skriv en funktion som undersöker om en lösning är korrekt. Som parameter ska funktionen få en tvådimensionell lista med heltal.

[överkurs]
8.11 Skriv ett program som läser in ett antal ord och placerar dem i en lista. Programmet ska sedan, med hjälp av standardfunktionen

`sorted`, skapa en ny lista i vilken orden kommer i längdordning, så att det kortaste ordet kommer först och det längsta sist.

[överkurs]
8.12 En fackförening erbjuds ett flerårsavtal enligt följande modell:

- Första året (år 1) får varje arbetstagare en månadslön på 25000 kr.
- Följande år (år 2, 3, 4 osv.) får man varje år en ökning med 3 % av föregående års lön, samt dessutom ett generellt påslag med 900 kr.

Skriv en rekursiv funktion som beräknar månadslönen ett visst år. Som enda parameter ska funktionen ha ett årsnummer.

[överkurs]
8.13 Skriv en rekursiv funktion `sgd` som beräknar den största gemensamma divisorn till två positiva heltal *m* och *n* utgående från följande definition.

$$sgd(m,n) = \begin{cases} m & \text{om } m = n \\ sgd(m-n,n) & \text{om } m > n \\ sgd(m,n-m) & \text{annars} \end{cases}$$

9 Moduler och paket

När man skriver ett lite större program som består av flera funktioner är det opraktiskt att placera koden för alla funktionerna i samma fil. I detta kapitel får du lära dig principerna för hur man sätter samman program som

består av flera filer. Vi kommer att diskutera moduler, vilka kan innehålla flera funktioner, och paket som kan innehålla flera moduler.

9.1 Skapa moduler

I kapitel 1 diskuterade vi skillnaden mellan att köra i *interactive mode* och i *script mode.* I *interactive mode* skrev vi uttryck som direkt beräknades av Python-interpretatorn. I *script mode* sparade vi först vårt program i en fil, innan vi körde det i interpretatorn. Fördelen med att använda *script mode* är förstås att de program man skrivit finns kvar så att man kan köra dem igen vid ett senare tillfälle eller utveckla dem vidare.

De flesta exempel vi arbetat med så här långt har vi kört i *script mode.* Men än så länge har varje program bara bestått av en enda egen fil. En fil som innehåller Python-kod ska som vi sett ha ett namn som slutar på `.py`. En sådan fil kallas en *modul.* En modul får innehålla en eller flera definitioner, t.ex. av funktioner, samt exekverbara satser. Om man har flera funktionsdefinitioner i en modul, spelar det för det mesta ingen roll i vilken ordning de placeras. Det viktiga är att en funktions definition är känd när funktionen *anropas.*

Man startar ett Python-program i *script mode* genom att t.ex. skriva något av följande (beroende på operativsystem) i ett terminalfönster:

```
PY hej.py
python3 hej.py
```

eller genom att starta filen från en IDE, såsom vi beskrev i avsnitten 1.4 och 1.5. I detta exempel startas modulen `hej`. Den modul man startar programmet i kallas *main module.* Här bli alltså `hej` modulen `main`. I programmet får `main`-modulen automatiskt namnet `_main_`.

När en modul laddas in till interpretatorn, registreras de definitioner som finns i den och de utförbara satser som finns i modulen exekveras. Så har alla de program vi kört hittills i *script mode* fungerat.

Ett program kan byggas upp av flera moduler, inte bara modulen `main`. Det gör man genom att från `main`-modulen importera andra moduler. (Hur det går till ska vi diskutera i nästa avsnitt.) En modul innehåller ofta en eller flera funktioner. Dessa kan vara av generell karaktär och därför användbara i flera olika program, eller så kan de vara speciella för det program man konstruerar. I det senare fallet används modulen för att dela upp programmet så att det blir mer hanterligt.

Som exempel ska vi bilda en modul som innehåller tre funktioner för statistiska beräkningar. Funktionen `medelv` ger medelvärdet av ett antal tal, `stdav` ger standardavvikelsen och `median` ger medianen.

[fullständigt program]

```
# Modul för statistik
import math

def medelv(a):
        sum = 0
        for x in a:
                sum += x
        return sum / len(a)

def stdav(a):
        m = medelv(a)
        sum = 0
        for x in a:
                sum = sum + (x - m)**2
        return math.sqrt(sum / len(a))

def median(a):
```

```
        b = sorted(a) # skapa en sorterad kopia
        n = len(b)
        if n % 2! = 0:
                return b[(n-1)//2] # udda antal
element
        else :
                return (b[n//2-1] + b[n//2])/2 # jämnt
antal element
```

Alla tre funktionerna kan ha en lista eller tupel som parameter. Formeln för standardavvikelse finns i övning 6.5 på sidan 124. Vad som menas med median och hur man kan beräkna den beskrivs i övning 6.4. Lägg märke till att vi i beräkningen av medianen behöver sortera talen som

---

finns i sekvensen `a`. Vi gör detta i en lista `b` som är en kopia av `a`. På detta sätt påverkas inte sekvensen `a`.

Vi lagrar denna modul i en fil som vi ger namnet `statistik.py`. Detta innehär att modulen får namnet `statistik`. Man kan försöka köra modulen genom att t.ex. skriva

```
py statistik.py
```

men detta är ganska meningslöst eftersom modulen statistik inte innehåller några direkt exekverbara satser, utan bara definitioner. För att kunna använda modulen måste vi importera den till en annan modul, t.ex. modulen `main`. Detta ska vi göra i nästa avsnitt.

Vi ska visa en modul till. Det är en modul som beskriver en kortlek. (Förklaringar kommer efter koden.)

[fullständigt program]

```
# Modulen kortlek
import random

def ny(): # skapar en ny, blandad kortlek
        lek = []
        for i in range(1, 5):
                for j in range(1, 14):
                        lek.append((i, j))
        random.shuffle(lek)
        return lek

def ge(lek): # ger det överstakortet i lek
        if len(lek) > 0:
                return lek.pop()
        else:
                return None

färg = '\N{BLACK CLUB SUIT}\N{WHITE DIAMOND SUIT}'\
        '\N(WHITE HEART SUIT}\N{BLACK SPADE SUIT}'
valör = ('E', '2', '3', '4', '5', '6', '7',
        '8', '9', '10', 'Kn', 'D', 'K')

def visa(k, sist='\n'): # skriver ut kortet k
        f, v = k
         print(färg[f-1], valör[v-1], end=sist)
```

Vi beskriver ett spelkort med hjälp av en tupel med två element. Det första elementet är kortfärgens nummer. Numren 1 till 4 representerar

färgerna klöver, ruter, hjärter respektive spader. Det andra elementet i tupeln är kortets valör, ett tal mellan 1 och 13. Spader kung representeras t.ex. med tupeln (4, 13) och klöver ess med (1, 1).

Kortleken beskriver vi med en lista med spelkort, dvs. en lista där varje element är en tupel. För att skapa en ny kortlek anropar man funktionen `ny`. Denna börjar med att skapa en ny tom lista, `lek`. Därefter löper den igenom alla kombinationer av kortfärger och valörer och skapar för varje kombination en tupel som den lägger i listan `lek`. Slutligen anropas funktionen `shuffle` för att blanda korten slumpmässigt. Som resultat ges den blandade listan.

Funktionen `ge` delar ut det översta kortet i den kortlek den får som parameter. Funktionen returnerar alltså en tupel med två element. Det kort som delats ut kommer att tas bort från kortleken.

Den sista funktionen, `visa`, skriver ut ett kort. Som första parameter får den ett kort, dvs. en tupel `k`. Värdena i denna tilldelas, med hjälp av multipel tilldelning, till de två variablerna `f` och `v`, vilka då kommer att innehålla kortets färgnummer respektive valör. För att utskriften ska bli snygg används `str`-variabeln `färg` och tupeln `valör`. Dessa har placerats som globala variabler i modulen så att de inte behöver skapas igen varje gång funktionen `visa` anropas. Texten `färg` innehåller fyra element, vilka är teckenkoderna för symbolerna för de olika kortfärgerna och tupeln `valör` innehåller 13 valörer. Om man t.ex. anropar funktionen `visa` med parametern (1, 12) kommer den att skriva ut

♣ D

Modulen `kortlek` är ett exempel på en modul som både innehåller definitioner och direkt exekverbar kod. När modulen laddas in kommer koden som skapar de två variablerna `färg` och `valör` att exekveras. Koden i funktionerna exekveras inte förrän de anropas.

Uppgift 9.1

Skriv en modul som man kan använda för att spela spelet sten-sax-påse. Modulen ska ha två funktioner. Den första ska slumpmässigt returnera något av värdena `'Sten'`, `'Sax'` och `'Påse'` varje gång man anropar den. Den andra funktionen ska få två texter som

parametrar. Dessa texter ska innehålla något de tre värden den första funktionen ger. Funktionen ska avgöra vem som vann eller om det blev oavgjort.

Moduler

Ett program kan bestå av flera separata delar, moduler.

En modul kan innehålla både definitioner och direkt exekverbar kod.

Den modul man startar programmet från kallas *main module.*

Om en modul sparas i en fil med namnet `nnn.py` får den namnet *nnn.*

Undantag är `main`-modulen. Den heter `_main_`.

En moduls namn kan avläsas från variabeln `_name_`.

9.2 Importera moduler

För att man ska kunna använda funktioner och annat som finns i en modul måste man importera den. Det gör man genom att lägga in en `import`-sats i programmet. Man brukar normalt lägga `import`-satser först i programmet, men det är inte nödvändigt. Som exempel kommer här ett program som använder modulen `statistik`:

```
# Demo statistik
import statistik

s = input('Skriv ett antal tal: ')
talen = [float(e) for e in s.split()]
print('Medelvärde: ', statistik.medelv(talen))
print('Standardavvikelse:', statistik.stdav(talen))
print('Median: ', statistik.median(talen))
```

När man importerar en modul, kommer bara modulens namn att importeras till programmet och bli direkt synligt där. Vill man anropa en funktion som är definierad i modulen måste man skriva modulens namn och en punkt framför funktionsnamnet, såsom markerats med rött i programmet. Tycker man att modulnamnet är långt att skriva, kan man ge modulen ett annat namn i sitt program. Man skriver t.ex.

```
import statistik as st
```

Sedan använder man det korta namnet när man anropar funktionerna:

```
print('Medelvärde:', st.medelv(talen))
```

Man kan också importera ett funktionsnamn direkt, t.ex.

```
from statistik import medelv
```

Då behöver man inte ge modulnamnet när man anropar funktionen:

```
print('Medelvärde:', medelv(talen))
```

Det går till och med att importera alla definitioner från en modul. Man skriver då tecknet `*`:

```
from statistik import *
```

Då kan alla funktioner i modulen anropas utan att man skriver modulnamnet först:

```
print('Medelvärde: ', medelv(talen))
print('Standardavvikelse:', stdav(talen))
print('Median: ', median(talen))
```

Men man brukar inte rekommendera att man gör på det sättet, eftersom det då kan vara svårare att se var funktionerna finns.

Importera moduler

För att importera en modul och göra dess namn känt: `import modulnamn`

För att komma åt en definition d i modulen, t.ex. ett funktionsnamn: *`modulnamn.`*`d`

För att ge en modul ett annat namn i sitt program: `import modulnamn` as `annat_namn`

För att importera en viss definition `d` från en modul: `from modulnamn import d`

För att importera alla definitioner från en modul: `from modulnamn import *`

Här kommer också ett litet program som använder modulen `kortlek`. När man kör programmet frågar det gång på gång om man vill fortsätta. Svarar man med ett `j` eller ett `J` delar programmet ut ett kort. Frågan ställs i funktionen `fortsätta` som har en fråga som parameter.

[fullständigt program]

```
import kortlek

def fortsätta(fråga): # ställer en fråga
        s = input(fråga + '? ')
        return len(s) > 0 and (s [0] == ' j ' or s[0] ==
'J')
```

```
lek = kortlek.ny()
while fortsätta('Vill du fortsätta') and len(lek) > 0:
        kort = kortlek.ge(lek)
        print('Du fick ', end='')
        kortlek.visa(kort)
```

Så här kan det se ut när man kör programmet:

```
Vill du fortsätta? j
Du fick ♣ Kn
Vill du fortsätta? n
```

Det är inte bara funktioner som man kan komma åt från en modul som man importerar. Allt som är definierat på global nivå i modulen är åtkomligt. Man kan t.ex. komma åt variabeln `färg` i modulen `kortlek`:

```
print(kortlek.färg[2], 'är trumf')
```

När man importerar en modul söker Python-interpretatorn efter filen som innehåller modulen. Filen ska ha samma namn som modulen, fast med `.py` på slutet. Om interpretatorn inte hittar modulen, får man en felutskrift. Sökningen sker i följande ordning:

- Om man kör i *script mode* söker interpretatorn först i den mapp där `main`-modulen ligger. Om man kör i *interactive mode* söker interpretatorn i stället i den mapp där man startat interpretatorn.
- Interpretatorn söker sedan igenom de mappar och zip-filer som räknas upp i miljövariabeln `PYTHONPATH` (om den finns).
- Sist söker interpretatorn igenom ett antal förutbestämda mappar. Vilka dessa är bestämdes när man installerade Python.

Om man har lagt sina moduler i en eller flera egna mappar, kan man instruera Python-interpretatorn att söka också i dessa mappar. Detta åstadkommer man genom att skapa en miljövariabel med namnet `PYTHONPATH`, om det inte redan finns en sådan. I denna räknar man upp de mappar man vill ska genomsökas. Man kan också räkna upp zip-filer som innehåller moduler. Exakt hur miljövariabeln ska se ut och hur man skapar den beror på vilket operativsystem man kör. Den ska ha samma form som variabeln `PATH` i detta system. Vi hänvisar därför till dokumentationen för ditt operativsystem.

Vill man se vilka mappar som söks igenom kan man skriva:

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

Det finns ett antal moduler som ingår i standarddistributionen av Python. Dessa hittar interpretatorn i det tredje steget ovan. Vi har hittills använt modulerna `math`, `random` och `datetime`, men det finns många fler, t.ex. `sys` som används ovan. (Beskrivningar av dem hittar man på `https://docs.python.org/3/library/index.html`.) Man importerar standarmoduler på samma sätt som sina egna. För att t.ex. slippa skriva `math` framför anrop av funktionerna i `math` kan vi skriva:

```
from math import *
```

Det finns många moduler som man kan utöka sitt system med, t.ex. numpy för numeriska beräkningar. Hur man installerar extra moduler beskrivs på `https://docs.python.org/3/installing/index.html`.

Uppgift 9.2

Skriv ett program som använder modulen du konstruerade i uppgift 9.1. Programmet ska låta två spelare, spelare 1 och 2, spela spelet sten-saxpåse. En spelare i taget ska slumpmässigt få ett av värdena och sedan ska programmet tala om vilken av spelarna som vann eller om det blev oavgjort. Utforma programmet så att man kan köra flera gånger. Efter varje omgång ska programmet fråga om man vill spela en gång till.

9.3 Ett större exempel

Nu är det dags för ett större exempel som är uppbyggt av flera funktioner och som använder sig av en färdig modul. Vi ska skriva ett program som låter användaren spela kortspelet Tjugoett mot datorn. Spelet går till så att man får ett kort i taget. Efter varje kort får man avgöra om man vill ha ytterligare kort eller inte. Det gäller att försöka få summan av kortens valörer så nära 21 som möjligt utan att överskrida detta tal. Ess räknas som antingen 1 eller 14. Om man får över 21 förlorar man och datorn vinner. Om man stannar under 21 får också datorn dra ett kort i taget och efter varje kort avgöra om den ska forstätta eller inte. Om datorn får mer än 21 poäng eller lägre poäng än spelaren vinner spelaren, annars vinner datorn. Datorn vinner alltså på samma poäng. Så här kan det se ut när man kör programmet:

```
Välkommen til Tjugoett
Du fick ♣ 9 och har 9 poäng
Ett kort till? j
Du fick ♠ 8 och har 17 poäng
Ett kort till? n
Datorn fick ♣ 5 och har 5 poäng
Datorn fick ♠ E och har 19 poäng
Du förlorade
Nytt parti? n
```

Programmet ska naturligtvis använda sig av modulen `kortlek` från sidan 167. Här är hela programtexten. Förklaringar följer efteråt.

[fullständigt program]

```
# Tjugoett
import kortlek
```

```
def spela(lek, spelare): # En av spelarna spelar
        p = 0
        hand = []
        while True:
                k = kortlek.ge(lek)
                hand.append(k)
                p = poäng(hand)
                print(spelare, 'fick ', end = ' ')
                kortlek.visa(k, '')
                print(f' och har {p} poäng')
                if spelare == 'Du' and\
                        (p >= 21 or not fortsätta('Ett
kort till')) or\
                        spelare == 'Datorn' and p >= 17:
                        break
        return p

def poäng(hand): # Beräknar poängen för en hand
        p, antal_ess = 0, 0
        for k in hand:
                if k[1] == 1: # Ess?
                        p+ = 14 # Räkna ess som 14
                        antal _ ess + = 1
                else:
                        p+ = k[1]

        while p > 21 and antal_ess > 0:
                p− = 13 # Räkna Ess som 1
                antal_ess -= 1
        return p
```

```
def fortsätta(fråga): # Ställer en fråga
        s = input(fråga + '? ')
        return len(s) > 0 and (s [0] == 'j' or s[0] ==
'J')
```

```
# Här börjar exekveringen
print('Välkommen til Tjugoett')
while True:
        lek = kortlek.ny()
        p1 = spela(lek, 'Du') # Låt användaren spela
        if p1 > 21:
                print('Du förlorade')
        elif p1 == 21:
                print('Du vann')
        else:
                p2 = spela(lek, 'Datorn') # Låt datorn
spela
                if p2 <= 21 and p2 >= p1:
                        print('Du förlorade')
                else :
                        print('Du vann')
        if not fortsätta('Nytt parti'):
                break
```

Programmet startar med att skriva ut en välkomsthälsning. Därefter börjar en `while`-sats som löper ett varv för varje spelomgång. Först skapas en ny kortlek. Därefter låter man användaren spela genom att anropa funktionen `spela`. Denna har två parametrar: den första är en referens till kortleken som används och den andra är en text som anger vem som spelar. Som returvärde från funktionen `spela` får man antalet poäng som spelaren fick. (Hur funktionen `spela` fungerar internt diskuteras nedan.) Om användaren fick 21 poäng eller mer avgörs spelomgången direkt, annars låter man också datorn spela. Man anropar då funktionen `spela` igen. Därefter jämför man användarens och datorns poäng och avgör vem som vunnit.

Låt oss nu se på funktionen `spela`. Den innehåller en `while`-sats som löper ett varv för varje nytt kort spelaren drar från kortleken. De kort spelaren får placeras i en lista som heter `hand`. Varje gång spelaren fått ett nytt kort räknar man ut hur många poäng handen är värd. Detta görs i en separat funktion som heter `poäng`. (Den diskuterar vi strax.) Varje gång spelaren fått ett nytt kort visar man detta kort och skriver ut den aktuella poängen. Sedan avgörs om spelaren ska ha ett kort till eller inte. Är det användaren som är spelaren, frågar programmet helt enkelt

om man ska ha ett kort till. Då anropas funktionen `fortsätta` som vi sett tidigare. Är det datorn som är spelaren använder man strategin att stanna om man har 17 poäng eller mer.

Slutligen ska vi diskutera funktionen `poäng`. Den får en lista med kort som parameter och räknar ut hur många poäng dessa kort motsvarar. Det gör den genom att löpa igenom listan och summera kortens valörer. Den håller samtidigt reda på hur många ess som finns. Ett ess får nämligen räknas som antingen 14 eller 1. När valörerna summerats ser man om man kommit över 21 poäng. I så fall ser man om man har några ess och räknar om deras värde från 14 till 1.

Detta program är ett exempel på hur man kan göra ett ganska komplicerat problem mer hanterbart genom att använda flera funktioner, där varje funktion har en speciell och avgränsad uppgift.

9.4 Paket

I ett filsystem i ett vanligt operativsystem kan det finnas hundratals filer. För att det ska vara möjligt att hålla reda på alla dessa, använder sig filsystemen som bekant av mappar *(directories).* Varje mapp kan innehålla ett antal filer och olika slag, och den kan också innehålla andra mappar, s.k. undermappar *(subdirectories).* På detta sätt får filsystemet en trädstruktur.

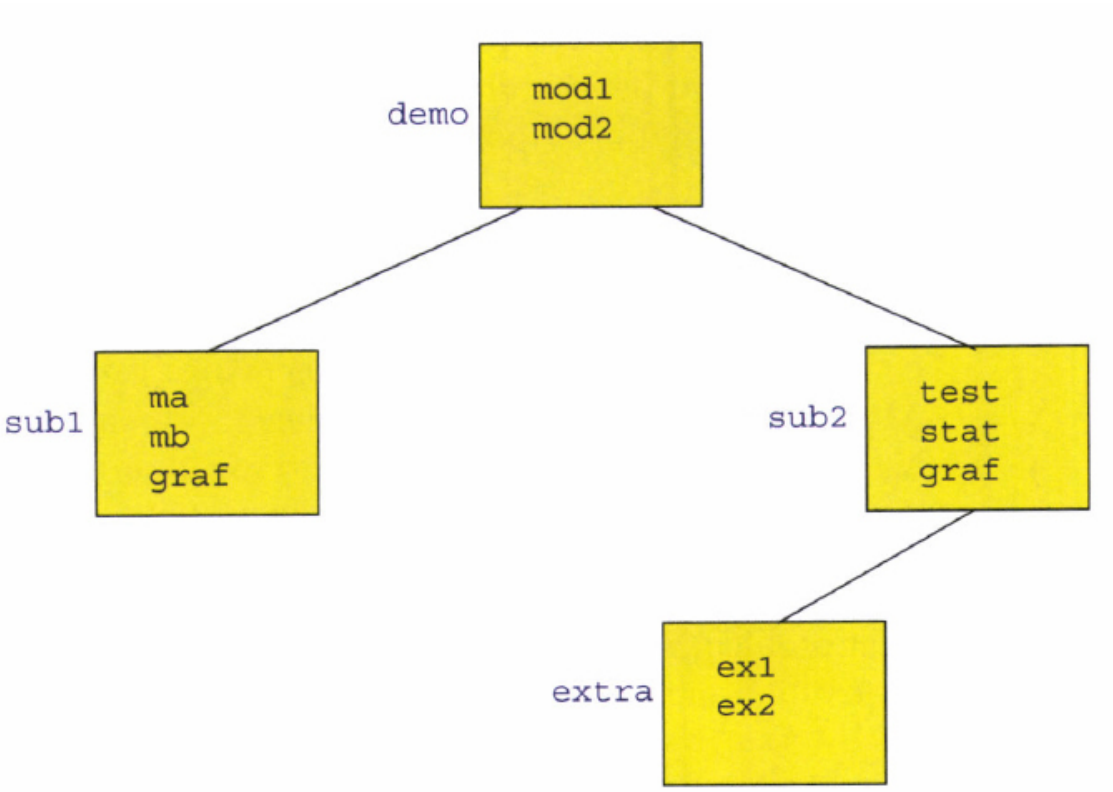
Samma teknik använder man i Python för att hålla reda på de moduler som kan finnas. Man använder sig av *paket* vilka organiseras på samma sätt som mappar i ett filsystem. Ett paket kan innehålla moduler (som motsvarar vanliga filer i ett filsystem) och subpaket (som motsvarar undermappar). I figur 9.1 visas ett exempel på en paketstruktur.

Vi har ett paket `demo`, som har två subpaket, `sub1` och `sub2`. Paketet `sub2` har i sin tur ett subpaket med namnet `extra`. Alla paketen innehåller

också moduler. Paketet `sub2` innehåller t.ex. modulerna `test`, `stat` och `graf`. Lägg märke till att man får ha moduler med samma namn om de ligger i olika paket.

Modulerna i ett paket namnges på samma sätt som filerna i ett filsystem, men man använder en punkt mellan namnen i stället för tecknet `\` eller `/`. I vårt figur 9.1 finns alltså bland annat modulen `demo.mod1`, subpaketet `demo.sub1` och modulen `demo.sub2.extra.ex1`.

Figur 9.1 Paketstruktur.



Figuren visar en paketstruktur med fyra paket: sub1 med ma, mb och graf; demo med mod1 och mod2; sub2 med test, stat och graf; och extra med ex1 och ex2.

När man lagrar sina filer med Python-program i filsystemet lagrar man dem så att deras placering motsvarar paketets struktur. Mapparna ska ha samma namn som paketen och filerna med moduler samma namn som modulerna, fast med `.py` på slutet. Om vi har paketstrukturen i figur 9.1 skapar vi alltså en mapp i filsystemet som heter `demo`. Mappen `demo` kan placeras var som helst i filsystemet, men den måste finnas på ett sådant ställe att Python-interpretatorn hittar den när den söker efter moduler. (Här används samma regler som vi beskrev på sidan 171.) I mappen `demo` lägger vi filerna `mod1.py` och `mod2.py`. Vi låter dessutom mappen `demo` innehålla två undermappar med namnen `sub1` och `sub2`. I mappen `sub1` lägger vi filerna `ma.py, mb.py` och `graf.py` och i mappen `sub2` filerna `test.py, stat.py` och `graf.py`. Vi låter slutligen mappen `sub2` innehålla en undermapp med namnet `extra` och i denna placerar vi filerna `ex1.py` och `ex2.py`.

Om man vill använda några moduler som finns i ett paket måste man importera dem, precis som vi gjort innan. Låt oss anta att vi ska skriva ett program och att vårt program ligger i samma mapp som mappen `demo`. Anta vidare att det finns en funktion med namnet `f` i modulen `mod1` i paketet `demo`. För att anropa denna skriver vi:

```
import demo.mod1
demo.mod1.f()
```

Vi kan i alternativt skriva

```
from demo import mod1
mod1.f()
```

Då behöver man bara ange modulnamnet i anropet. Vill vi göra det ännu enklare att anropa funktionen `f` kan vi skriva

```
from demo.mod1 import f
f()
```

Då behöver vi inte ge något modulnamn i anropet.

Anta att vi i stället vill använda någon modul som ligger i ett subpaket. Vi vill t.ex. anropa en funktion som heter `run` i modulen `stat` som ligger i subpaketet `sub2`. Då kan vi skriva:

```
import demo.sub2.stat
demo.sub2.stat.run()
```

Lägg märke till att vi då måste ge det fullständiga namnet i funktionsanropet. Det enklare alternativet är att skriva

```
from demo.sub2.stat import run
run()
```

Vill vi importera alla funktioner i en viss modul på en gång, kan vi precis som förut använda tecknet `*`.

```
from demo.sub2.stat import *
```

[överkurs]

Man kan nu ställa sig frågan: Går det att importera alla moduler från ett visst paket på en gång? Kan vi t.ex. skriva på följande sätt för att importera de tre modulerna `test, stat` och `graf`?

```
from demo.sub2 import *
```

Svaret är att det går att skriva på detta sätt, men att det inte fungerar utan vidare. För att interpretatorn ska kunna veta vilka moduler som ingår i ett visst paket måste man ha lagt i denna information i paketstrukturen. Det gör man genom att i ett paket lägga in en extra fil som ska heta `_init_.py`. Denna fil exekveras när paketet laddas in. Filen kan innehålla vilken Python-kod som helst, men för att det ska gå att använda tecknet `*` som ovan måste filen innehålla en variabel som heter `_all_`. Denna ska vara en lista med namnen på de moduler som ingår i paketet. Om vi sparar följande fil i mappen `sub2`

```
# Filen _init_ i paketet sub2
_all_ = ['test', 'stat', 'graf']
```

kan vi nu t.ex. skriva

```
from demo.sub2 import *
stat.run()
```

```
test.start()
```

(Här antar vi att det finns en funktion med namnet `start` i modulen `test`.)

Det kan vara lämpligt att lägga en fil med namnet `_init_.py` i alla mappar med paket. Den är nämligen en markör för Python-interpretatorn, så att den vet att mappen innehåller ett Python-paket. Filen kan i det enklaste fallet vara tom.

9.5 Sammanfattning

Efter att ha läst detta kapitel bör du:

- veta vad en modul är och hur man skapar moduler,
- veta hur man importerar moduler till sitt program,
- förstå vad paket är och hur de lagras i filsystemet,
- veta hur man importerar moduler från paket.

9.6 Övningar

9.1 Konstruera en modul, `area`, som innehåller funktioner för att beräkna areor av rektanglar, cirklar och trianglar.

9.2 Skriv en modul som kan användas för att hålla reda på aktuell temperatur. Modulen ska ha fyra funktioner. Den första, `observation`, ska användas för att rapportera in uppmätt temperatur. Den har tre parametrar: timmar och minuter (d.v.s. tidpunkten för mätningen) samt temperaturen. De tre funktionerna `aktuell_temp`, `obs_tim` och `obs_min` ska som resultat ge den senast inrapporterade temperaturen och tidpunkten för denna. Låt funktionen `observation` kontrollera att de rapporterade värdena

är korrekta och rimliga. Den kan returnera ett sanningsvärde som anger om det gick bra eller inte.
Skriv också en `main`-modul i vilken man kan antingen rapportera in en avläst temperatur eller ber att få reda på senast inrapporterade temperatur och tidpunkt.

9.3 Ibland vill man kunna jämföra två spelkort med varandra. Hur denna jämförelse ska göras beror på vilket kortspel det gäller. I vissa spel räknar man bara kortens valörer, medan man i andra spel även tar hänsyn till kortens färger. Utöka modulen `kortlek` med en funktion `jfr` som kan användas när man ska jämföra spelkort. Denna ska kunna användas på samma sätt som funktionen `alfa` användes när vi jämförde texter. (Se sidan sidan 150.) När två kort ska jämföras ska i första hand färgerna jämföras. I dennas uppgift kan du använda ordningen i bridge, vilken är klöver, ruter, hjärter och spader. Om två kort har samma färg, jämförs deras valörer. Ess räknas som högsta kort.
Testa sedan din funktion genom att skriva ett program som upprepade gånger, tills korten tar slut, drar de två översta korten från en blandad kortlek. Porgrammet ska jämföra de två korten och tala om vilket som räknas som högst.

10 Felhantering


Resume

Det är inte lätt att programmera. Alla gör fel, även vana programmerare. När man ska lära sig programmera är det därför viktigt att man också lär sig att hitta felen i sina program. Man kan bara lära sig detta genom att själv konstruera, provköra och rätta program.

10.1 Olika typer av fel

Det kan finnas tre olika typer av fel i ett program:

- *Syntaxfel.* Dessa är fel som uppstår eftersom man inte har följt språkreglerna. Denna typ av fel upptäcks i Python för det mesta när man skriver in sitt program. Syntaxfel diskuterades i avsnitt 2.9 på sidan 41. I de kompilerade programspråken, t.ex. Java och C, upptäcks syntaxfel vid kompileringen och kallas ofta där *kompileringsfel.*

- *Exekveringsfel.* Dessa är fel som uppstår när man kör programmet. Programmet kan vara korrekt rent språkligt, men ändå innehålla fel som gör att det inte kan fortsätta normalt. Exempel på sådana fel är försök att indexera utanför gränserna i en sekvens och försök att omvandla en text som innehåller bokstäver till ett numeriskt värde. En del slag av fel, t.ex. typfel, försök att använda odefinierade variabler eller misslyckade försök att importera moduler, upptäcks i kompilerade programspråk redan vid kompileringen. Sådana fel betraktas där som kompileringsfel. Men i Python upptäcks dessa fel inte förrän man kör programmet, och uppträder därför som exekveringsfel.

- *Logiska fel.* Dessa är fel som beror på att man helt enkelt har tänkt fel när man har konstruerat programmet. Man har t.ex. använt en felaktig algoritm. Denna typ av fel är svårast att hitta,

eftersom programmet både går att skriva in och att köra utan att man får några felutskrifter. Ett logiskt fel visar sig först vid provkörning av programmet, genom att man får ett resultat som inte är korrekt.

Olika typer av fel

Tabellbeskrivning

Tabellen har 3 rader och 2 kolumner.

syntaxfel	Språkreglerna har inte följts. Upptäcks när man skriver in programmet (eller kompilerar det).
exekveringsfel	Felsignal uppstår vid körning. Kan avbryta programmet.
logiskt fel	Felaktig lösning av problemet. Ger felaktigt resultat.

10.2 Ett exempel

Här ska du få se ett enkelt exempel där det finns både ett syntaxfel och ett logiskt fel. Avsikten är att skriva ett program som läser in en text och som kontrollerar att den inlästa texten bara innehåller siffror, dvs. att texten kan tolkas som ett matematiskt tal.

Syntaxfel

Den första versionen av programmet ser ut på följande vis. Med avsikt har fel lagts in.

```
# Ett avsiktligt felaktigt program, version 1
s = input('Skriv ett tal: ')
for e in s:
        if e >=‘0‘ and <=‘9‘ :
                print('Talet är OK')
        else :
                print('Inget tal')
```

Programmet läser först in en text och placerar texten i `str`-variabeln `s`. Därefter följer en `for`-sats som ska löpa igenom hela texten. För varje tecken i texten ska man undersöka om tecknet är en siffra, dvs. om tecknet ligger i intervallet `'0'` till `'9'`. När man försöker skriva in programmet kommer man att få en felmarkering på den fjärde raden (den som börjar med `if`) och felmeddelandet: `Syntax error`.

Man har alltså fått fel av det första slaget, *syntaxfel.* Markören visar var felet finns. Avsikten är att testa om tecknet `e` är större än eller lika med `'0'` och mindre än eller lika med `'9'`. Så kan man uttrycka sig i ord, men i programtexten kan man inte bilda uttryck på det sättet. Det korrekta sättet att skriva är

```
if e >='0' and e <='9':
```

Det måste alltså stå ett fullständigt logiskt deluttryck på båda sidor om operatorn `and`.

I Python hade det alternativt gått att skriva

```
if '0' <= e <='9':
```

Men detta skrivsätt är inte tillåtet i andra vanliga programspråk.

Logiska fel

Nu kan man provköra programmet. Man testar med texten `12a3` som indata. Eftersom texten innehåller en bokstav borde programmet säga att talet är felaktigt, men när man kör programmet ser det ut så här:

```
Skriv ett tal: 12a3
Talet är OK
Talet är OK
Inget tal
Talet är OK
```

Det är något som inte stämmer. Programmet innehåller ett *logiskt fel.*

Programmeraren har tänkt fel. `for`-satsen löper ett varv för varje tecken i `str`-variabeln `s`. På varje varv undersöks om ett av tecknen är en siffra. Om så är fallet skriver man ut texten *Talet är OK,* annars skriver man *Inget tal.* Men detta kommer att upprepas för *varje* tecken i det inlästa talet! Det är därför flera utskrifter dyker upp efter varandra.

Hur löser man då detta? Idén är att man löper igenom texten tills man hittar ett tecken som *inte* är en siffra. Det räcker med att ett enda tecken inte är en siffra för att talet ska betraktas som felaktigt. Därför ska man se till att genomlöpningen avbryts om man hittar ett sådant tecken. Detta kan man enklast göra med en `break`-sats.

Utskriften med svaret ska inte visas förrän *efter* det att `for`-satsen genomlöpts. Då kommer den bara att visas en gång. Men man måste då veta vilken text som ska visas; den som säger att talet var OK eller den som säger att det inte var det. Denna typ av problem löser man enklast med hjälp av en variabel av typen `bool`. Före första varvet i `for`-satsen sätter man variabeln till värdet `True`, vilket betyder att talet än så länge är OK. Om man på något varv hittar ett tecken som inte är en siffra ändrar man variabeln till `False`. När `for`-satsen genomlöpts innehåller

---

därför variabeln fortfarande värdet `True` om talet var korrekt. Med dessa riktlinjer kan man skriva om programmet:

```
# Version 2
s = input('Skriv ett tal: ')
ok = True
for e in s:
```

```
        if e<'0' or e>'9':
                ok = False
                break
if ok:
        print('Talet är OK')
else :
        print('Inget tal')
```

Man har här vänt på villkoret i den första `if`-satsen eftersom man vill undersöka om tecknet `e` *inte* innehåller en siffra.

Detta program går bra att kompilera. Om man provkör det med samma indata som tidigare, fungerar det perfekt.

```
Skriv ett tal: 12a3
Inget tal
```

Uppgift 10.1

Varför måste man krångla till det med variabeln `ok`? Räcker det inte med att lägga en `break`-sats i `else`-delen av `if`-satsen i version 1 av programmet?

Uppgift 10.2

I flera av de program du skrivit tidigare har du läst in numeriska indata. Provkör ett sådant program, t.ex. programmet i uppgift 2.7 på sidan 38, och skriv avsiktligt ett felaktigt tal. Du kan t.ex. låta talet innehålla en bokstav. Studera vad som händer och se vilken utskrift du får.

Uppgift 10.3

[överkurs]

Skriv en version av programmet i detta avsnitt där du använder en `for`-sats med `else`-del.

10.3 Automatiskt skapade felsignaler

I resten av detta kapitel ska vi studera fel av kategorin exekveringsfel. När man exekverar ett program uppstår det ibland situationer som normalt inte borde inträffa. När ett exekveringsfel uppstår genereras en felsignal (en s.k. *exception* på engelska). Om man inte gör något särskilt i programmet får man då en felutskrift och programmet avbryts.

När man konstruerar ett program vill man utgå från en algoritm som är så ren som möjligt. Om man efter varje steg i sin algoritm skulle lägga in kontroller av alla slag av tänkbara fel och andra onormala händelser, skulle programkoden bli ganska klumpig och svår att förstå. I Python finns en mekanism med vars hjälp man kan hantera felsignaler, s.k. *exceptions.* En funktion i vilken ett fel uppstår kan generera en felsignal som sedan, enligt bestämda regler, skickas

vidare till andra funktioner, där man kan "fånga" felsignalen och vidta någon lämplig åtgärd.

När ett exekveringsfel inträffar genererar Python-interpretatorn automatiskt en felsignal. Anta att vi försöker köra följande programrader:

```
def plus(a, i, j):
        return a[i] + a[j]
v = [4, 3, 6]
x = plus(v, 1, 3)
```

Vi får då en utskrift där de sista raderna ser ut ungefär så här:

```
File "c:\PythonExempel\excep1.py", line 5, in
<module>
        x = plus(v, 1, 3)
File "c:\PythonExempel\excep1.py", line 2, in plus
        return a[i] + a [j]
IndexError: list index out of range
```

Man får en s.k. *traceback* (eller *stack trace)* där man kan följa vilken väg programmet tagit. På sista raden anges vilken typ av fel det var. Här har vi fått en automatiskt genererad felsignal av typen `IndexError`.

Här kommer ett annat exempel. Anta att vi har skrivit programraden

```
x = y
```

och glömt att skapa variabeln `y` tidigare. Då får vi utskriften

```
NameError: name 'y' is not defined
```

Här får felsignalen typen `NameError`.

Här är ytterligare ett exempel. Vi har skrivit programraderna

```
s = input('Skriv ett tal: ')
x = float(s)
```

När vi kör programmet skriver användaren `12,5`. Vi får du utskriften

```
ValueError: could not convert string to float:
'12,5'
```

Felsignalen blir av typen `ValueError`, eftersom det inte får finnas något kommatecken i ett numeriskt värde. (Det ska vara en punkt.)

Det finns många fler standardtyper för felsignaler, t.ex. `TypeError`, `ZeroDivisionError`, `ModuleNotFoundError` och `FileNotFoundError`. Alla dessa kan genereras automatiskt av Python-interpretatorn.

10.4 Generera felsignaler

Ett sätt att generera felsignaler är att använda en `assert`-sats. En sådan har formen

```
assert uttryck, (parametrar)
```

Denna sats kontrollerar att *`uttryck`* har värdet `True`. I så fall fortsätter exekveringen normalt. Om *`uttryck`* är `False`, genereras en felsignal av typen `AssertionError`. Man kan lägga till en eller flera parametrar, t.ex. ett felmeddelande. Som exempel kan vi ta funktionen `är_skottår` på sidan 137. Den skulle avgöra om ett visst år var ett skottår. Sverige övergick inte till den Gregorianska kalendern förrän år 1754. Funktionen `är_skottår` fungerar därför bara från och med detta årtal. Vi lägger därför in en kontroll av parametern med hjälp av en `assert`-sats:

[fullständigt program]

```
# Skottår
def är_skottår(år):
        assert år >= 1754, ('För tidigt årtal', år)
        return (år % 4 == 0 and år % 100! = 0) or år %
400 == 0
```

Om vi nu försöker anropa funktionen med ett årtal som är tidigare än 1754, t.ex. år 1740, får vi utskriften

```
AssertionError: ('För tidigt årtal', 1740)
```

Uppgift 10.4

Lägg in kontroller av att radien är större än noll i de två funktioner du konstruerade i uppgift 8.1 på sidan 141.

Man kan också generera felsignaler av vilken typ som helst genom att använda en `raise`-sats. De olika formerna framgår av faktarutan.

`raise`-sats och `assert`-sats

Här betecknar *`E`* en exception-klass: *`parametrar`* blir en tupel med information om felet.

Tabellbeskrivning

Tabellen har 5 rader och 2 kolumner.

raise *`E`*	ger en felsignal av typen *`E`*
raise *`E (parametrar)`*	ger en felsignal av typen *`E`*
raise	skickar vidare en felsignal
assert *`uttryck`*	
assert *`uttryck, (parametrar)`*	

ger en felsignal av typen `AssertionError` om *`uttryck`* är False

Här kommer ett exempel. I funktionen `byt_elem` på sidan 148 skulle den första parametern `v` vara en lista. För att kontrollera detta kan vi lägga in följande rader först i funktionen:

```
if not type(v) is list:
        raise ValueError('Felaktig första
parameter')
```

(Vi har inte sett operatorn `is` förut. Den används för att jämföra objekt. Mer om detta följer i ett senare kapitel.) Om vi nu anropar funktionen `byt_elem` och t.ex. ger en tupel som första argument, får vi utskriften

```
TypeError: Felaktig första parameter
```

I nästa exempel vill vi kontrollera att en variabel med namnet `a` innehåller en text som kan vara en mejladress. Det måste finnas exakt ett `@`-tecken i texten och minst en punkt efter `@`-tecknet.

```
if not type(a) is str\
        or a.count('@') != 1 or a.rfindt'.') < a
.find('©'):
                raise NameError('Felaktig
```

```
mejladress: ', a)
```

Om `a` t.ex. innehåller texten `'x. yz@abc'`, får vi utskriften

```
NameError: ('Felaktig mejladress: ', 'x.yz@abc')
```

10.5 Ta hand om felsignaler

Hittills har vi bara diskuterat hur man genererar felsignaler. Om ingen tar hand om en felsignal stannar programmet. För vissa tillämpningar är det emellertid inte acceptabelt att programmet bara stannar om något fel inträffar. Programmet måste kanske vara i gång hela tiden. Då måste det kunna ta hand om det som skett, t.ex. genom att ge en varningsutskrift till en operatör eller genom att stänga av någon kritisk funktion. Det är för det mesta inte heller acceptabelt att ett program stannar för att användaren råkar skriva felaktiga indata.

För att ta hand om felsignaler använder man en speciell s.k. `try`-sats. Det är en ganska komplicerad sats som man kan sätta ihop på flera sätt. Hur den fungerar förklaras bäst med exempel. Här kommer ett program där vi vill läsa in ett antal tal och beräkna deras medelvärde. Själva beräkningen av medelvärdet sker i funktionen `medelv` som vi importerar från modulen `statistik` på sidan 166. I programmet har vi lagt in en `try`-sats för att kunna göra felkontroller.

Vi läser in talen till en lista med namnet `talen` på det sätt som beskrevs i avsnitt 6.3.

```
# try_demo
from statistik import medelv
try:
        s = input('Skriv ett antal tal: ')
        ls = s.split()
        talen = [float(t) for t in ls]
        print('Medelvärdet av talen är:',
medelv(talen))
except ValueError:
        print('Felaktiga tal:', s)
except KeyboardInterrupt:
        print('Programmet avslutas')
        exit()
print('Programmet fortsätter ...')
```

En `try`-sats består av en `try`-del och en eller flera `except`-delar. Den eller de satser som är "farliga", dvs. de som kan generera felsignaler, ska placeras i `try`-delen. Om inget fel uppstår, utförs satserna i `try`-delen på normalt sätt och sedan fortsätter programmet med första sats efter `except`-delarna. Om däremot ett fel uppstår, avbryts exekveringen i `try`-delen omedelbart och programmet hoppar till den `except`-del som

kan ta hand om felet. I varje `except`-del anger man vilken eller vilka typer av fel `except`-delen kan ta hand om. I programmet ovan finns två `except`-delar. Den första kan ta hand om fel av typen `ValueError` och den andra kan ta hand om fel av typen

`KeyboardInterrupt`. För att demonstrera hur det fungerar provkör vi programmet:

```
Skriv ett antal tal: 2 3 5 4
Medelvärdet av talen är: 3.5
Programmet fortsätter ...
```

Här exekverar programmet helt normalt. Om vi i stället skriver in ett felaktigt tal kan det se ut så här:

```
Skriv ett antal tal: 2 3 z 4
Felaktiga tal: 2 3 z 4
Programmet fortsätter ...
```

Här uppstod ett fel av typen `ValueError` på den tredje raden i `try`-delen när vi försökte göra om de element vi läst in, från text till `float`. Då avbröts exekveringen i `try`-delen och programmet hoppade till den första `except`-delen. Där skriver det ut variabeln `s` som innehåller den inlästa texten. Ett problem är att vi inte direkt kan se vilket av de inlästa talen som var felaktigt. Vi kan få mer information genom att lägga till ett *argument* i den första `except`-delen. Varje avbrottssignal har nämligen ett argument som innehåller information om vad som orsakade felet. Denna information är för det mesta en text.

```
except ValueError as e:
        print('Felaktiga tal:', s)
        print(e)
```

Om vi nu kör en gång till med samma felaktiga indata, ser det ut så här:

```
Skriv ett antal tal: 2 3 z 4
Felaktiga tal: 2 3 z 4
could not convert string to float: 'z'
Programmet fortsätter ...
```

Vi försöker köra en gång till. Denna gång skriver vi Ctrl-c när programmet frågar efter ett tal.

```
Skriv ett antal tal: här skriver vi Ctrl-c
Programmet avslutas
```

Nu hamnar vi i den andra `except`-delen eftersom `input` genererar en händelse av typen `KeyboardInterrupt` när man skriver Ctrl-c.

Vi provkör programmet `try_demo` ytterligare en gång. Denna gång trycker vi direkt på *Enter* när programmet frågar efter talen. Då får vi följande felutskrift och programmet avbryts.

```
ZeroDivisionError: division by zero
```

Vad som händer är att variabeln `s` kommer att innehålla en tom text, vilket innebär att variabeln `talen` blir en tom lista. När man anropar

funktionen `medelv` kommer den därför att få en lista med längden noll som parameter. Inne i funktionen `medelv` summerar man talen i listan och dividerar sedan med listans längd för att beräkna medelvärdet. Då kommer man att försöka dividera med noll. När detta sker genereras en felsignal av typen `ZeroDivisionError`.

Om det uppstår en felsignal i en funktion och den inte fångas upp i en `try`-sats, kommer felsignalen att skickas vidare till det ställe där funktionen anropades. Eftersom funktionen `medelv` inte har någon `try`-sats som fångar fel av typen `ZeroDivisionError`, kommer felsignalen att skickas vidare till den fjärde raden i `try`-satsen i vårt program. Men eftersom `try`-satsen inte har någon `except`-del som hanterar fel av typen `ZeroDivisionError`, kommer felet att skickas vidare ytterligare ett steg uppåt, till det ställe där programmet `try_demo` anropades. Detta skedde från själva Python-interpretatorn. När en felsignal uppstår i denna har den som standardåtgärd att ge en felutskrift och avbryta programmet. Det var alltså det som hände när vi körde programmet sista gången.

Vill vi undvika att ett program avbryts kan vi se till att vi fångar alla typer av felsignaler. Det kan vi göra genom att lägga in en `except`-del allra sist. I denna kan vi ange att vi ska ta hand om felsignaler av typen `Exception`:

```
except Exception as e:
        print(e)
        print(type(e))
```

Alla felsignalstyper ärver egenskaper från typen `Exception`. Detta betyder bl.a. att en `except`-del som fångar felsignaler av typen `Exception` fångar *alla* felsignaler, vilken typ de än har. Det är därför man måste lägga denna `except`-del sist, i `try`-satsen.

Här har vi lagt in en utskrift av felsignalens argument och vi skriver dessutom ut vilken typ felsignalen har. Om vi nu kör programmet och ännu en gång bara trycker på *Enter* kommer det att se ut så här:

```
Skriv ett antal tal: här trycker vi på Enter
division by zero
<class 'ZeroDivisionError'>
Programmet fortsätter ...
```

Vi kan göra funktionen `medelv` lite säkrare genom att lägga in en `try`-sats också i den:

```
def medelv(a) :
        try:
                sum = 0
                for x in a:
                        sum += x
                return sum / len(a)
        except:
                print('Felaktig parameter till
medelv')
                raise # skickar vidare felsignalen
```

Vi har lagt in de programrader som gör beräkningen i `try`-delen. Vi har sett att det kan uppstå fel av typen `ZeroDivisionError`, men det kan också uppstå fel av typen `TypeError` om parametern `a` inte är en lista eller en tupel. Vi har bara en `except`-del. I denna har vi inte angivit vilken typ av felsignaler den kan ta hand om. Då kommer den att ta hand om *alla* felsignaler.

Det är ofta inte helt självklart vad man ska göra när ett fel uppstått. Om det uppstår något fel inne i en funktion kan man förstås strunta i felet. Då kommer som vi sett felsignalen att skickas vidare till det ställe varifrån funktionen anropades. Man kan också inne i en funktion försöka fånga de fel som kan uppstå. Det är det vi gjort nu i funktionen `medelv`. Frågan är bara vad som är vettigt att göra när man upptäckt ett fel. Om det är en funktion som normalt ska returnera ett värde kan man välja ett speciellt värde som markerar att något gick fel. Vi hade t.ex. kunnat låta funktionen `medelv` returnera värdet `None`. Problemet med detta är att det tvingar den som anropar funktionen att efter varje anrop testa om den fick det speciella värdet som resultat. Då är det bättre att göra såsom vi gjort här i funktionen `medelv`. Vi fångar upp felet, skriver ut ett meddelande om var felet uppstod och skickar sedan vidare felsignalen. Det gör vi med en `raise`-sats där vi bara skriver ordet `raise`.

---

try-sats

```
try:
    satser

except E1:
    satser

except E2 as a:
    satser

except (E3, E4, E5 ...) as a:
```

```
    satser

    ...

relse:
    satser

finally:
    satser
```

Man får ha hur många `except`-delar som helst (men minst en).

`E1, E2` etc. är felsignalens typ, t.ex. `ValueError`.

Om typnamnet är `Exception` eller om det utelämnas, fångas alla typer av felsignaler.

`a` är det argument felsignalen fick när den genererades.

Det innehåller normalt ett felmeddelande. Argumentet kan utelämnas.

Om ingen felsignal uppstår i `try`-delen, utförs ingen `except`-del.

Om en felsignal uppstår i satserna i `try`-delen, sker hopp till den första `except`-del som matchar felsignalens typ.

När satserna i `except`-delen har exekverats, avslutas `try`-satsen och exekveringen fortsätter med nästa sats.

`else`- och `finally`-delarna kan utelämnas (vilket normalt görs).

Om det finns en `else`-del, exekveras satserna i denna bara om det inte uppstått något fel i `try`-delen.

Om det finns en `finally`-del, exekveras alltid satserna i denna allra sist, oberoende av om det uppstått någon fel eller inte.

Vi kör nu programmet `try_demo` ännu en gång och trycker på *Enter* direkt. Då kommer det att se ut så här: (Den röda raden har skrivits ut i funktionen `medelv` och de andra raderna i `try_demo`.)

```
Skriv ett antal tal: här trycker vi på Enter
Felaktig parameter till medelv
division by zero
<class 'ZeroDivisionError'>
Programmet fortsätter ...
```

Här kommer ett exempel till. Det är en funktion som undersöker om den parameter den får kan omvandlas till typen `float`.

```
def isfloat(p):
        try:
                float (p)       # ger felsignal om
det misslyckas
                return True
        except:
                return False
```

Vi har nu demonstrerat de vanligaste formerna av `try`-satsen. I faktarutan visas fler varianter.

10.6 Kontroll av indata

Lägg märke till att man aldrig kommer tillbaka in i `try`-delen om ett fel inträffar där. Vill man upprepa koden i `try`-delen måste man lägga hela `try`-satsen i en repetitionssats. Här kommer en funktion `läs_tal` som visar hur det kan se ut. Funktionen försöker läsa in ett tal. Om användaren skriver fel upprepas försöket tills ett korrekt tal skrivits in eller tills användaren skriver Ctrl-c.

[fullständigt program]

```
def läs_tal():
        while True:
                try:
                        s = input('Skriv ett tal:
')
                        x = float(s)
                        return x
                except ValueError:
                        print('Felaktigt tal:', s,
'Försök igen')
```

Vi kan t.ex. anropa funktionen på följande sätt

```
x = läs_tal()
print('Talet i kvadrat är:', x*x)
```

Så här kan det se ut när man kör programraderna:

```
Skriv ett tal: z
Felaktigt tal: z Försök igen
Skriv ett tal: 4
Talet i kvadrat är: 16.0
```

I funktionen `läs_tal` har vi valt att inte fånga fel av typen `KeyboardInterrupt`. Det betyder att om användaren skriver Ctrl-c, skickas felsignalen vidare till det ställe där `läs_tal` anropades. Man får alltså där bestämma om man ska låta programmet avslutas eller inte.

Vi avslutar detta kapitel med att visa en funktion `läs_lista` som läser in ett antal tal och lägger dem i en lista. Listan ges som resultat. Man kan t.ex. anropa funktionen `läs_lista` på följande sätt:

```
a = läs_lista()
print(a)
```

Om man kör dessa programrader kan det t.ex. se ut så här:

```
Skriv ett tal per rad. Avsluta med en tom rad.
> 5
> 3
> z
Felaktigt tal: z Försök igen
> 2
>
[5.0, 3.0, 2.0]
```

Här följer funktionen:

```
def läs_lista():
        print('Skriv ett tal per rad. Avsluta med
en tom rad.')
        v = []
        while True:
                s = input('> ')
                if s == ' ' :
                        return v # listan klar
                try:
                        x = float(s)
                        v.append(x) # talet OK,
lägg till listan
                except ValueError:
                        print('Felaktigt tal:', s,
'Försök igen')
```


10.7 Sammanfattning

Efter att ha läst detta kapitel bör du:

- veta vilka olika kategorier av fel som finns,
- förstå varför det är viktigt att testköra ett program noga, trots att man inte får några felutskrifter, vare sig när man skriver in eller kör programmet,

- kunna tolka felutskriften som ges när det uppstår ett exekveringsfel,
- veta hur man kan använda `assert`,
- veta hur man kan generera felsignaler,
- kunna använda `try`-satser för att hand om (fånga) felsignaler,
- veta hur man gör för att kontrollera numeriska indata.

10.8 Övningar

10.1 Lägg in en `assert`-sats som kontrollerar att alla parametrarna är korrekta i funktionen `fyll` i uppgift 8.8 på sidan 149.

10.2 Lägg in `try`-satser i funktionerna `stdav` och `median` i modulen `statistik` på sidan 166 för att kontrollera parametern.

10.3 I övning 8.10 på sidan 162 skulle man skriva en funktion som undersökte om en spelplan i Sudoku var ifylld på rätt sätt. Komplettera funktionen med en `assert`-sats som kontrollerar att spelplanen har den rätta storleken.

[fullständigt program]
10.4 En rekursiv funktion kan förbruka mycket minne, eftersom varje nytt rekursivt anrop skapar en egen upplaga av lokala variabler och parametrar. Om minnet tar slut, genereras en felsignal av typen `RecursionError`. Provkör den rekursiva funktionen `nfak` på sidan 159 och undersök hur stora `n`-värden du kan ge som argument innan detta inträffar.

11 Textfiler

Data som man lägger i variablerna i ett program finns kvar bara så länge som programmet exekveras. När programmet avslutas försvinner variablerna och därmed också de data de innehåller. Men många typer av program behöver kunna spara data mellan de tillfällen då programmet körs. Som exempel kan bara nämnas ett vanligt ordbehandlingsprogram. De dokument som man producerar med hjälp av programmet ska förstås finnas kvar efter det att man avslutat programmet. Man kan då vid ett senare tillfälle starta programmet igen och fortsätta att arbeta med dokumenten.

För att lagra data permanent i en dator använder man *filer.* En fil sparas normalt på datorns hårddisk, men kan t.ex. också kopieras till ett usbminne eller skickas till en annan dator via nätet. Filer kan innehålla olika typer av data. En *textfil* är, som namnet antyder, en fil som innehåller vanlig text. Det är fråga om enkel text som inte innehåller någon information om olika typsnitt, färger etc. Vill man uttrycka det lite mer tekniskt kan man säga att en textfil innehåller en följd av tecken, där varje tecken representeras med hjälp av en teckenkod. En textfil kan redigeras med hjälp av ett vanligt texteditor-program, t.ex. programmet *Anteckningar (Notepad)* i Windows. I detta kapitel ska du få lära dig hur man läser och skriver textfiler i ett Python-program.

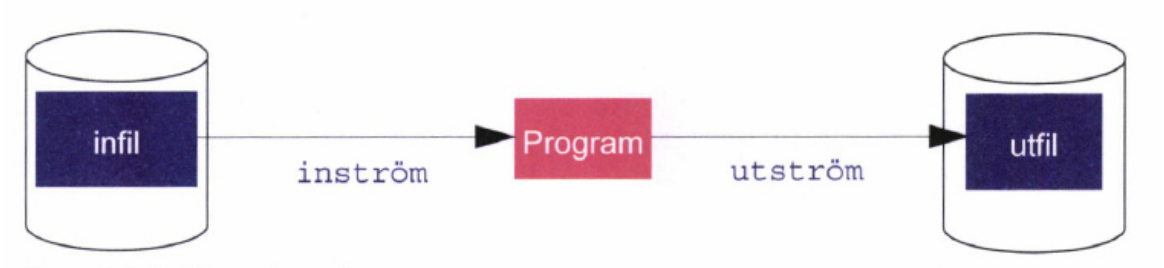
11.1 Strömmar och filer

I textfiler lagras texter som en följd av teckenkoder. Den enklaste formen av textfiler använder en byte (8 bitar) för varje tecken. Då kodas varje tecken enligt ASCII-standarden eller LATIN_1. (Se avsnitt 5.2.) Men lagringstekniken UTF-8 är också mycket vanlig. Där lagras lite mer ovanliga tecken i två, tre eller t.o.m fyra byte. En fil som är lagrad enligt UTF-8 kan alltså innehålla fler byte än antal tecken.

Python använder sig av s.k. *strömmar* för att läsa och skriva data. Man kan uppfatta en ström som en vattenslang. Man stoppar in data i ena änden av slangen och plockar ut data i den andra änden. Den ena änden av slangen kopplas till programmet och den andra kopplas till

kommandofönstret eller till en fil. Inströmmar är sådana strömmar i vilka data flödar in i programmet och utströmmar sådana strömmar i vilka data strömmar ut ur programmet. Detta illustreras i figur 11.1.

Figur 11.1 Filer och strömmar.

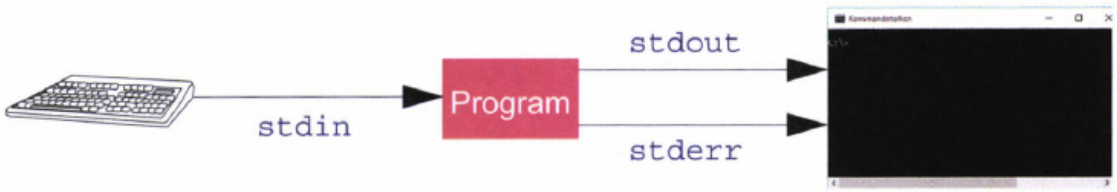


Figuren visar infil som gör en inström i ett program som gör en utström till en utfil.

I Python beskrivs strömmar med hjälp av s.k. *filobjekt (file objects).* Ett filobjekt innehåller all information som behövs för att styra en ström, t.ex. vilken fil den är kopplad till, aktuell position i strömmen och information om man uppnått *end of file* eller om något fel inträffat. Exakt hur ett filobjekt ser ut behöver man inte veta. Det enda man behöver göra är att hålla reda på referenserna till filobjekten och ge dessa referenser som argument till de standardfunktioner man anropar.

Det finns tre fördefinierade strömmar: *standard input, standard output* och *standard error.* Standard input hämtar data från tangentbordet och de två andra skriver ut data i kommandofönstret. För var och en av dessa strömmar finns en fördefinierad variabel. I standardmodulen `std` definieras de tre variablerna `std.stdin, std.stdout` och `std.stderr` vilka är referenser till filobjekten. Se figur 11.2.

Figur 11.2 De tre fördefinierade strömmarna.



Figuren visar ett program som hämtar data från tangentbordet, stdin, och som sedan skriver ut data som stdout eller stderr i kommandofönstret.

Funktionerna `input` och `print` använder som default standardströmmarna `stdin` och `stdout`. Den tredje strömmen, `stderr`, är lämplig att använda när man vill skriva felutskrifter. Det går nämligen att från operativsystemet koppla om strömmen `stdout` så att utskriften hamnar i en fil i stället för i kommandofönstret. Skriver man felutskrifter till `stdout` riskerar man därför att dessa utskrifter göms i en fil, så att

användaren inte ser dem. Här visas hur man kan skriva en felutskrift till `stderr`. Funktionen `print` har en parameter med namnet `file` som anger vilken ström den ska skriva till. Defaultvärdet för `file` är `std.stdout`, men det går att ändra till en annan ström:

```
import sys
print('Detta är en felutskrift', file=sys.stderr)
```

11.2 Öppna och stänga filer

Du ska nu få se att det går att skapa egna strömmar och koppla dem till filer och att man sedan kan läsa och skriva till dessa strömmar.

Funktionen `open` skapar ett filobjekt som beskriver den fil man vill läsa från eller skriva till. Som resultat ger `open` en referens till filobjektet. Som första exempel visas ett program som frågar efter namnet på en textfil, öppnar filen och skriver ut dess innehåll i kommandofönstret.

[fullständigt program]

```
namn = input('Filens namn? ')
f = open(namn, 'r')
for rad in f:
        print(rad, end='')
f.close()
```

Funktionen `open` har flera parametrar, men vi diskuterar här bara tre av dem. De ska alla vara texter. Den första parametern är filens namn. Den andra anger vad man vill göra med filen och vilken sorts fil det är. Man använder olika koder. Koden `r` står för "read" och betyder att man ska läsa från filen och att filen därför måste finnas tidigare. Om man vill läsa från en fil går det att utelämna den andra parametern, eftersom den har defaultvärdet `'r'`. En sammanställning av koderna visas i faktarutan. Man kan också ge parametern, `encoding`, för att ange hur den fil man ska läsa eller skriva är kodad. Om man inte ger denna, kommer defaultvärdet att bero på vilket system man kör på. Vet vi t.ex. att filen vi ska läsa är kodad enligt lagringstekniken UTF-8, kan vi skriva

```
f = open(namn, 'r', encoding='utf-8')
```

Andra tillåtna kodningar är t.ex. `'cp1252'`, som ofta används i Windows och `'iso8859_10'`, som också kallas LATIN_1. (Se avsnitt 5.2.)

Filnamnet behöver inte vara ett enkelt namn. Det kan innehålla en sökväg *(path),* t.ex. `c:\Pythonexempel\minfil.txt`. Om `open` hittar en fil med det angivna namnet kopplar den en ström till filen och skapar ett filobjekt som beskriver strömmen. Som resultat får vi en referens till filobjektet. Referensen sparas i variabeln `f`. Om funktionen `open` inte kan

öppna filen, om filen t.ex. inte finns, ger `open` en felsignal av typen `FileNotFoundError`. Om man inte fångar denna felsignal kommer programmet att ge en felutskrift och avslutas.

Referensen som man får som resultat från `open` kan man sedan i fortsättningen av programmet använda när man vill läsa ifrån eller skriva till filen. I programmet har vi använt en `for`-sats med operatorn `in` för att löpa igenom och läsa filen. På varje varv i `for`-satsen läser programmet in en rad från filen och placerar den inlästa raden i variabeln `rad`. I en textfil avslutas varje rad, utom eventuellt den sista, med en speciell radslutsmarkör som kan bestå av ett eller två tecken. När raderna läses in till programmet omvandlas varje sådan radslutsmarkör automatiskt till tecknet `'\n'` och detta läggs sist i den inlästa raden. Lägg märke till att den sista raden i filen kanske inte har avslutats med någon radslutsmarkör. I så fall finns inte tecknet `'\n'` i den sista inlästa raden. (Eftersom raderna själva innehåller tecknet `'\n'` har vi skrivit `end=' '` i anropet av `print` i programmet. Hade vi inte gjort det, hade det skrivits ut en extra tom rad vid varje anrop av `print`.)

Sist i programmet anropas funktionen `close`. Denna kopplar bort filobjektet `f` från den verkliga filen och frigör alla resurser som filobjektet har använt. När man anropat `close` är `f` inte längre kopplad till filen.

Vi ska nu ändra i vårt program. I stället för att skriva ut den lästa filens innehåll i kommandofönstret ska programmet kopiera filens innehåll till en ny fil. Dessutom ska det skriva ut ett meddelande som anger hur många rader och tecken som kopierats. Så här kan det t.ex. se ut när man kör programmet:

```
Infilens namn? minfil.txt
Utfilens namn? minkopia.txt
24 rader och 904 tecken kopierade
```

Så här ser programmet ut:

Öppna och stänga filer

`f = open(`*`filnamn`*`, `*`mode`*`, encoding=kodning)`

Parametrarna är texter, `open` ger som resultat en referens till ett filobjekt. Om `open` misslyckas genereras felsignalen `FileNotFoundError`.

`mode` anger hur filen ska användas:

Tabellbeskrivning

Tabellen har 9 rader och 2 kolumner.

`r`	ska läsas, filen måste finnas (default)
`w`	ska skrivas, om den finns skrivs den över, annars skapas en ny
`x`	ska skrivas, filen får inte finnas tidigare, en ny skapas
`a`	append, som `w` men lägger till från slutet om filen redan finns
`t`	anger att det är en textfil (default)
`b`	anger att det är en binärfil (en fil som inte innehåller text)
`+`	kan läggas till (t.ex. `r+`, `w+`), filen kan också skrivas resp, läsas
`f.`	stänger den fil `f` refererar till. Tömmer eventuella buffertar.

`close()`	
`f. closed`	ett värde av typen `bool` som anger om filen är stängd.

[fullständigt program]

```
n1 = input('Infilens namn? ')
n2 = input('Utfilens namn? ')
f1 = open(n1, 'r')
f2 = open(n2, 'w')
r, t = 0, 0 # antal rader och tecken
for rad in f1:
        t += f2.write(rad)
        r += 1>
print(f'{r} rader och {t} tecken kopierade')
f1.close()
f2.close()
```

I det anrop av `open` som markerats med rött ges koden `w`. Detta står förstås för "write". Om det inte finns någon fil med det angivna namnet, kommer `open` att skapa en ny fil, annars kommer den att skriva över en befintlig fil. Programmet läser sedan en rad i taget från infilen. Varje inläst rad skrivs ut i den fil `f2` refererar till. Utskriften sker med hjälp av en standardfunktion med namnet `write`. Denna fungerar ungefär som funktionen `print` som du känner till sedan tidigare, fast framför funktionens namn anger man till vilken ström utskriften ska ske. Här har vi angivit att utskriften ska ske till strömmen `f2`. Funktionen `write` ger som resultat antalet tecken den skrivit. För varje utskriven rad ökas radräknaren

`r` med 1. Variabeln `t` som innehåller antalet kopierade tecken ökas med den skrivna radens längd. Eftersom det kan finnas radslutstecken sist i de

inlästa raderna, kommer dessa också att räknas med.

Sist i programmet anropas funktionen `close` som stänger filerna. Anropet `f2.close()` garanterar att allt man skrivit till strömmen `f2` verkligen skrivs ut i filen och inte ligger kvar i någon intern buffert.

Det finns ett litet problem med ovanstående program. Om det skulle gå att öppna `f1` med inte `f2`, kommer en felsignal att genereras vid det andra anropet av funktionen `open`. Då kommer exekveringen att avbrytas där, vilket innebär att filen `f1` inte stängs på rätt sätt. I Python finns en speciell konstruktion som gör att man enkelt kan hantera denna typ av problem. Man använder en `with`-sats. Så här ser det ut:

[fullständigt program]

```
# Demo av with-sats
n1 = input('Infilens namn? ')
n2 = input('Utfilens namn? ')
with open(n1, 'r') as f1, open(n2, 'w') as f2:
        r, t = 0, 0 # antal rader och tecken
        for rad in f1:
                t+=f2.write(rad)
                r+=1
        print(f'{r) rader och {t} tecken kopierade')
```

Den röda raden inleder `with`-satsen. Där skriver man de anrop av funktionen `open` som ska finnas i programavsnittet. Man anger också vad variablerna som refererar till filobjekten ska heta. Sedan skriver man, indraget, de satser som ska exekveras inne i `with`-satsen. Om inga fel inträffar, exekveras dessa satser normalt och alla öppnade filer stängs automatiskt. Om det skulle uppstå något fel vid anropen av `open` eller i de indragna satserna, garanterar `with`-satsen att de filer som eventuellt har öppnats kommer att stängas. Fördelen med `with`-satsen är att den är enklare att använda än om man skulle försöka skapa motsvarande effekt med `try`-satser. Man behöver inte heller anropa `close` explicit.

with-sats

```
with open(n1, mode) as f1, open(n2, mode) as f2,
... :
    satser
```

Garanterar att alla filer som öppnats också stängs innan satsen avslutas.

Uppgift 11.1

Skriv ett program som läser in ett antal namn från tangentbordet och sparar de inlästa namnen i en textfil med namnet `personer.txt`. I textfilen ska det stå ett namn på varje rad. Om det redan finns en fil med det namnet ska filen inte förstöras, utan de nya namnen ska läggas till sist i filen. Kör programmet två gånger och lägg varje gång till några personer. Innan du kör programmet den första gången ska du se till att det inte finnas någon fil med namnet `personer.txt`. Efter varje körning kan du använda en text-editor för att se vad som finns i filen `personer.txt`.

Uppgift 11.2

Skriv ett program som läser in en textfil som innehåller ett Python-program. Programmet ska fråga efter filens namn. Programmet ska

undersöka vilka rader i filen som innehåller kommentarer. För enkelhets skull får du förutsätta att det bara finns kommentarer av typen `#`. Programmet ska skriva ut hur många procent av raderna i filen som innehåller kommentarer.

11.3 Läsa och skriva textströmmar

Du har redan sett hur man kan läsa från en ström med hjälp av operatorn `in` och hur man kan skriva till en ström med funktionen `write`. I faktarutan visas några fler operationer för textströmmar.

I de exempel som visats hittills har vi bara läst och skrivit text utan att ändra den. Vi ska nu visa ett exempel där man behöver tolka de texter man läser in. Anta att en textfil innehåller en lista med elever och att det för varje elev finns uppgifter om hur många poäng eleven hade på ett visst prov. För varje elev finns det en rad i filen. Först på raden står namnet och sedan följer poängen. Filen kan t.ex. innehålla raderna:

```
Linda Olsson 52
Mikael Bergman 45
Karin Jansson Fernandez 69
Daniel Lindman 58
```

Anta nu att det krävs minst 50 poäng för att bli godkänd på provet. Följande program läser filen och skapar en ny fil i vilken den lägger in alla godkända elever och deras poäng. Programmet börjar med att fråga efter de två filernas namn.

Operationer på strömmar med text

Här är `f` är en referens till ett filobjekt som beskriver en ström med text.

Tabellbeskrivning

Tabellen har 8 rader och 2 kolumner.

`f. read (n)`	Läser `n` (default 1) st tecken. Ger `''` vid filslut Om `n` är negativ eller `None` läses hela strömmen
`f. readline ()`	Läser en rad. Ger `''` vid filslut Radslutstecken `'\n'` finns ev. med sist på raden
`for r in f:`	Läser en rad i taget till `r`. Som upprepade `readline`
`f. readlines ()`	Läser alla raderna. Ger en lista med rader
`f. write (s)`	Skriver ut texten `s` till `f`. Ger antal skrivna tecken
`print (..., file=f)`	Som vanliga `print`, men skriver till `f`
`f. encoding`	En text som anger hur strömmen är kodad
`f. name`	En text som anger filens namn

Om man kör programmet med filen ovan som indata, kommer den nya fil som skapas att få innehållet:

```
Linda Olsson 52
Karin Jansson Fernandez 69
Daniel Lindman 58
```

Så här ser programmet ut:

[fullständigt program]

```
# Godkända elever, version 1
n1 = input('Filen med resultat? ')
n2 = input('Filen med godkända? ')
with open(n1, 'r') as f1, open(n2, 'w') as f2:
        for s in f1:
                s = s.strip(' \n') # tar bort blanka
och \n
                i = s.rfind(' ') # index för sista
blanka
                poäng = int(s[i+1:]) # gör om betyg
till en int
                if poäng >= 50:
                        f2.write(s + '\n')
```

För enkelhets skull använder vi en `with`-sats för att öppna de två filerna. På varje varv läser programmet en rad från infilen. Vi börjar med att anropa funktionen `strip` med parametern `' \n'`. Denna ger en ny text där den tagit bort sådana radslutstecken och blanka tecken

som eventuellt finns och först och sist i `s`. sist. Vi letar sedan bakifrån efter det sista blanka tecknet på raden. Det gör vi med funktionen `rfind` som ger oss indexet, `i`, för detta tecken. Då vet vi att siffrorna börjar i position `i+1`. Sedan skär vi ut en skiva som innehåller siffrorna och gör om denna skiva till en `int`. Då kan vi på nästa rad testa om resultatet är 50 eller större. Om så är fallet, skriver vi ut hela raden till den nya filen `f2`. Lägg märke till att funktionen `write` (till skillnad från `print`) inte själv lägger till några radslutstecken. Vi måste därför lägga till ett sådant i slutet på varje rad. Annars skulle allt hamna på samma rad i den nya filen. Som ett alternativ kan vi på den sista raden i stället använda funktionen `print`. Vi ger då parametern `file` värdet `f2`:

```
print(s, file=f2)
```

Vi ska nu göra en utökning till av programmet. Anta att filen med resultat inte innehåller en totalpoäng per elev, utan poängen för flera delprov. Det kan t.ex. se ut på följande sätt:

```
Linda Olsson 12 20 15 5
Mikael Bergman 15 30
Karin Jansson Fernandez 19 30 10 10
Daniel Lindman 10 28 20
```

När man har indata som ser ut på detta sätt med flera delar, s.k. "ord" avgränsade med vita tecken (mellanslag, tabulatorer och radslutstecken), kan man använda funktionen `split` för att bilda en lista där varje ord på raden hamnar i ett eget element i listan:

[fullständigt program]

```
# Godkända elever, version 2
n1 = input('Filen med resultat? ')
n2 = input('Filen med godkända? ')
with open(n1, 'r') as f1, open(n2, 'w') as f2:
```

```
        for rad in f1:
                ord = rad.split()
                namnen = [e for e in ord if not
e.isdecimal()]
                poäng = [int(e) for e in ord if
e.isdecimal()]
                namn = ' '.join(namnen)
                summan = sum(poäng)
                if summan >= 50:
                        print(namn, summan, file=f2)
```

Liksom tidigare läser vi en hel rad i taget och bildar en lista, `ord`, med alla orden på raden. Sedan använder vi tekniken med *list comprehension*

---

för att bilda två nya listor. Listan `namnen` består av alla element från listan `ord` som inte innehåller ett decimalt heltal. I listan `poäng` innehåller varje element poängen på ett delprov. Lägg märke till att när vi skapar listan `poäng` passar vi också på att omvandla texterna i orden till heltal genom att anropa `int`. Vi slår sedan ihop alla elementen i listan `namnen` till en enda text med hjälp av funktionen `join` och beräknar summan av poängen i listan `poäng` med funktionen sum.

Om poängen kan innehålla decimaler, fungerar det inte att anropa funktionen `isdecimal`. Vi kan då i stället använda vår egen funktion `isfloat` som definierades på sidan 193 och skriva så här:

```
namnen = [e for e in ord if not isfloat(e)]
poäng = [float(e) for e in ord if isfloat(e)]
```

Uppgift 11.3

I en datasal håller man reda på hur lång tid eleverna använder datorerna genom att skapa en loggfil som innehåller uppgifter om hur lång tid varje användare varit inloggad. Loggfilen, som är en textfil, innehåller en rad för varje användare. På varje rad finns först användarens inloggningsnamn. Därefter finns ett godtyckligt antal heltal. Varje heltal motsvarar ett inloggningstillfälle och anger hur många minuter användaren har varit inloggad vid det tillfället. Om filen t.ex. innehåller raden:

```
kallep 25 40 15
```

betyder det att användaren `kallep` varit inloggad vi tre olika tillfällen och att han sammanlagt varit inloggad 80 minuter. Din uppgift är nu att skriva ett program som läser loggfilen och som undersöker vilken användare som varit inloggad sammanlagt längst tid. Programmet ska skriva ut användarnamnet för denna användare och hur länge han eller hon varit inloggad.

11.4 Ändra i en fil

Ibland behöver man göra förändringar inne i en textfil. Man kanske vill ändra texten på någon rad, lägga till nya rader eller ta bort rader. Då blir det genast lite mer komplicerat. Det är visserligen möjligt att ändra enstaka tecken direkt i en fil, men det är inte så enkelt eftersom den nya texten inte alltid är lika lång som den gamla. Det enklaste är att läsa in hela filen till en lista med rader, ändra i listan och sedan skriva

ut listan i filen. Hur man kan göra detta ska vi visa i det första delavsnittet. Men om filen är stor kanske detta inte är möjligt. Man kan då använda en temporär fil. Hur det går till visas i avsnitt 11.4.2.

Vi ska visa två versioner av ett program som ändrar i en fil med elevers namn och poäng. Filen kan se ut som på sidan 205. Programmet kan lägga till nya elever i filen eller ändra poängen för en elev som redan finns i filen. Det kan t.ex. se ut så här när man kör programmet:

```
Avsluta med Ctrl-c
Filen med resultat? resultat.txt
>Mikael Bergman 20 30
>Jenny Nilsson 10 13 15
>Här skriver användaren Ctrl-c
```

I båda versionerna av programmet utnyttjas en hjälpfunktion som heter `namnet`. Den används när man ska jämföra de namn som finns i filen med de namn användaren skriver in. Funktionen `namnet` plockar ut en elevs namn från en rad. Den använder samma teknik som användes i programmet på sidan 205. Den skapar en lista med alla ord som inte innehåller heltal och sedan slår den ihop elementen i listan till en enda text. För att det inte ska spela någon roll om man råkar använda stora eller små bokstäver när man skriver namnen, översätts alla stora bokstäver till små. Så här ser funktionen `namnet` ut:

[fullständigt program]

```
def namnet(rad):
        ord = rad.split()
        namnen = [e for e in ord if not e.isdecimal()]
        namn = ' '.join(namnen)
        return namn.lower()
```

11.4.1 Använda en lista

Den första versionen av programmet innehåller tre steg. I det första steget frågar det efter filens namn och läser in alla raderna i filen till en lista. I det andra steget ber det användaren skriva in en rad i taget. Dessa rader ska se ut på samma sätt som raderna i filen. De ska alltså innehålla en elevs namn plus poängen på de olika delkurserna. Om den rad användaren skriver har samma namn som någon av raderna i filen, ändras elementet i listan så att det innehåller den nya raden. Om det namn som användaren skriver inte finns i listan, läggs en ny rad till sist i listan. I det tredje och sista steget skrivs hela listan ut till filen. Hela

---

programmet kapslas in i en `try`-sats som fångar felsignalen `KeyboardInterrupt`, vilken genereras när användaren skriver Ctrl-c.

[fullständigt program]

```
try:
        # Steg 1
        print('Avsluta med Ctrl-c')
        f_namn = input('Filen med resultat? ')
        with open(f_namn, 'r') as f:
                rader = f.readlines() # läs in hela
filen
        # Steg 2
        while True:
                ny_rad = input('> ') + '\n'
                nytt_namn = namnet(ny_rad)
                if nytt_namn == '':
                        print('Felaktig rad')
                        continue # hoppa över
                inlagd = False
                for i in range(0, len(rader)):
                        namn_i = namnet(rader[i])
```

```
                        if nytt_namn == namn_i:
                                rader[i] = ny_rad #
ändra denna rad
                                inlagd = True
                                break
                if not inlagd:
                        rader.append(ny_rad) # namnet
fanns inte
except KeyboardInterrupt:
        Steg 3
        with open(f_namn, 'w') as f:
                for rad in rader:
                        f.write(rad)
```

I steg 1 används funktionen `readlines` för att läsa in hela filen på en gång. Varje rad i filen blir ett element i listan `rader`. I steg 2 finns en `while`-sats som löper ett varv för varje ny rad användaren skriver in. Vi lägger till tecknet `'\n'` sist på den inlästa raden så att den får samma slut som raderna i listan. Funktionen `namnet` ger en tom text som resultat om inte den inlästa raden innehåller ett namn. Detta kontrollerar vi och hoppar över den inmatade raden om den inte är korrekt. (Satsen `continue` hoppar över resten av det aktuella varvet i `while`-satsen.)

När vi läst in en rad som användaren skrivit, löper vi genom alla elementen i listan `rader`. Om något av elementen innehåller samma

namn som den inmatade raden ersätts detta element med den inmatade raden. Variabeln `inlagd` används för komma ihåg att detta skett. När den inre `for`-satsen är slut undersöks om den nya raden redan är inlagd. Om den inte är det, lägger vi till den sist i listan.

Det tredje steget i programmet utförs när användaren skrivit Ctrl-c. Då skrivs hela listan ut till filen.

Uppgift 11.4

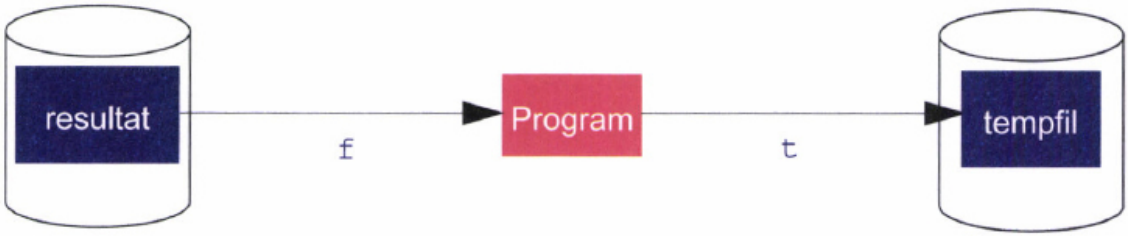
Anta att man ska kunna ta bort elever från resultatfilen. Det ska man kunna göra genom att bara skriva elevens namn, utan några poäng efter. Föreslå ändringar som man kan göra i programmet för att åstadkomma detta.

11.4.2 Använda en temporär fil

[överkurs]

Att läsa in allting till en lista kanske inte fungerar om man har en stor fil. Då kan man använda sig av en temporär fil i stället. För varje ny person användaren skriver in låter man programmet utföra två steg. I det första steget läser man data från filen, en person i taget, och kopierar raderna till en temporär fil. Om någon av raderna gäller för den person som användaren skrev in, ska raden i filen ersättas med den rad användaren skrev. Om den person användaren skrev in inte fanns i filen lägger man till en ny rad sist. Programmet använder två strömmar: `f` som är kopplad till filen med resultat och `t` som är kopplad till den temporära filen. Detta illustreras i figur 11.3.

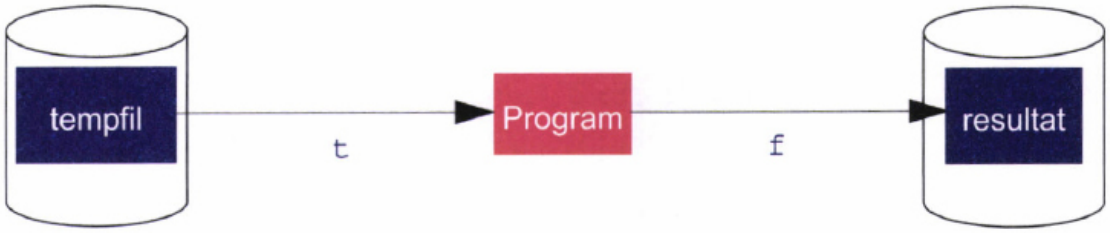
Figur 11.3 Steg 1: kopiering till en temporär fil.



Bilden visar hur resultatet via strömmen f kopplas till programmet som via strömmen t kopplar till tempfilen.

När detta första steg är klart finns alltså alla personerna i den temporära filen och dessutom är eventuellt en ny person inlagd sist. I nästa steg, steg 2, kopierar man helt enkelt innehållet i den temporära filen tillbaka till den ursprungliga resultatfilen. Filens gamla innehåll förstörs då och ersätts med det nya. Detta visas i figur 11.4. Så här ser programmet ut. (Fler kommentarer följer efter texten.)

Figur 11.4 Steg 2: kopiering tillbaka från den temporära filen.



Bilden visar hur tempfilen via strömmen t kopplas till programmet som via strömmen f kopplar till resultatet.

[fullständigt program]

```
import tempfile
```

```
try:
        print('Avsluta med Ctrl-c')
        f_namn = input('Filen med resultat? ')
        with tempfile.TemporaryFile('w+t') as t:
                while True:
                        # Steg 1
                        with open(f_namn, 'r') as f:
                                ny_rad = input('> ') +
'\n'
                                nytt_namn = namnet
(ny_rad)
                                if nytt_namn == ' :

print('Felaktig rad')
                                        continue
                                inlagd = False
                                for rad in f:
                                        if namnet(rad)
== nytt_namn:

t.write(ny_rad)
                                                inlagd
= True
                                        else;

t.write(rad)
                                if not inlagd:

t.write(ny_rad)
                        # Steg 2
                        with open(f_namn, 'w') as f:
                                t.seek(0)
                                for rad in t:
                                        f.write(rad)
                                t.seek(0)
                                t.truncate(0)
except KeyboardInterrupt:
        exit()
```

Programtexten kapslas in i en `try`-sats som fångar in felsignaler av typen `KeyboardInterrupt`. Syftet är att den som kör programmet ska

kunna avsluta inmatningen genom att skriva Ctrl-c utan att få en massa felutskrifter. Programmet börjar med att fråga efter namnet på resultatfilen. Därefter skapas den temporära filen. Detta görs med hjälp av funktionen `TemporaryFile` som finns i modulen `tempfile`. Denna funktion skapar en fil med ett unikt namn så att man inte av misstag förstör någon befintlig fil. Den fil man får har också egenskapen att den kommer att tas bort från filsystemet när den stängs. Vi lägger anropet av `TemporaryFile` i en `with`-sats så att den stängs automatiskt. Parametern `'w+t'` betyder att den temporära filen ska vara en textfil som i första hand ska skrivas, men också kan läsas.

Programmet innehåller en yttre `while`-sats som löper ett varv för varje ny elev man vill lägga till eller uppdatera. För varje ny elev läses en rad från tangentbordet. Därefter plockar man ut elevens namn från den inlästa raden med hjälp av funktionen `namnet` som vi beskrev i förra avsnittet. Om man har skrivit en felaktig rad som inte innehåller något namn, får man en felutskrift och programmet använder en `continue`-sats för att hoppa till ett nytt varv så att man kan skriva in en ny rad.

I den första `for`-satsen läses en rad i taget från resultatfilen. Namnet för eleven på raden plockas ut på samma sätt som namnet för den nya eleven. Därefter jämförs de två namnen. Om namnen är lika, skriver man den nya raden till den temporära filen. Annars kopierar man raden från resultatfilen. När man gått igenom alla raderna i resultatfilen kontrollerar man om den nya raden lagts in. Om inte detta har skett, skrivs den ut sist i den temporära filen.

Därefter avslutas den första `with`-satsen, vilket innebär att filen `f`, resultatfilen, automatiskt stängs. Men den temporära filen `t` är fortfarande öppen eftersom den öppnades i den yttre `with`-satsen.

I det andra steget startar en ny `with`-sats. I denna öppnar vi resultatfilen igen, men denna gång för utskrift. Det tidigare innehållet i resultatfilen kommer då att skrivas över. Nu ska vi läsa från den temporära filen. Då måste vi först backa tillbaka den till början. Detta görs med funktionen `seek`. Därefter läser vi den temporära filen rad för rad och kopierar den till

resultatfilen. När detta är klart avslutar vi med att åter backa tillbaka den temporära filen till början. Sedan tömmer vi den genom att anropa funktionen `truncate`. Då är den redo för nästa varv.

Fler funktioner för att hantera filer

Tabellbeskrivning

Tabellen har 3 rader och 2 kolumner.

`f. seek (0)`	Backar tillbaka aktuell position i strömmen till början
`f. truncate (0)`	Raderar innehållet i en fil
`tempfile.TemporaryFile ('w+t')`	Skapar en temporär fil

11.5 Parametrar till main-modulen

När man ska köra ett Python-program i *script mode* kan man starta Python-interpretatorn genom att skriva kommandot `py` eller `python3` i kommandofönstret. Om man t.ex. vill köra ett program som finns i en fil med namnet `arg_demo.py` ger man kommandot:

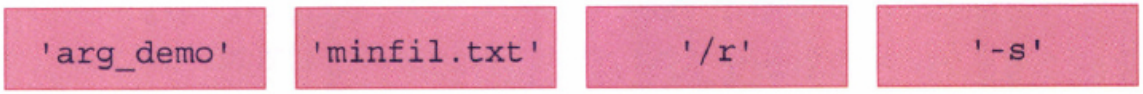
```
py arg_demo.py
```

(Eventuellt ska man skriva `python3` i stället för `py`). När man skriver kommandot `py` kan man emellertid lägga till argument på slutet. Man säger då att man ger *argument på kommandoraden.* Det kan t.ex. se ut på följande sätt:

```
py arg_demo.py minfil.txt /r -s
```

Här har man gett de tre argumenten `minFil . txt, /r` och `-s`. Man skriver ett eller flera blanka tecken mellan varje argument. Man kan skriva citationstecken (inte apostrofer) runt ett argument om det innehåller blanka tecken. När Python-interpretatorn startar läser den argumenten som finns på kommandoraden. Den skapar sedan en lista där varje argument blir ett element. Lägg märke till att namnet på programfilen då har lagts i det första elementet i listan. När programmet sedan startar i `main`-modulen kan det hämta listan med argument. Listan heter `argv` och ligger i modulen `sys`. Om vi antar att man skrev kommandoraden ovan, kommer listan `sys.argv` att se ut som i figur 11.5.

Figur 11.5 Listan `sys.argv`.



Figuren visar listan ’arg_demo’, ’minfil.txt’, ’/r’, och ’-s’.

I `main`-modulen kan `sys.argv` användas som en vanlig lista. Här kommer nu ett litet demonstrationsprogram. Det enda programmet gör är att skriva ut sitt namn och de argument som gavs på kommandoraden.

[fullständigt program]

```
from sys import argv
print('Programnamn:', argv[0])
for i in ranged, len(argv)) :
        print('Argument', i, ':', argv[i])
```

Om man startar programmet med kommandoraden:

```
py arg_demo.py minfil.txt /r -s
```

ger det utskriften:

```
Programnamn: arg_demo.py
Argument 1 : minfil.txt
Argument 2 : /r
Argument 3 : -s
```

Man använder ofta argument på kommandoraden för att ange namn på filer som programmet ska använda. Här kommer som exempel en ny version av programmet från sidan 202 som kopierade en fil till en annan och räknade det antal rader och tecken som kopierades.

[fullständigt program]

```
# Kopiera en fil till en annan
from sys import argv
if len(argv) != 3:
        print('Fel antal argument')
        exit()
with open(argv[1], 'r') as f1, open(argv[2], 'w') as
f2:
        r, t = 0, 0 # antal rader och tecken
        for rad in f1:
                t+ = f2.write(rad)
                r+ = 1
        print(f'{r} rader och {t} tecken kopierade')
```

Programmet ska ha två filnamn som argument. Om programmet ligger i filen `kopiera.py` kan man t.ex. starta det med kommandot:

```
py kopiera.py data.txt kopia.txt
```

Utskriften kan t.ex. bli:

```
24 rader och 645 tecken kopierade
```

Programmet kontrollerar att listan `argv` innehåller tre element. Programnamnet finns i element nummer 0, det första filnamnet i element nummer 1 och det andra i element nummer 2. Om något filnamn är felaktigt genereras en felsignal när `open` anropas. Man får då en felutskrift, och programmet stoppas.

Uppgift 11.5

Skriv ett program som räknar hur många rader det finns i en fil. Programmet ska startas från kommandoraden. Man ska kunna ge flera filnamn som argument. Programmet ska då beräkna antalet rader i alla filerna. Detta ska göras för en fil i taget och man ska få en utskriftsrad för varje fil.

11.6 Sammanfattning

Efter att ha läst detta kapitel bör du:

- veta vad strömmar är,
- veta hur man öppnar en fil och kopplar den till en ström och hur man stänger en fil,
- förstå hur `with`-satsen fungerar,
- veta hur man läser och skriver till strömmar,
- känna till hur man kan ge argument till `main`-modulen från kommandoraden.

11.7 Övningar

11.1 En textfil innehåller temperaturer som uppmätts kl 13 på en viss plats under en månad. Det finns en temperatur på varje rad i filen. Skriv ett program som läser filen och som skriver ut den högsta uppmätta temperaturen samt temperaturernas medelvärde. Användaren ska skriva in filnamnet när programmet körs.

11.2 Skriv ett program som läser en textfil och som skriver ut textfilens innehåll i kommandofönstret. Vid utskriften ska alla tabulatortecken ersättas med tre mellanslag.

11.3 Gör samma sak som i föregående uppgift, men utforma programmet så att textfilen ändras i stället för att den ändrade texten skrivs ut i kommandofönstret. *Tips:* Använd en lista.

11.4 Man har i en textfil samlat uppgifter om ett antal personer. För varje person finns två rader i filen. På första raden står personens namn och adress och på andra raden finns personens ålder, längd och vikt. Längden anges i cm och vikten i kg. Man vill göra en medicinsk studie av överviktiga personer och söker därför personer vilkas s.k. *body mass index* (BMI) överstiger värdet 30. BMI beräknas enligt formeln $^{m}/_{h}{}^{2}$ där *m* är vikten i kg och *h* längden i meter. Skriv ett program som läser filen med personuppgifter. Programmet ska skapa en ny textfil som bara innehåller uppgifterna för de personer vilkas BMI överstiger 30. Användaren ska skriva in filnamnen när programmet körs.

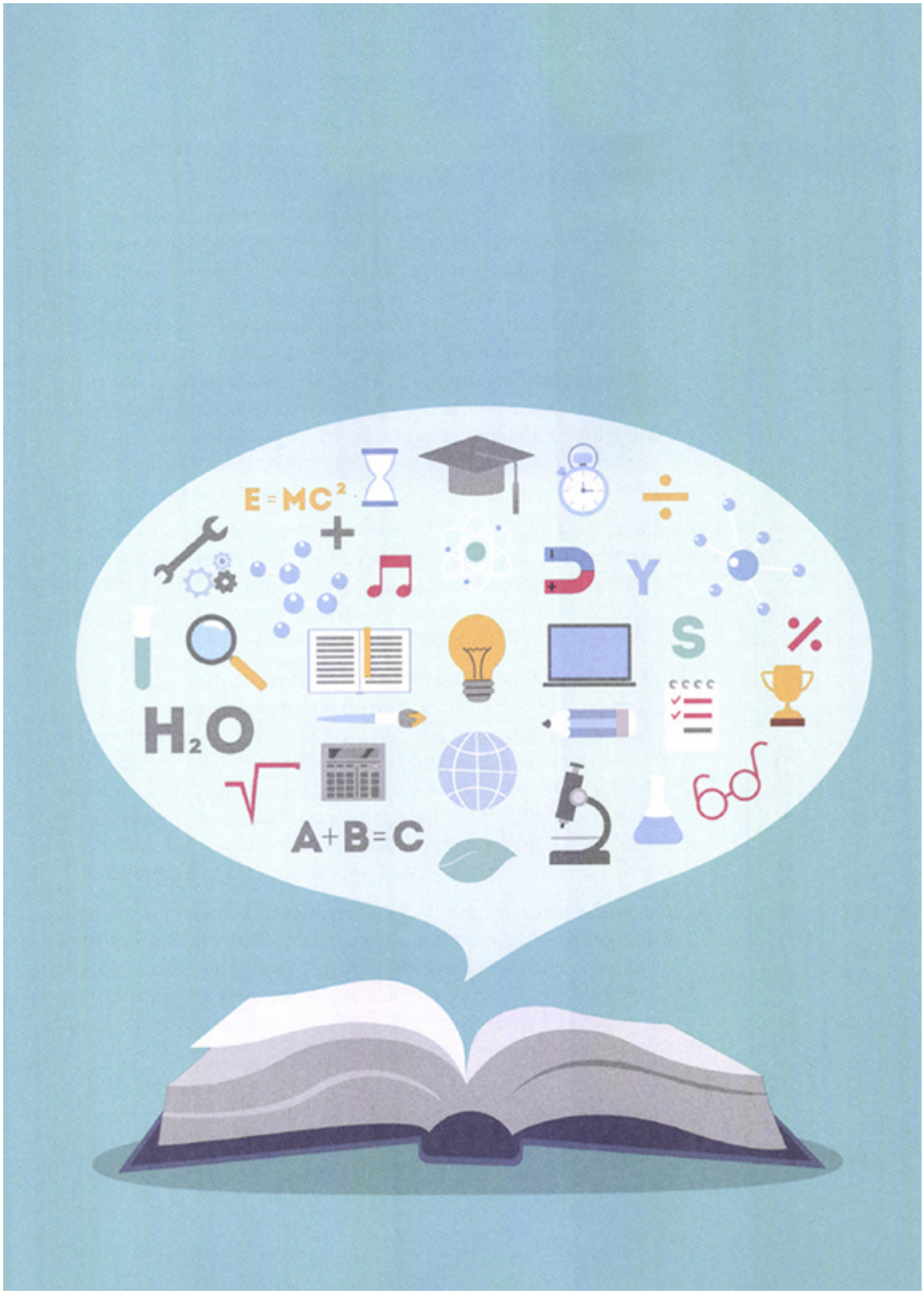
11.5 Skriv ett program som läser en textfil och söker efter de rader i filen som innehåller en viss text. De funna raderna ska kopieras till en ny fil. Namnen på den befintliga filen och på den nya filen samt den text man söker ska ges som argument på kommandoraden när programmet startas.

11.6 Anta att man har en textfil som innehåller en lista med personer. (Det finns en rad för varje person, och den innehåller personens namn.) Uppgiften är att skriva ett program som slumpmässigt väljer ut en av personerna och skriver ut personens namn. *Tips:* Läs först igenom filen en gång och räkna det totala antalet rader. Slumpa därefter med hjälp av standardfunktionen `randint` (se sidan 39)

fram ett radnummer *k* i det beräknade intervallet. Läs sedan filen igen från början, fram till och med rad nummer *k.*

11.7 Skriv ett program som läser en textfil som innehåller rövarspråk. Programmet ska översätta rövarspråket tillbaka till vanligt språk och skriva ut det i kommandofönstret. Jämför med övning 5.4 på sidan 90. Du får för enkelhets skull förutsätta att texten i filen innehåller korrekt rövarspråk, dvs. att alla konsonanter är fördubblade, med ett 'o' emellan.

12 Mängder och avbildningar


E=MC²
H₂O
A+B=C

En viktig grupp av standardtyper är s.k. *containertyper* (eller *containerklasser* som de kallas om man programmerar i ett objektorienterat språk). Speciellt för en variabel av en sådan typ är att den, inne i sig, innehåller referenser till andra dataobjekt. Exempel på containertyper är typerna `list` och `tuple` vilka vi redan behandlat. Dessa används för att bilda sekvenser av dataobjekt. I detta kapitel ska vi beskriva två andra slag av contanertyper, nämligen typer som används för att skapa mängder och avbildningar.

12.1 Typerna `set` och `frozenset`

Typerna `set` och `frozenset` används för att bilda motsvarigheten till matematikens mängder. Ett visst element kan antingen ingå i en mängd eller inte göra det. Till skillnad från en lista kan varje element som tillhör en mängd kan bara finnas en enda gång i mängden. Elementen ligger inte heller i någon speciell ordning. Därför kan man inte indexera i en mängd eller skära skivor.

Man kan skapa en mängd genom att räkna upp elementen omgivna av tecknen { och }:

```
färger = {'blå', 'gul'}
lycko_nr = {7, 16, 3}
```

Ett annat sätt är använda funktionen `set`:

```
mix = set(['hallon', 'blåbär'])
```

Argumentet till `set` ska vara en sekvens eller en annan mängd. Här har vi gett en lista som argument, men det går också med texter och tupler:

```
m1 = set('abcdef')
t1 = (1, 2, 3)
m2 = set(t1) # m2 blir: {1, 2, 3}
```

Vill man skapa en tom mängd, måste man använda funktionen `set`:

```
m3 = set() # m3 blir: en tom mängd {}
```

Mängder av typen `frozenset` skapas med hjälp av funktionen `frozenset`:

```
binära_siffror = frozenset((0,1))
vokaler = frozenset('aouåeiyäö')
```

Skillnaden mellan typerna `set` och `frozenset` är att mängder av typen `frozenset` inte kan ändras.

Ett alternativt sätt att bilda mängder är att använda s.k. *set comprehension.* Det fungerar på samma sätt som *list comprehension,* fast resultatet blir en mängd i stället för en lista. Här är ett par exempel:

```
ktal = {t*t for t in lycko_nr} # blir: {256, 9,
49}
udda_tal = {t for t in range(0, 10) if t % 2! = 0}
```

Man kan lägga till element till en mängd med funktionen `add`:

```
färger.add('röd') # färger blir: {'blå', 'gul',
'röd'}
m2.add(7) # m2 blir: {1, 2, 3, 7}
```

Antalet element i en mängd kan avläsas med funktionen `len`:

```
len(m2) # blir: 4
```

Vill man ta bort ett element från en mängd kan man använda någon av funktionerna `remove` och `discard`:

```
färger.remove('blå') # färger blir: {'gul', 'röd'}
m2.discard(1) # m2 blir: {2, 3, 7}
```

Skillnaden mellan funktionerna `remove` och `discard` visar sig om man försöker ta bort ett element som inte finns i mängden. Då fungerar de på olika sätt:

```
m2.discard(9) # inget händer
m2.remove(9) # ger felsignal
```

```
KeyError: 9
```

Det går att ta bort ett slumpmässigt valt element med funktionen `pop`:

```
x = m2.pop() # x blir: något av elementen
```

Man kan ta bort alla elementen från en mängd med funktionen `clear`:

```
m2.clear() # m2 blir en tom mängd
```

Funktionerna `add`, `remove`, `discard`, `pop` och `clear` kan inte användas för mängder av typen `frozenset`.

Man kan skriva ut en mängd direkt med funktionen `print`:

```
print(lycko_nr)
print(mix)
```

Detta ger utskriften

```
{16, 3, 7}
```

```
{'blåbär', 'hallon'}
```

Inläsning till en mängd kan göras genom att man först läser in till en lista (se avsnitt 6.3 på sidan 107) och sedan skapar en mängd utgående från den inlästa listan. Vi kan t.ex. skriva:

```
s = input('Ingredienser? ')
ordlist = s.split() # en lista med orden
ordmängd = set(ordlist) # en mängd med orden
```

När man kör dessa rader kan det se ut så här

```
Ingredienser? mjöl ägg mjölk smör salt
```

och `ordmängd` blir `{'smör', 'mjölk', 'mjöl', 'ägg', 'salt'}`.

Om man vill läsa in numeriska värden till en mängd, kan man använda *set comprehension* för att omvandla de inlästa texterna till tal:

```
s = input('Turnummer? ')
tallist = s.split()
talmängd = {int(e) for e in tallist}
```

Operatorn `in` kan precis som för listor användas för att undersöka om ett visst element ingår i en mängd:

```
'gul' in färger # ger värdet True
```

```
'h' in vokaler # ger värdet False
```

Operatorn `in` kan också användas när man vill löpa igenom alla elementen i en mängd:

```
for f in färger:
        print(f)
```

Utskriften blir

```
gul
röd
```

Precis som när det gällde listor är de variabler man använder för att beskriva mängder i själva verket referenser till själva mängderna. (Se avsnitt 6.5 på sidan 113.) Detta betyder att om vi gör en tilldelning så kopieras inte själva mängderna, utan bara referenserna.

```
a = {2, 5}
b = a # b blir: {2, 5}
b.add(8) # a och b blir: {8, 2, 5}
```

Här ändras alltså också mängden `a` när vi lägger till ett element i `b`. Vill man ha en kopia av själva mängden kan man liksom för listor använda funktionen `copy`:

```
a = {2,5}
b = a.copy() # b blir: {2, 5}
b.add(8) # b blir: {8, 2, 5}, a ändras inte
```

Man kan jämföra mängder med varandra. Den enklaste operatorn är `==`. Två mängder betraktas som lika om de innehåller samma element.

```
x = {5,2}
x == a # ger värdet True
y = {5,2,7}
y == a # ger värdet False
```

Operatorerna `<` och `>` kan användas för att undersöka om en mängd är en äkta delmängd av en annan.

```
x < y # ger värdet True
x < a # ger värdet False
```

Här är det första uttrycket sant eftersom `x` är en delmängd av `y`, men `x` inte lika med `y`. I det andra uttrycket är visserligen `x` en delmängd av `a`, men inte en äkta sådan, eftersom mängderna är lika. Däremot är följande uttryck sant

```
x <= a # ger värdet True
```

Det finns också ett antal operatorer och funktioner som utför vanliga operationer på mängder, t.ex. För att bilda unionsmängden `u` och snittmängden `v` av mängderna `x` och `y` kan man använda operatorerna `|` och `&`. Vi kan t.ex. skriva

```
u = x|y # u blir: {2, 5, 7}
v = x & y # v blir: {2, 5}
```

De flesta av operationerna kan göras antingen med hjälp av en funktion eller med en operator. Vi hade alternativt kunnat skriva

```
u = set.union(x, y)
v = set.intersection(x, y)
```

I faktarutan ges en sammanställning av de operationer man kan utföra på mängder.

Operationer för typerna `set` och `frozenset`

Här betecknar `m`, `m1` och `m2` mängder och `e` ett element.

Operationer som ändrar mängder är inte tillåtna för `frozenset`.

Tabellbeskrivning

Tabellen har 24 rader och 2 kolumner.

`set(s)`	funktion som skapar en mängd från sekvensen `s`
`frozenset(s)`	funktion som skapar en mängd från sekvensen `s`
`len(m)`	ger antalet element i `m`
`m.add(e)`	lägger till elementet `e` i `m`
`m.remove(e)`	tar bort elementet `e` från `m` (kan ge felsignal)
`m.discard(e)`	tar bort elementet `e` från `m` om det finns
`e = m.popt()`	tar bort ett slumpmässigt element (kan ge felsignal)
`m.copy()`	ger en kopia av `m`
`e in m`	ger `True` om `e` ingår i `m`
`m1.issubset(m2)`	ger `True` om `m1` är en delmängd av `m2`
$\mathbf{m1} >= \mathbf{m2}$	samma som förra raden
$\mathbf{m1} < \mathbf{m2}$	ger `True` om `m1` är en äkta delmängd av `m2`
`m1.issuperset(m2)`	ger `True` om `m2` är en delmängd av `m1`
$\mathbf{m1} >= \mathbf{m2}$	samma som förra raden
$\mathbf{m1} > \mathbf{m2}$	ger `True` om `m2` är en äkta delmängd av `m1`
`m1.isdisjoint(m2)`	ger `True` om `m1 & m2` är tom
`set.union(m1, m2)`	ger unionsmängden av `m1` och `m2`
`m1 | m2`	samma som förra raden

`set.intersection(m1, m2)`	ger snittmängden av `m1` och `m2`
`m1 & m2`	samma som förra raden
`m1.difference(m2)`	ger en mängd med element i `m1`, men inte i `m2`
`m1 - m2`	samma som förra raden
`m1.symetric_difference(m2)`	ger en mängd med elementsom finns i `m1` eller `m2`, men inte i båda
`m1 ^ m2`	samma som förra raden

222

Som exempel på användning av mängder kommer här ett program som läser in två textfiler och som skriver ut de ord som finns i båda filerna.

[fullständigt program]

```
# Jämför texter
def skapa_ordmängd(f) : # hjälpfunktion
        m = set() # en tom mängd
        bort = '.,?!:;' # tecken som ska ersättas
med ' '
        for s in f:
                for c in bort:
                        s = s.replace(c, ' ')
                s = s.lower()
```

```
                ord = s.split() # skapa en lista
med ord på raden
                m = m | set(ord) # utöka mängden
        return m
# Här börjar exekveringen
n1 = input('Den första filens namn?')
n2 = input('Den andra filens namn?')
with open(n1, 'r') as f1, open(n2, 'r') as f2:
        m1 = skapa_ordmängd(f1) # orden i första
filen
        m2 = skapa_ordmängd(f2) # orden i andra
filen
        m = m1 & m2 # gemensamma ord
        print(m)
```

Programmet börjar med att läsa in namnen på de två filer som ska jämföras. Därefter öppnas båda filerna i en `with`-sats. För var och en av filerna anropas sedan funktionen `skapa_ordmängd`. Denna bildar och returnerar en mängd som innehåller de ord som finns i den fil `f` den får som parameter. Sist i programmet bildas snittmängden av de två mängderna.

Funktionen `skapa_ordmängd` börjar med att skapa en tom mängd `m` som ska utökas med de ord som finns i filen. För att man lätt ska kunna hitta orden i den inlästa texten ska alla specialtecken, som t.ex. punkt och komma, ersättas med blanka tecken. Variabeln `blanka` innehåller de specialtecken som ska ersättas. En rad i taget läses från filen. För varje rad anropar man funktionen `replace` för vart och ett av de tecken som finns i variabeln `blanka`. Därefter översätts alla stora bokstäver till små. Efter detta bildar man en lista med alla ord på raden och dessa ord läggs till i mängden `m` genom att man bildar unionsmängden av de ord som redan finns i `m` och orden på raden. När alla raderna har läst från filen returneras mängden `m`.

Uppgift 12.1

Skriv ett program som läser in de färger som ett antal länder har i sina flaggor. Programmet ska sedan skriva ut alla de färger som förekommer i flag-gorna samt den eller de färger som eventuellt ingår i samtliga flaggor. *Tips:* Skapa en lista med lika många element som antalet länder. Låt varje element i listan vara en mängd som innehåller färgerna i landets flagga. Använd sedan operationerna union och snitt.

12.2 Typen `dict`

En *avbildningstabell (map, associative array* eller *dictionary* på engelska) är en tabell där man använder en s.k. *söknyckel* för att komma åt information. Ett exempel är ett bilregister. Där utgör registreringsnumret söknyckeln. Med hjälp av detta kan man ta fram information om bilen, t.ex. bilmärke och årsmodell. Söknyckeln *avbildas* på ett *värde* (informationen). En söknyckel och tillhörande värde bildar ett par, en s.k. *avbildning.* På engelska används ofta termen *key-value pair.* Varje söknyckel kan bara avbildas på *ett* värde. (Samma bil kan inte ha flera registreringsnummer.) En viss söknyckel kan därför bara finnas en enda gång i en avbildningstabell. Däremot kan ett visst värde förekomma flera gånger. Om man t.ex. har en avbildningstabell där söknycklarna är namn på personer och värdena är åldrar, kan flera personer ha samma ålder. Man kan tänka sig att en avbildningstabell ser ut som i figur 12.1, där man använder ett namn som söknyckel för att slå upp ett telefonnummer.

Tabellbeskrivning

Tabellen har 5 rader och 2 kolumner. Kolumnrubrikerna visar Key och Value.

Figur 12.1 En avbildningstabell.

Key	Value
Andersson Johan	0703123456
Bengtsson Åsa	0701987654
...	...
Åhman Jenny	081234567

I Python finns standardtypen `dict` som man kan använda för att skapa avbildningstabeller. Det finns flera olika sätt som man kan använda för att skapa en tabell. Ett sätt är att, inom tecknen `{` och `},` räkna upp

---

avbildningarna, dvs. söknycklarna och tillhörande värde. Man skriver ett kolon mellan söknyckeln och värdet, t.ex.

```
ålder = {'Julia' : 10, 'Anton' : 12, 'Ellen' : 13}
```

Ett annat sätt är att använda sig av funktionen `dict`.

```
rum = dict(Andersson = 201, Bergman = 204, Ek =
201)
```

Observera att man här inte ska skriva några apostrofer runt namnen. I båda dessa exempel blir söknycklarna texter av typen `str`. Det ser vi om vi skriver ut tabellerna:

```
print(ålder)
print(rum)
```

Utskrifterna blir

```
{'Julia': 10, 'Anton': 12, 'Ellen': 13}
{'Andersson': 201, 'Bergman': 204, 'Ek': 201}
```

Söknycklarna behöver inte vara texter. De kan t.ex. vara numeriska värden. Däremot får de inte vara listor eller av någon annan ändringsbar typ. När det gäller värdena finns inga begränsningar; de kan vara av vilken typ som helst. Vi kan t.ex. konstruera en tabell där söknycklarna är rumsnummer och där värdena är listor med de personer som sitter i rummen:

```
personal = {201 : ['Andersson', 'Ek'], 204 :
['Bergman']}
```

Det finns fler varianter när man använder funktionen `dict`. Man kan t.ex. som argument ge en lista med tupler, där varje tupel motsvarar en avbildning. Vi kan t.ex. skriva

```
ålder = dict([('Julia', 10), ('Anton', 12),
('Ellen', 13)])
```

Resultatet blir detsamma som ovan.

Det går att foga samman två listor till en avbildningstabell genom att använda en standardfunktion som heter `zip`. Denna får två listor som parametrar och ger avbildningar som resultat. Vi kan t.ex. skriva:

```
anställda = ['Andersson', 'Bergman', 'Ek']
rumsnr = [201, 204, 201]
rum = dict(zip(anställda, rumsnr))
```

Även här får vi samma resultat som tidigare.

Ytterligare ett sätt att använda funktionen `dict` är att ge en annan avbildningstabell som argument. Då får man en kopia av denna:

```
rum_kopia = dict(rum)
```

Funktionen `len` ger antalet avbildningar i en tabell:

```
len(ålder) # ger 3
```

Man kan använda indexering för att avläsa och ändra värden i en tabell och för att lägga till nya avbildningar:

```
rum['Ek'] # ger 201
ålder ['Ellen'] # ger 13
personal [201] # ger ['Andersson', 'Ek']
rum['Bergman'] = 207 # ändra rumsnummer för
Bergman
rum['Larsson'] = 211 # lägg till en ny avbildning
```

På sista raden ser vi att om man anger en söknyckel som inte finns, läggs det till en ny avbildning. Däremot blir det fel om man försöker avläsa en söknyckel som inte finns:

```
ålder['Alva'] # ger KeyError: 'Alva'
```

Ett alternativt sätt att avläsa ett värde är att använda funktionen `get`. Då kan man ange ett defaultvärde som man får om den angivna söknyckeln inte finns i tabellen. Då undviker man `KeyError`.

```
ålder.get('Julia', −1) # ger 10
ålder.get('Alva', −1) # ger −1
```

Om man vill lägga till en avbildning, men inte ändra den ifall den redan finns, kan man anropa funktionen `setdefault`.

```
rum.setdefault('Ek', 215) # ändrar inget, ger 201
rum.setdefault('Quist', 215) # lägger till Quist,
ger 215
```

Vill man ta bort en avbildning kan man använda operatorn `del`:

```
del rum['Andersson'] # tar bort Andersson
del rum['Okänd'] # ger KeyError: 'Okänd'
```

Vill man undvika att få `KeyError` kan man använda funktionen `pop` i stället för `del`. Om nyckeln finns, tas den bort och man får värdet som resultat. Om nyckel inte finns, ges ett defaultvärde som resultat.

```
rum.pop('Quist', -1) # tar bort Quist, ger 215
rum.pop('Okänd', -1) # ändrar inget, ger -1
```


Uppgift 12.2

Kör Python-interpretatorn i *interactive mode* och skapa en avbildningstabell som översätter de romerska siffrorna I, V, X, L, C, D och M till vanliga heltal. Siffrorna står för 1, 5, 10, 50, 100, 500 respektive 1 000. Låt söknycklarna vara texter. Testa olika

sätt att skapa tabellen. Skriv också uttryck som indexerar i tabellen.

Med operatorn `in` kan man testa om en viss söknyckel finns i en tabell:

```
'Julia' in ålder # ger True
'Arvid' in ålder # ger False
```

Operatorn `in` kan också användas för att löpa igenom en tabell. Man kan antingen löpa igenom söknycklarna:

```
for k in ålder:
        print(k)
```

vilket ger utskriften

```
Julia
Anton
Ellen
```

eller så kan man löpa igenom värdena:

```
for v in ålder.values():
        print(v)
```

Det går också att löpa igenom söknycklarna och värdena på en gång:

```
for k, v in ålder.items():
        print(k, v)
```

Vi får då utskriften

```
Julia 10
Anton 12
Ellen 13
```

Precis som när det gällde listor och mängder gäller det att namnet på en tabell i själva verket är en *referens* till själva tabellen. Detta betyder att om vi t.ex. skriver `ålder2 = ålder` pekar de två variablerna `ålder` och `ålder2` till *samma* tabell. Vill vi ha en kopia av själva tabellen måste vi använda funktionen `dict` eller funktionen `copy`:

```
ålder2 = ålder.copy()
```

Vi kan undersöka om de två tabellerna är lika med operatorn `==`:

```
ålder2 == ålder # ger True
```

Men om vi ändrar något i en av tabellerna är de inte längre lika:

```
ålder2 ['Ellen'] = 14
ålder2 == ålder # ger False
```

Man kan alltså använda operatorerna `==` och `!=` för att jämföra avbildningstabeller. De övriga jämförelseoperationerna, t.ex. `<`, finns inte.

Operationer för typen `dict`

Här betecknar `t` en avbildningstabell, `k` en söknyckel och `v` ett värde.

Tabellbeskrivning

Tabellen har 17 rader och 2 kolumner.

`dict` *`(argument)`*	skapar en avbildningstabell (se exempel)
`len(t)`	ger antalet avbildningar i `t`
`t[k]`	ger värdet för söknyckeln `k`
`t[k] = v`	ändrar eller lägger till värdet för söknyckeln `k`
`t.get(k, d)`	ger värdet för söknyckeln `k` om den finns, annars `d`

`t.setdefault(k, d)`	lägger till värdet `d` för söknyckeln `k` om den saknas; ändrar inget om den redan finns
`del t[k]`	tar bort avbildningen för söknyckeln `k`
`t.pop(k, d)`	tar bort avbildningen för `k` och ger dess värde som resultat; om `k` saknas ges i stället värdet `d`
`t.popitem()`	tar bort och returnerar (som en tupel) den senast inlagda avbildningen i `t`
`t.clear()`	tar bort alla avbildningar i `t`
`t.copy()`	ger en kopia av `t`
`t.update(t2)`	uppdaterar avbildningarna i `t` från `t2`, befintliga ändras, nya läggs till
`t.keys()`	ger en lista med söknycklarna i `t`
`t.values()`	ger en lista med värdena i `t`
`t.items()`	ger en lista med avbildningarna i `t` som tupler
`k in t`	ger `True` om söknyckeln `k` finns i `t`
`t1 == t2`	ger `True` om tabellerna har samma avbildningar

På sidan 222 visades ett program som jämförde två filer och undersökte vilka ord som var gemensamma. Vi ska nu visa ett liknande program, men denna gång ska vi bara undersöka en fil. Programmet ska skriva ut

hur många gånger varje ord förekommer i filen. Programmet använder två olika avbildningstabeller. Så här ser det ut:

[fullständigt program]

```
# Textanalys
n = input('Filens namn? ')
with open(n, 'r') as f:
        d = dict() # en tom avbildningstabell
        bort = '.,?!:;' # tecken som ska ersättas
med ' '
        t = str.maketrans(bort, ' ' * len(bort))
        for s in f:
                s = s.translate(t) # ersätt alla
specialtecken
                s = s.lower()
                ord = s.split()
                for e in ord:
                        d[e] = d.setdefault(e, 0)
+ 1 # öka antalet
        for k, v in d.items():
                print(k, v)
```

Först läses filens namn in och filen öppnas i en `with`-sats. En tom avbildningstabell skapas sedan. Den ska senare ha orden som söknycklar och antal som värden. Texten `bort` innehåller alla specialtecken som kan finnas i filen och som måste ersättas med blanka tecken för att man ska kunna plocka ut orden på ett korrekt sätt. I programmet på sidan 222 löpte vi igenom dessa tecken för varje inläst rad och ersatte dem med blanka tecken. Här ska vi i stället utnyttja en standardfunktion som heter `translate` för att ersätta alla specialtecken med blanka. Funktionen `translate` ska

ha en avbildningstabell som parameter. I programmet heter denna tabell `t`. Avbildningstabellen `t` innehåller en avbildning för varje specialtecken, där specialtecknet är söknyckel och tecknet `' '` värdet. Ett enkelt sätt att skapa tabellen `t` är att använda en annan standardfunktion som heter `maketrans`. Denna ska ha två lika långa texter som parametrar. Den första texten innehåller de tecken som ska vara söknycklar och den andra texten de tecken som ska vara motsvarande värden.

När alla specialtecken tagits bort för en inläst rad översätts alla stora tecken till små och funktionen `split` används som tidigare för att skapa en lista med orden på raden. För varje ord anropas funktionen `setdefault` med den andra parametern satt till 0. Om ordet redan finns i tabellen kommer `setdefault` att som resultat ge antalet gånger ordet tidigare förekommit i texten, men om ordet inte fanns tidigare

---

skapas en ny avbildning med värdet 0 och detta värde ges som resultat. Genom att addera 1 till resultatet får man på detta sätt antalet gånger ordet nu förekommit i texten. Programmet avslutas genom att man löper igenom avbildningstabellen `d` och skriver ut orden (söknycklarna) och antalet gånger orden förekommit (värdena).

Vill vi att söknycklarna ska skrivas ut i alfabetisk ordning kan vi göra en ändring på näst sista raden:

```
for k, v in sorted(d.items(), key=alfa):
```

Funktionen `items` ger som resultat en lista med avbildningar, där varje avbildning är en tupel med två element: nyckeln och värdet.

Standardfunktionen `sorted` används för att sortera denna lista. Men för att sorteringen ska ske korrekt enligt det svenska alfabetet har vi gett funktionen `alfa` som extra argument till funktionen `sorted`. Jämför med hur vi gjorde på sidan 156. Funktionen `alfa` ser ut som på sidan 150, men vi har gjort ett litet tillägg som markerats med rött:

[fullständigt program]

```
def alfa(s):
        if type(s) is tuple: # ska tuplar
jämföras?
                s = s[0] # välj första elementet
        s = s.lower()
        v = list(s)
        for i in range(0, len(v)):
                if v[i] == 'ä':
                        v[i] = 'å'
                elif v[i] == 'å':
                        v[i] = 'ä'
        return v
```

Tidigare användes funktionen `alfa` enbart för att jämföra texter. Då hade parametern `s` alltid typen `str`. Men nu vill vi även kunna jämföra tupler. Det första elementet i varje tupel är en text som innehåller nyckeln, och det är den texten som ska jämföras med andra nycklar.

12.3 Lagra data med JSON

Vi har tidigare diskuterat hur man kan skriva och läsa text till och från filer. Om vi gör som tidigare och försöker lagra hela listor eller

avbildningstabeller på en gång uppstår ett problem, nämligen att listorna och

---

avbildningstabellerna förlorar sin typ och blir rena texter. Anta t.ex. att vi har en lista `la` och skriver ut den till en fil `f`:

```
la = [1, 2, 3]
print(la, file=f)
```

Anta vidare att vi senare i programmet, eller i ett annat program, försöker läsa in hela filen `f`:

```
1b = f.read() # Blir FEL
```

Då kommer variabeln `lb` *inte* att bli en lista, utan en `str` som innehåller texten `'[1, 2, 3]'`. Vi kan alltså inte på detta sätt spara en lista i en fil och läsa tillbaka den som en lista. Samma problem uppstår om vi skulle försöka spara en avbildningstabell.

För att kunna spara hela listor och avbildningstabeller i filer och kunna läsa in dem och återställa dem, ska vi i stället använda ett lagringsformat som heter `JSON` *(JavaScript Object Notation).* Detta är ett standardiserat format som används för att överföra data mellan olika program. Informationen lagras som text i ett format som kommer från språket JavaScript och som liknar formatet för listor och avbildningstabeller i Python. Men man behöver inte känna till syntaxen för detta format för att kunna använda JSON. I Python kan man

enkelt lagra listor och avbildningstabeller med hjälp av funktionerna `dump` och `load` som finns `i` en standardmodul med namnet `json`. Vi kan nu göra om samma sak som ovan. Först lagrar vi listan `la` i filen `f`:

```
import json
la = [1, 2, 3]
json.dump(la, f)
```

Senare läser vi in hela listan från filen `f`:

```
lb = json.load(f)
```

Variabeln `lb` kommer nu att bli en lista som är en kopia av `la`.

Det är lika enkelt att lagra avbildningstabeller med JSON. Det finns bara en restriktion för att det ska fungera perfekt: söknycklarna måste vara texter. Här är ett exempel. Först lagrar vi tabellen `ta` i filen `f`:

```
li = [11, 12, 13, 14]
ta = {'ett' : 1, 'två' : True, 'tre' : li, 'fyra'
: 'hej'}
json.dump(ta, f)
```

Senare återskapar vi tabellen:

```
tb = json.load(f)
```

Tabellen `tb` kommer att bli en kopia av `ta`. Lägg märke till att värdena i avbildningstabellen kan vara av vilken typ som helst. Det är bara nycklarna som bör vara texter när man använder JSON.

Det går också att lagra tupler med JSON, men då omvandlas de till listor när man läser in dem. Däremot går det inte att lagra mängder.

Som exempel på avbildningstabeller och lagring med JSON ska vi visa en ny lösning på det problem vi studerade i avsnitt 11.4 på sidan 206. Det gällde att ändra i en fil som innehöll namn på elever och deras poäng på ett antal prov. I avsnitt 11.4 visade vi två lösningar på detta problem: Den första läste in hela filen till en lista, ändrade i listan och skrev ut den igen. Den andra lösningen använde en temporär fil. Här ska vi i stället använda en fil som innehåller en hel avbildningstabell i JSON-format. Programmet läser in hela tabellen, gör ändringar i den och skriver sedan ut den igen i JSON-format. Så här kan det se ut när man kör programmet:

```
Filen med resultat? resultat.json
Avsluta med tom rad
Namn? Mikael Bergman
Poäng? 20 30
Namn? Jenny Nilsson
Poäng? 10 13 15
Namn? Här trycker användaren på Enter
```

Om filen inte redan finns, kommer en ny fil att skapas som innehåller det man skrivit in. Om filen finns, läggs nya elever till eller ändras.

Här kommer programmet. Förklaringar följer efteråt.

[fullständigt program]

```
import json
# Steg 1, läs in tabellen eller skapa en ny
f_namn = input('Filen med resultat? ')
try:
        with open(f_namn, 'r') as f:
                tab = json.load(f) # läs in
tabellen
except FileNotFoundError:
        tab = dict() # skapa en tom tabell
```

```
# Steg 2, ändra i tabellen
print('Avsluta med tom rad')
while True:
        namn = input('Namn? ').lower()
        if namn == ' ':
                break
        rad = input('Poäng? ')
        poängen = [int(p) for p in rad.split()]
        tab[namn] = poängen
# Steg 3, lagra tabellen i filen
with open(f_namn, 'w') as f:
        json.dump(tab, f)
```

I det första steget försöker vi öppna filen. Om filen finns innehåller den en avbildningstabell i JSON-format. Vi läser i så fall in denna med

funktionen `load`. Om filen inte finns, skapar vi en tom avbildningstabell.

I det andra steget gör vi ändringar i avbildningstabellen. Varje avbildning i tabellen innehåller ett elevnamn som nyckel och en lista med poäng som värde. För varje elev läser vi först in namnet och sedan poängen på en egen rad. Vi använder som vanligt funktionen `split` för att skapa en lista med orden på raden. Med hjälp av *list comprehension* går vi igenom listan och skapar en lista med heltal. Denna lista lägger vi sedan in som värde i avbildningstabellen för den aktuella eleven. Fanns eleven redan i tabellen, kommer värdet för eleven att ändras och om eleven inte fanns, läggs en ny elev in.

I det sista steget lagrar vi helt enkelt avbildningstabellen i filen. Filens gamla värde skrivs då över.

Uppgift 12.3

Skriv ett program som läraren kan använda för att slå upp och visa elevers resultat. Använd samma teknik som i programmet ovan.

12.4 Sammanfattning

Efter att ha läst detta kapitel bör du:

- veta vad en containertyp är,

- kunna använda typerna `set` och `frozenset` för att bilda mängder,
- kunna skapa och använda avbildningstabeller med typen `dict`,
- veta hur man kan lagra data i en fil med hjälp av JSON.

12.5 Övningar

12.1 Morsealfabetet visas i följande tabell, där punkter och streck används för att beteckna korta respektive långa signaler.

```
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 --.. Å .-.- Ä .-.-. Ö ---.
```

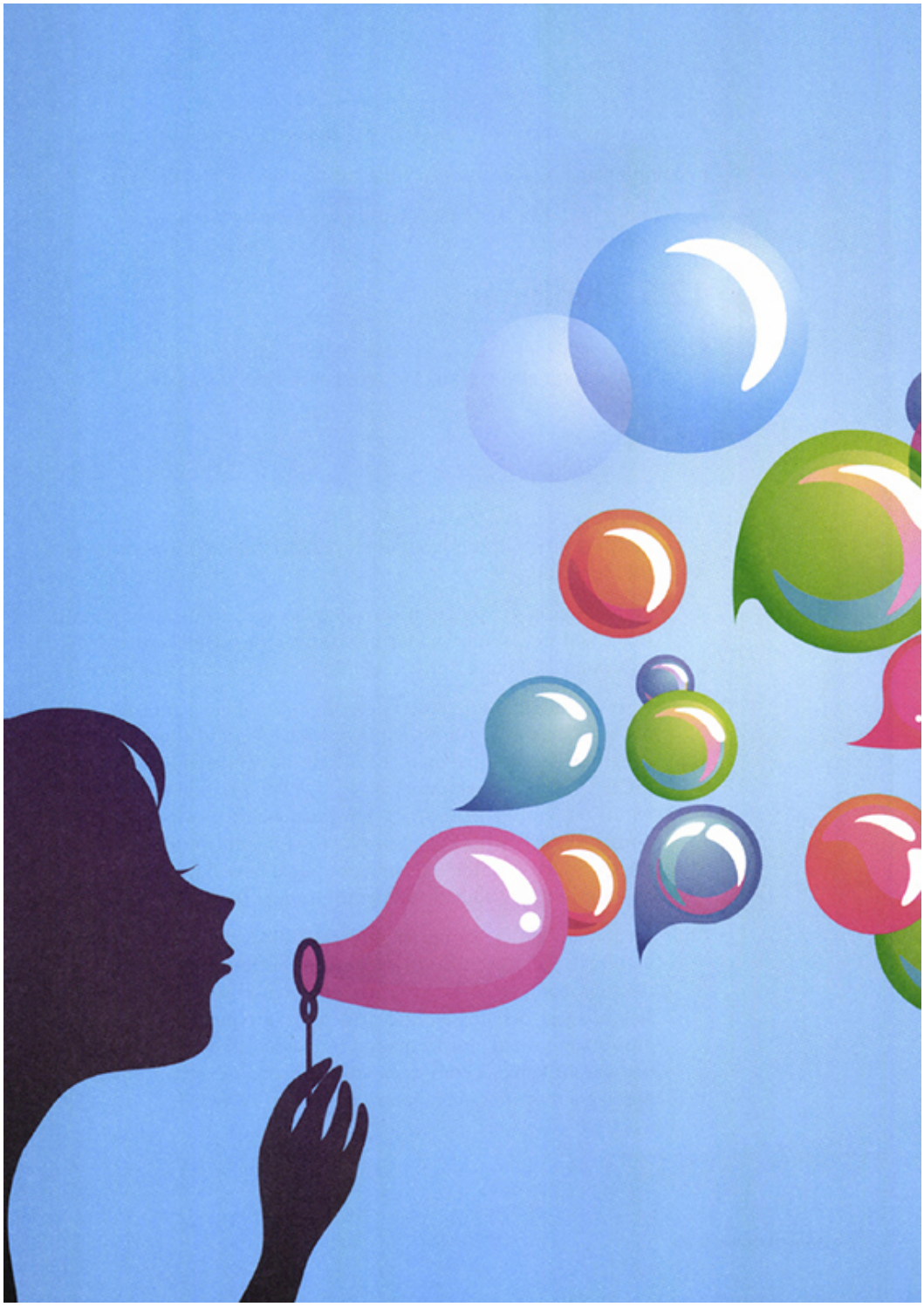
Skriv ett program som läser in ett meddelande och översätter det till morsekod.
Skriv därefter ett program som läser in ett meddelande givet i morsekod och som avkodar och skriver ut meddelandet. I det inlästa meddelandet finns ett blankt tecken mellan varje bokstav.

12.2 Skriv ett program som läser in ett romerskt tal och som översätter det romerska talet till ett vanligt heltal. Använd dig av avbildningstabellen från uppgift 12.2. Om användaren t.ex.

skriver MCMXLIX, ska programmet skriva ut 1949. I ett romerskt tal gäller att om en romersk siffra P står omedelbart till vänster om en annan romersk siffra Q och om P betecknar ett mindre tal än Q, ska värdet av P subtraheras från det totala talet (LIX betyder t.ex. 59), annars ska P adderas till det totala talet (LXI betyder 61).

12.3 Ett företag har ett varulager med flera olika slag av artiklar. För varje artikelslag är följande information intressant: artikelbeteckning (en kod), artikelbeskrivning (en text), antal artiklar av detta slag i lagret och försäljningspris. Skriv ett program som låter användaren mata in information om lagret till en avbildningstabell där artikelbeteckningarna är söknycklar. Låt sedan programmet lagra informationen i en fil med hjälp av JSON.

13 Klasser och objekt



Vi har hittills i boken använt Pythons inbyggda typer, t.ex. numeriska typer, listor och avbildningstabeller, men man kan också konstruera egna typer, så kallade *klasser.* Med hjälp av klasser kan man sedan skapa *objekt.* I detta kapitel ska vi beskriva hur klasser och objekt fungerar i Python. Vi

kommer då in på en teknik inom programmering som kallas *objektorienterad programmering.* Programspråk som har de konstruktioner som behövs för att skapa och hantera objekt brukar kallas *objekt-orienterade språk.* Typiska exempel på sådana är Java och C++. Även om man kanske inte i första hand tänker på Python som ett objekt-orienterat språk, är det faktiskt genomsyrat av objektorientering. Alla variabler i Python, även enkla som t.ex. heltal, har nämligen i språket implementerats som objekt. Konstruktionerna i Python för att beskriva klasser och objekt är visserligen inte lika "rena" som t.ex. i Java, men möjligheterna finns, och det ska vi diskutera i detta kapitel.

13.1 Objektorientering

Vi ska börja med att beskriva objektorientering rent allmänt. Den klassiska bilden av ett datorprogram är en låda i vilken man stoppar in indata och får ut utdata. Programmets uppgift är alltså att transformera ett dataflöde. Detta traditionella sätt att uppfatta datorprogram på brukar kallas det *funktionsorienterade* synsättet. Det *objektorienterade* synsättet är annorlunda. Enligt detta uppfattar man ett datorprogram som en *modell* av den verklighet programmet ska samverka med. De enskilda enheterna i ett program, de s.k. *objekten,* är då modeller av verkliga eller tänkta ting i programmets omgivning. Ett datorprograms uppgift är att manipulera objekten. I programmet använder man s.k. *klasser* för att avbilda och beskriva objektens egenskaper.

Låt oss börja med att diskutera begreppet objekt. Varje objekt har en unik identitet. I ett Python-program kommer man åt objekten via variabler som refererar till dem. Ett objekt karakteriseras av ett antal *attribut.* Dessa kan delas in i två kategorier: attribut som beskriver objektets *tillstånd* och attribut som beskriver dess *operationer.* Varje objekt har ett bestämt tillstånd, eller *status,* som kan förändras under exekveringen.

För att hålla reda på detta tillstånd används variabler, s.k. *tillståndsvariabler.* Varje objekt har en egen uppsättning sådana variabler. Normalt brukar man gömma tillståndsvariablerna inne i objektet, så att de blir oåtkomliga utifrån och bara kan förändras av objektet självt på ett kontrollerat sätt. Detta kallas *inkapsling* eller *information hiding.* Som exempel kan vi tänka oss ett objekt `hissen` som beskriver en verklig hiss. Dess tillstånd kan beskrivas med hjälp av de två variablerna `riktning` och `våning`. Variabeln `riktning` kan ha något av värdena `stilla`, `uppåt` eller `neråt` och variabeln `våning` kan innehålla ett heltal som anger vilken våning hissen är på. Olika objektorienterade språk använder olika termer för "tillståndsvariabel". Ord som används ofta är "instansvariabel" och "datamedlem". Vi kommer att använda ordet *instansvariabel.* (Du kanske undrar varför det heter instansvariabel. Det beror på att ordet *instans* används i betydelsen "förekomst av".)

Det andra som karakteriserar ett objekt är de *operationer* som man kan utföra på objektet. Objektet `hissen` kan ha t.ex. operationerna `kör_till`, `stanna` och `vilken_våning`. Operationen `kör_till` använder man för att få hissen att köra till en viss våning, `stanna` används för att stoppa hissen och `vilken_våning` används för att ta reda på var hissen f.n. befinner sig. I de objektorienterade språken kallas ofta operationer av detta slag "metoder". Det gör de också i Python. Därför kommer vi i fortsättningen att använda ordet *metod.*

Objekt

Modell av ett verkligt eller tänkt föremål.

Karakteriseras av: tillstånd och operationer.

Beskrivs med attribut som är instansvariabler resp. metoder.

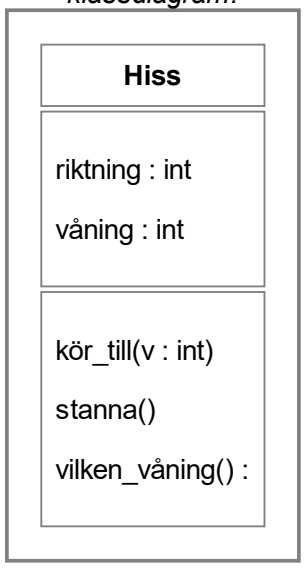
Ett objekt är alltså en modell av ett verkligt eller tänkt föremål. Men hur beskrivs då objekt i ett program? Det är här begreppet *klass* kommer in. En klass är en mall, eller mönster, som beskriver hur objekt med samma uppbyggnad och egenskaper ser ut. En klass är alltså en *beskrivning.* Man kan ha olika klasser som beskriver olika sorters objekt.

Klass

Beskrivning av objekt med samma egenskaper.

I figur 13.1 ges en grafisk bild av klassen `Hiss`. Vi använder i figuren ett diagram med notation enligt UML, *Unified Modeling Language.* I ett klassdiagram placeras klassnamnet överst, instansvariablerna i mitten och metoderna underst. Variablernas typer och typerna för metodernas parametrar har angetts, även om detta inte är nödvändigt.

Figur 13.1 Ett klassdiagram.



Man säger att ett objekt som tillhör en viss klass är en *instans* av klassen. Det kan finnas flera objekt som tillhör en viss klass. I figur 13.2 visas ett UML-diagram med två objekt av klassen `Hiss`.

Figur 13.2 Ett objektdiagram.

För varje objekt i figuren anges överst (understruket) objektets namn samt vilken klass objektet tillhör, och därefter visas instansvariablernas värden. Det finns en hiss med namnet `hissA` som är på väg uppåt från våning 2 och en hiss med namnet `hissB` som står stilla på våning 5.

13.2 Klassdefinitioner – enkla objekt

I detta avsnitt ska vi visa hur man i Python definierar enkla klasser som bara innehåller instansvariabler, dvs. inte några metoder. Som första exempel kommer här en klass som beskriver personer:

```
class Person:
        def_init_(self):
                self.förnamn = ' '
                self.efternamn = ' '
                self.född_år = ' '
                self.singel = True
```

Klassen `Person` används för att beskriva egenskaper för personer. Genom att definiera klassen `Person` definierar man en *ny datatyp.* Denna datatyp kan sedan användas i programmet när man vill skapa flera olika personer, på samma sätt som man t.ex. kan ha flera variabler av typen `list` för att beskriva olika listor.

Klassen `Person` innehåller fyra instansvariabler: `förnamn, efternamn, född_år` och `singel`. I Python skapar man normalt instansvariabler inne i en speciell initieringsfunktion som heter `_init_`. Observera att det ska vara *två* understrykningstecken före och efter ordet `init`. Observera också att funktionen `_init_` ska skrivas indragen så att den ligger inne i klassen.

Funktionen `_init_` ska ha en parameter som heter `self`[1]. Parametern `self` är en referens som pekar på det objekt man ska skapa. (Mer om detta följer senare.) I Python skapas variabler när man tilldelar värden till dem. Det är därför man måste ge värden till de fyra variablerna. Här har vi gett dem initieringsvärden som bestämmer vilken typ de får. Vad som händer inne i funktionen `_init_` är alltså att fyra variabler skapas och placeras inne i det objekt `self` pekar på, dvs. i det objekt man vill skapa.

Du ska få se ett exempel till. Det handlar om väderobservationer. De väderprognoser som SMHI gör grundar sig på att man får in ett stort antal observationer av vädret från olika meteorologiska stationer på olika orter. Förr i tiden gjordes väderobservationer manuellt och rapporterades in per telefon, men numera är de flesta stationerna automatiserade. En väderobservation kan i enklaste fall innebära att man mäter temperatur, vindhastighet och vindriktning. Varje meteorologisk station har ett unikt nummer. Om man sparar detta nummer vet man

därför var observationen gjorts. Man behöver också veta tiden för observationen. Därför sparas klockslaget. För att beskriva resultatet av en väderobservation kan man använda följande klass:

```
class Observation:
        def _init_(self):
                self.nr = 0 # stationens nummer
                self.tim = 0 # tidpunkt för
observationen
                self.min = 0
                self.temp = 0.0 # temperatur
                self.vindhast = 0.0 # mäts i m/s
                self.vindrikt = 0 # mäts i grader (0-
360)
```

Lägg märke till att i båda exemplen har klassnamnet skrivits med en stor begynnelsebokstav. Man brukar ofta låta klassnamn börja på en stor bokstav. Det blir då lättare att se vad som är namn på klasser.

Enkel klassdefinition och instansvariabler

```
class Klassnamn:
        def _init_ (self):
                self.var1 = initieringsvärde
                self.var2 = initieringsvärde
                ...
```

Uppgift 13.1

Definiera en klass som beskriver bilar. Kalla klassen `Bil`. En bil ska ha registreringsnummer, fabrikat, årsmodell, tjänstevikt och motoreffekt. Lägg klassen i en egen fil med namnet `Bil.py`. Du kan köra filen, men det händer inget, eftersom det inte finns något fullständigt program ännu.

13.3 Hur man skapar objekt

En klassdefinition är ett sätt att beskriva en *typ.* Men att det finns en typ betyder inte att man automatiskt får några variabler av den typen. Att t.ex. standardtypen `int` finns betyder ju inte att det finns några färdiga `int`-variabler när man börjar skriva sitt program. Vill man ha några sådana måste man skapa dem själv. På samma sätt är det med klasser

som man deklarerar själv. Vill man ha variabler av den typ som klassen beskriver, t.ex. personer, måste man skapa dem själv.

När det gäller ting som beskrivs av klasser brukar man använda ordet objekt i stället för variabel. Om man vill ha tre personer i sitt program, skapar man alltså tre objekt av typen `Person`. Man kan också säga att man skapar tre objekt av klassen `Person`. När man väl har definierat en klass kan man skapa hur många objekt som helst av denna klass, precis som man kan skapa hur många variabler som helst av typen `int`.

Du ska nu få se hur man skapar objekt av en klass. Som exempel kommer klassen `Observation` från förra avsnittet att användas. När vi tidigare skapade t.ex. listor och avbildningstabeller använde vi typnamnen `list` och `dict` som om de vore funktioner. Vi kunde t.ex. skriva

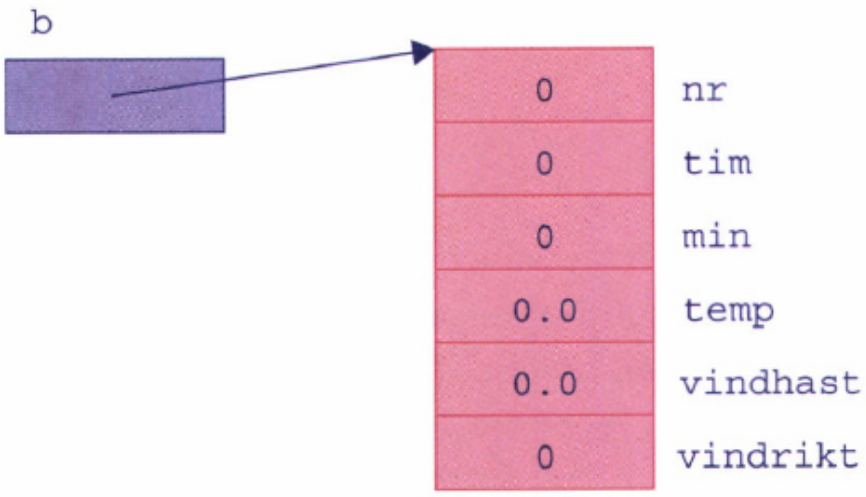
```
li = list() # skapa en lista
```

På samma sätt gör man om man vill skapa ett objekt av en egen klass. För att skapa ett objekt av klassen `Observation` skriver vi:

```
b = Observation() # skapa ett objekt
```

Här skapas utrymme för ett objekt av klassen `Observation` och referensvariabeln `b` sätts att peka på objektet. Se figur 13.3.

Figur 13.3 En referensvariabel som pekar på ett objekt.



Figuren visar variabeln b som pekar på ett objekt med sex variabler: 0 nr, 0 tim, 0 min, 0.0 temp, 0.0 vindhast, och 0 vindrikt.

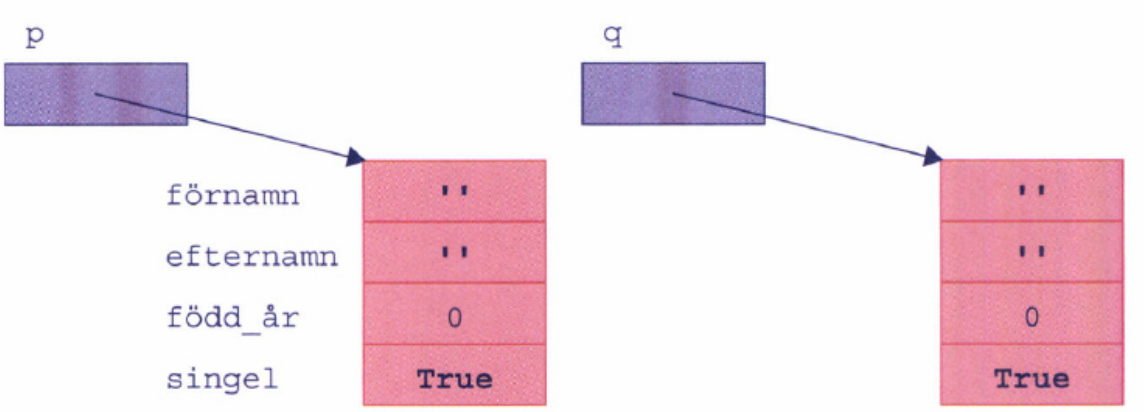
Här får vi ett objekt med sex variabler. När man skapar ett nytt objekt initieras alltid instansvariablerna i funktionen `_init_`.

Man kan naturligtvis skapa flera objekt av samma typ:

```
p = Person()
q = Person()
```


Här initieras de två variablerna `p` och `q` så att de kommer att peka på var sitt nytt objekt av klassen `Person`. Se figur 13.4.

Figur 13.4 Två objekt av klassen Person.



Figuren visar först variabeln p som pekar på ett objekt med fyra variabler: förnamn ’ ’, efternamn ’ ’, född_år 0, och singel True. Sedan variabeln q som pekar på ett objekt med fyra variabler: ’ ’, ’ ’, 0, och True.

Om man som i figur 13.4 har skapat två objekt av klassen `Person`, kan man säga att det finns två *instanser* av klassen `Person`. Man använder ordet instansvariabel för att markera att det är en variabel som det finns ett exemplar av för *varje* instans av klassen. I klassen `Person` finns instansvariabeln `född_år`. Om man har skapat två objekt av klassen `Person`, har ju varje `Person` sitt eget födelseår.

Att skapa objekt

```
referens = Klassnamn()
```

Funktionen `_init_` anropas automatiskt

Uppgift 13.2

Skriv ett program som skapar två objekt av klassen `Bil` som du definierade i uppgift 13.1. När du kör programmet bör inget hända eftersom programmet ännu inte innehåller några rader som ger utskrifter.

13.4 Hur man kommer åt instansvariabler

För att komma åt enskilda element i en sekvens, t.ex. i en lista, använder man indexering. Man kan säga att indexet är elementets nummer i

sekvensen. De instansvariabler som ingår i ett objekt har inga nummer. Man kan därför inte indexera när man vill komma åt en enskild instansvariabel, för att ändra den eller avläsa dess värde. Man använder sig i stället av s.k. *punktnotation.* Först skriver man vilket objekt det gäller, därefter en punkt och sedan namnet på den instansvariabel man vill komma åt. Om man har skapat ett objekt av klassen `Observation`:

```
b = Observation()
```

kan man skriva:

$$\mathrm{b.nr} = 8431$$

$$\mathrm{b.tim} = 19$$

$$\mathrm{b.min} = 30$$

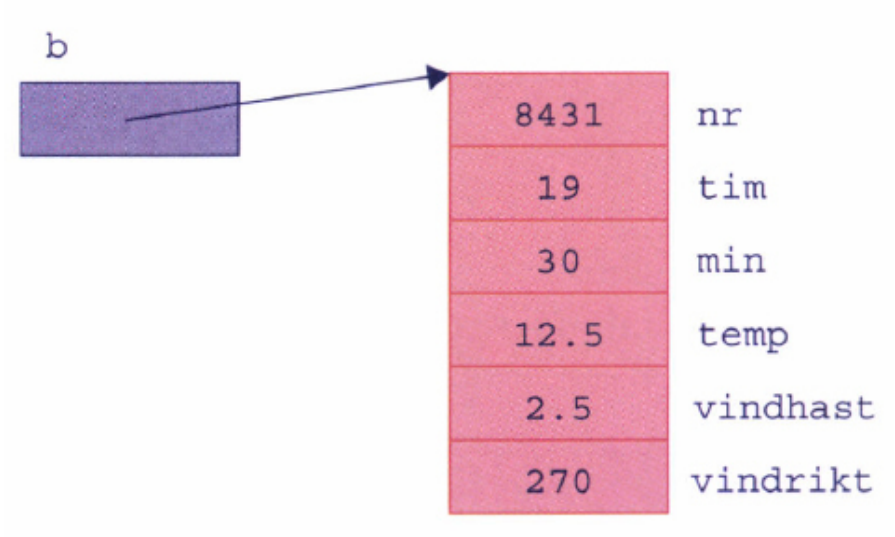
$$\mathrm{b.temp} = 12.5$$

$$\mathrm{b.vindhast} = 2.5$$

$$\mathrm{b.vindrikt} = 270$$

Då ändras värdena för instansvariablerna i det objekt som `b` refererar till. Hur det kommer att se ut visas i figur 13.5.

Figur 13.5 Ett objekt med ändrade värden på instansvariablerna.



Figuren visar variabeln b som pekar på ett objekt med sex variabler: 8432 nr, 19 tim, 30 min, 12.5 temp, 2.5 vindhast, och 270 vindrikt.

Man kan använda punktnotation även när man vill avläsa värdena utan att ändra dem. Man kan t.ex. ha satserna:

```
print('Mätstation', b.nr)
print(f'{b.tim}:{b.min} var temperaturen '
        f'(b.temp:0.1f} grader')
```

Utskriften blir då:

```
Mätstation 8431
19:30 var temperaturen 12.5 grader
```

Uppgift 13.3

Komplettera programmet i uppgift 13.2 så att instansvariablerna i de två `Bil`-objekten tilldelas lämpliga värden. Välj värden för två olika bilmodeller. Låt sedan programmet skriva ut informationen för de båda bilarna.

Det bör också nämnas här att man kan lägga till nya instansvariabler genom att tilldela värden till en variabel som inte redan finns, och att man kan ta bort instansvariabler med hjälp av operatorn `del`. Men detta är dock tveksamt ur ett objektorienterat perspektiv, eftersom man då kan bilda objekt som tillhör samma klass, men som har olika uppsättningar instansvariabler.

13.5 Referenser till objekt

Eftersom man alltid kommer åt objekten via referenser får man vara försiktig med tilldelningar. Anta t.ex. att vi skapar ett objekt av klassen `Observation` och låter variabeln `b1` refererar till objektet:

```
b1 = Observation()
```

Om vi nu skriver

```
b2 = b1
```

kopieras bara själva referenserna. Detta betyder att variablerna `b1` och `b2` kommer att peka på *samma* objekt. Om vi t.ex.skriver

```
b2.nr = 8800
```

kommer även `b1.nr` att få värdet `8800`.

Det fungerar alltså på samma sätt som när vi tilldelade listvariabler till varandra. För listor fanns en funktion med namnet `copy` som man kunde använda när man ville ha en kopia av själva listan. För klasser man definierar själv finns det också en funktion `copy`. Denna ligger i en modul som också den heter också `copy`, och man måste importera den modulen. Så här kan det se ut:

```
import copy
b3 = copy.copy(b1) # skapa en kopia av objektet
b3.nr = 8750 # b1 påverkas inte
```

Här skapas ett helt nytt objekt som `b3` pekar på och som innehåller samma data som `b1`. Ändrar vi något i `b3` påverkar det alltså inte `b1`.

En instansvariabel kan vara av vilken typ som helst. Den kan alltså även vara en referensvariabel. Som exempel på detta kan du studera följande klass som beskriver ett bankkonto:

```
class Konto:
        def _init_ (self):
                self.kontohavare = None
```

```
        self.saldo = 0
        self.intjänad_ränta = 0
```

På varje bankkonto finns en viss behållning, kontots saldo. För varje konto måste man också hålla reda på hur mycket ränta som tjänats in hittills under året. (Räntan måste beräknas dag för dag, eftersom man ju kan ta ut och sätta in pengar på kontot när som helst. Men man lägger normalt inte till räntan till saldot förrän vid årsskiftet. Därför måste man hålla reda på den intjänade räntan.) De två sista instansvariablerna beskriver saldot och den intjänade räntan.

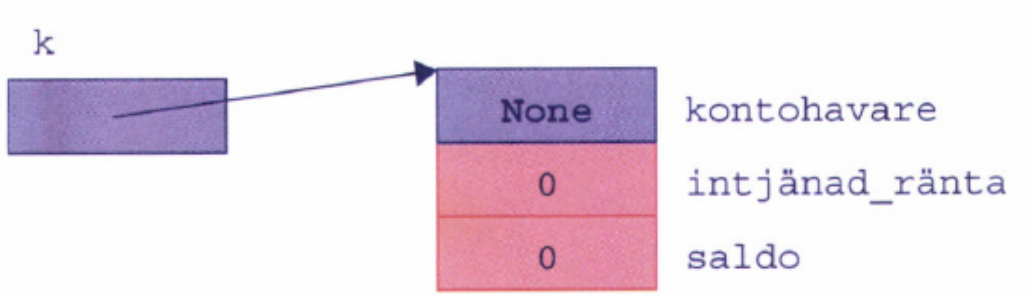
Det som är mest intressant just nu är emellertid den första instansvariabeln, `kontohavare`. För varje konto finns det förstås någon som äger pengarna på kontot. Här antas att kontohavaren alltid är en fysisk person. Det är meningen att variabeln `kontohavare` senare ska referera till ett objekt av typen `Person` som vi definierade på sidan 238. I funktionen `_init_` har vi gett variabeln `kontohavare` värdet `None`, vilket kan tolkas som att den ännu inte refererar till något objekt.

Om vi skapar ett nytt objekt av typen `Konto`:

```
k = Konto()
```

får vi bilden i figur 13.6.

Figur 13.6 Ett nytt Konto-objekt.



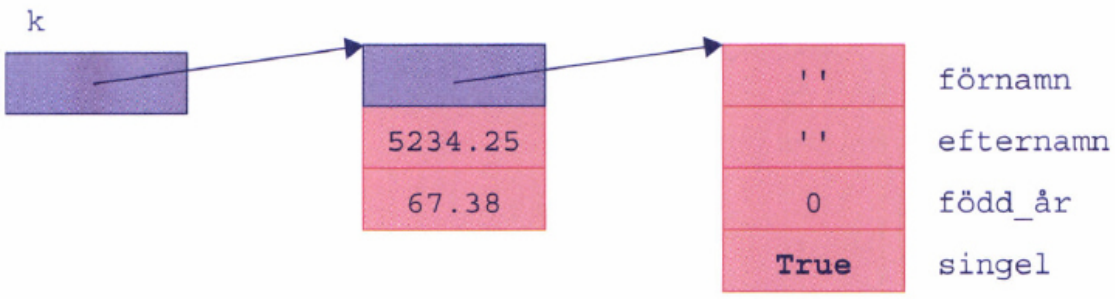
Figuren visar variabeln k som pekar på ett objekt med tre variabler: None kontohavare, 0 intjänad_ränta, och 0 saldo.

Nästa steg är att ge instansvariablerna värden. Vi kan t.ex. skriva:


```
k.kontohavare = Person()
k.saldo = 5234.25
k.intjänad_ränta = 67.38
```

(För att detta ska fungera måste klassen `Person` ligga i samma modul som klassen `Konto`, eller i en annan modul som importeras.) Efter detta kommer det att se ut som i figur 13.7.

Figur 13.7 Läget efter tilldelning till Konto-objektet.



Figuren visar variabeln k som pekar på ett objekt med tre variabler som pekar på ett objekt med fyra variabler. De första tre variablerna visar k, 5234.25, och 67.38. De senare

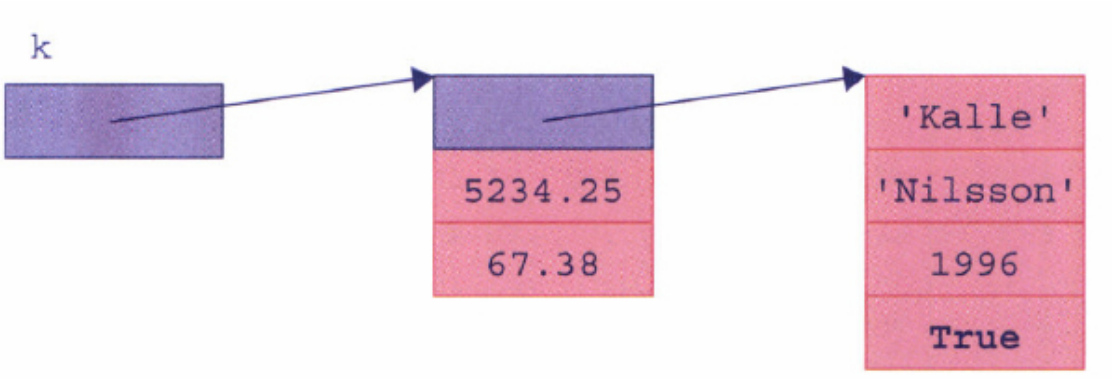
fyra variablerna visar: ’ ’ förnamn, ’ ’ efternamn, 0 född_år, och True singel.

Du ser att instansvariabeln `kontohavare` nu pekar på ett nytt objekt av typen `Person`. Man har alltså fått en situation där ett objekt innehåller en referens till ett annat. Man brukar i objektorienterade sammanhang säga att det första objektet *känner till* det andra. Ett `Konto`-objekt känner alltså till vem som är kontohavare. Man kan nu nå det andra objektet via det första. Det går t.ex. att skriva:

```
k.kontohavare.förnamn = 'Kalle'
k.kontohavare.efternamn = 'Nilsson'
k.kontohavare.född_år = 1996
```

Lägg märke till att dubbel punktnotation har använts här. Den första punkten väljer ut instansvariabeln `kontohavare` i det objekt som `k` pekar på. Den andra punkten väljer ut de olika instansvariablerna i det `Person`-objekt som instansvariabeln `kontohavare` pekar på. Efter detta kommer det att se ut som i figur 13.8.

Figur 13.8 Läget efter tilldelning till Person-objektet.



Figuren visar variabeln k som pekar på ett objekt med tre variabler som pekar på ett objekt med fyra variabler. De första tre variablerna visar k, 5234.25, och 67.38. De senare fyra variablerna visar: ’ Kalle’, ’Nilsson’, 1996, och True.

Lägger vi nu in satsen:

```
print(k.kontohavare.förnamn, 'har', k.saldo, 'kr')
```

får vi utskriften

```
Kalle har 5234.25 kr
```

Uppgift 13.4

Du ska nu göra några tillägg i programmet i övning 13.3. Lägg i klassen `Bil` in en referens till en person som är bilens ägare. Initiera sedan de två bilarna så att de får var sin ägare.

Låt oss se vad som händer om vi skapar en kopia av det objekt `k` pekar på. Vi använder funktionen `copy` och skriver

```
k2 = copy.copy(k)
```

Om vi nu ändrar instansvariablerna i `k2`, påverkar detta inte `k`:

```
k2.saldo = 6000
print(k.saldo) # ger utskriften 5234.25
```

Men låt oss se vad som händer om vi ändrar något i objektet som beskriver kontohavaren:

```
k2.kontohavare.singel = False
print(k.kontohavare.singel) # ger utskriften False
```

Här ändrades alltså instansvariabeln `singel` även för kontohavaren i objektet `k`, fast vi hade skapat en riktig kopia med hjälp av `copy`! Anledningen till att det blev så här är att funktionen `copy` kopierar alla instansvariabler rakt av. Detta kallas att den skapar en *grund kopia.* Den kopierade alltså variabeln `kontohavare`, som ju är en referens till ett `Person`-objekt. Detta betyder att objekten `k` och `k2` innehåller var sin referens till *samma* `Person`-objekt.

I modulen `copy` finns en mer avancerad kopieringsfunktion som heter `deepcopy`. Den skapar en s.k. *djup kopia.* Om någon av instansvariablerna som den ska kopiera är en referens till ett annat objekt, skapar den en kopia också av detta objekt. Vi kan i stället göra så här:

```
k2 = copy.deepcopy(k)
k2.kontohavare.singel = False
print(k.kontohavare.singel) # ger utskriften True
```

Nu kopieras objektet som beskriver kontohavaren. När vi ändrar i kontohavaren för `k2`, påverkar detta därför inte kontohavaren för `k`.

I exemplet du just har sett innehöll ett objekt av klassen `Konto` en referens till ett objekt av klassen `Person`, ett objekt av en annan klass. Men ett objekt kan också innehålla en referens till ett annat objekt av *samma* klass. Som exempel visas här en ny version av klassen `Person`:

```
class Person:
        def _init_ (self):
                self.förnamn = ' '
                self.efternamn = ' '
                self.född_år = ' '
                self.partner = None
```

Instansvariabeln `singel` har ersatts med en ny instansvariabel med namnet `partner`. Denna kan innehålla en referens till en annan person. Om variabeln har värdet `None`, innebär det att den aktuella personen är singel. Anta att vi utfört programraderna:

```
p = Person()
p.förnamn = 'Kalle'
p. född_år = 1996
print(p.förnamn, 'är', end=' ')
if p.partner == None:
                print('singel')
else:
        print('tillsammans med', p.partner.förnamn)
```

Eftersom instansvariabeln `partner` för objektet `p` fortfarande innehåller värdet `None` får vi utskriften

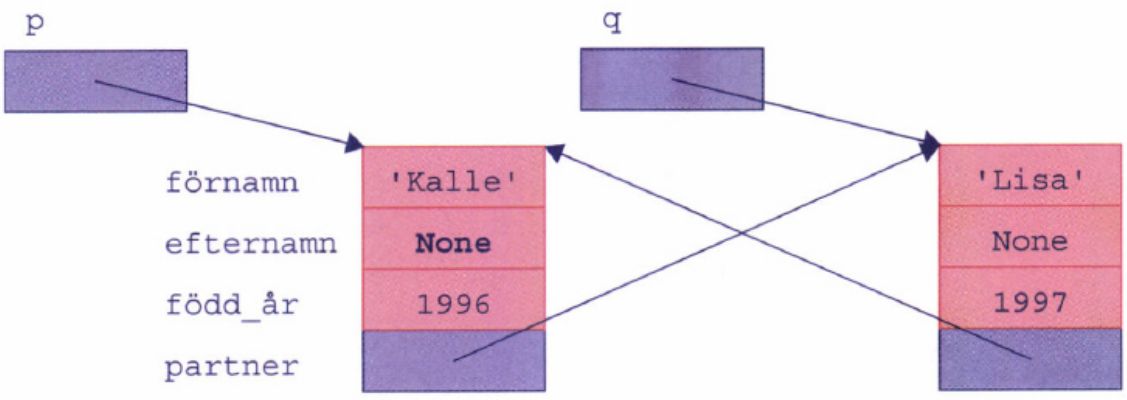
```
Kalle är singel
```

Om vi nu lägger till programraderna:

```
q = Person()
q.förnamn = 'Lisa'
q.född_år = 1997
p.partner = q
q.partner = p
```

skapas en ny person med namnet Lisa. De två personerna kopplas ihop så att instansvariabeln för Kalle kommer att peka på Lisa och tvärtom. Hur det ser ut demonstreras i figur 13.9.

Figur 13.9 Två objekt av klassen Person som känner till varandra.



Figuren visar först variabeln p som pekar på ett objekt med fyra variabler: förnamn ’Kalle’, efternamn ’None’, född_år 1996, men variabeln för partner saknas. Istället går en pil till variabeln q som pekar på ett objekt med fyra variabler: ’Lisa’, ’None’, 1997, och även här saknas variabeln för partner. Istället går en pil till Kalle.

Om vi nu åter utför satserna som skriver ut information om objektet `p`, får vi i stället utskriften

```
Kalle är tillsammans med Lisa
```

Om Kalle och Lisa tröttnar på varandra, är det lätt att skiljas:

```
p. partner = None
q. partner = None
```

13.6 Initiering av instansvariabler

Låt oss nu titta lite närmare på vad som händer när man skapar ett nytt objekt. Som exempel kan vi ta följande klass som beskriver cirklar:

```
class Cirkel:
        def _init_ (self):
                self. x = 0
                self. y = 0
                self. r = 0
```

Klassen har tre instansvariabler: dels koordinaterna `x` och `y` som anger mittpunkten, och dels radien. När man skapar ett nytt objekt, t.ex.

```
cl = Cirkel()
```

anropas alltid automatiskt funktionen `_init_` som finns i klassen. Denna funktions uppgift är att initiera instansvariablerna. (I Python skapas dessutom faktiskt instansvariablerna genom att man tilldelar till dem.) I objektorienterade sammanhang kallas en funktion som har till uppgift att initiera instansvariabler i nya objekt en *konstruktor (constructor*

på engelska). Det som utmärker konstruktörer är att man inte anropar dem direkt, utan att de anropas automatiskt när nya objekt skapas.

I funktionen `_init_` i klassen `Cirkel` har alla instansvariablerna till-delats konstanta initieringsvärden. Men man kan lägga till fler parametrar efter parametern `self`. Då kan den som skapar ett nytt objekt av klassen `Cirkel` själv bestämma vilka initieringsvärden instansvariablerna ska ha. Vi gör därför lite ändringar i klassen:

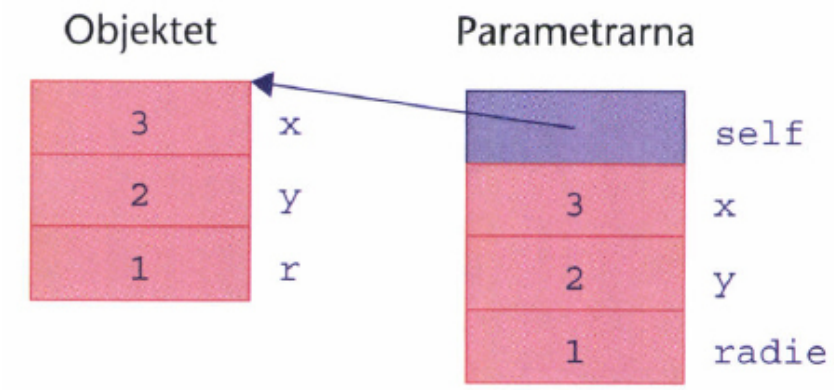
```
class Cirkel:
        def _init_ (self, x=0, y=0, radie=0):
                self.x = x
                self.y = y
                self.r = radie
```

Tre nya parametrar, `x, y` och `radie,` har lagts till. Vi använder parametrarnas värden för att initiera instansvariablerna. När vi nu skapar ett nytt objekt kan vi ge argument som motsvarar parametrarna. Vi kan t.ex. skapa en cirkel med mittpunkten (3, 2) och med radien 1:

```
c2 = Cirkel(3, 2, 1)
```

Här kommer funktionen `_init_` att anropas med argumenten 3, 2, och 1. Hur det ser ut när anropet skett illustreras i figur 13.10. Där visas tillståndet när alla satserna inne i funktionen har utförts, men innan funktionen har avslutats.

Figur 13.10 Initiering av Cirkel-objekt.



Figuren visar ett objekt och flera parametrar. Objektet har de tre variablerna 3 x, 2 y, och 1 r. Parametrarna visar fyra variabler: self (som pekar på objektet), 3 x, 2, y, och 1 radie.

Parametrarna har ritas till höger och objektet med sina instansvariabler till vänster. Den första parametern, `self`, läggs alltid till automatiskt vid anropet av `_init_`. Som visas i figuren pekar den på det objekt som ska initieras.

Om du studerar funktionen `_init_` lite närmare, ser du att den sista parametern heter `radie`, men att motsvarande instansvariabel heter `r`. Parametrarna och instansvariablerna är, som också framgår av figuren, *olika* variabler. De kan därför ha olika namn. De kan också ha samma namn, men i så fall är de ändå *olika* variabler. Parametrarna `x` och `y` har t.ex. samma namn som instansvariablerna `x` och `y`. Parametrarna fungerar på samma sätt som parametrar i vanliga funktioner. Det betyder att när funktionsanropet avslutas kommer de att försvinna. Instansvariablerna, däremot, kommer att finnas kvar så länge objektet existerar.

Lägg märke till att vi angett defaultvärden för parametrarna. Det betyder att om vi utelämnar några parametrar vid anropet, kommer deras defaultvärden att användas. Det går alltså fortfarande att skriva

```
c1 = Cirkel()
```

Då får alla instansvariablerna värdet 0. Vill vi skapa en cirkel med radien 10 i punkten (0, 0) kan vi skriva

```
c3 = Cirkel(radie=10)
```

Uppgift 13.5

Ändra i funktionen `_init_` i klassen `Bil` som du definierade i uppgift 13.1 så att instansvariablerna kan ges initieringsvärden när man skapar objekten. Skapa sedan två olika `Bil`-objekt. Välj värden för två olika bilmodeller. Låt sedan programmet skriva ut informationen för de båda bilarna.

13.7 Metoder

I avsnitt 13.1 diskuterade vi att objekt hade två kategorier av attribut: attribut som beskriver objektets *tillstånd* och attribut som beskriver dess *operationer.* Så här långt har vi bara behandlat attribut av det första slaget, nämligen instansvariabler. Det är nu dags att introducera attribut av det andra slaget: funktioner som kan användas för att utföra olika slag av operationer på objekt. Sådana funktioner kallas i den objektorienterade världen *instansmetoder.* De kallas så eftersom deras uppgift är att göra något med ett visst objekt, t.ex. avläsa ett värde ifrån det eller att ändra någon av dess instansvariabler. De kallas instansmetoder för att skilja dem från en annan kategori av metoder som heter klassmetoder. För enkelhets skull, och när det inte kan bli något missförstånd,

---

använder vi ibland bara ordet *metod,* när vi menar en instansmetod. Klassmetoder diskuteras i avsnitt 13.13.

Som första exempel ska vi utöka klassen `Cirkel` från förra avsnittet med tre metoder: en där man kan ändra en cirkels radie, en som beräknar en cirkels area och en som beräknar omkretsen.

```
from math import pi
class Cirkel:
        def _init_ (self, x=0, y=0, radie=0):
                self.x = x
                self.y = y
                self.r = radie
        def set_r(self, r):
                assert r >= 0, ('Negativ radie', r)
                self.r = r
        def area(self) :
                return pi * self.r ** 2
        def omkr(self):
                return 2 * pi * self.r
```

Det är två ting som gör att funktionerna `set_r, area` och `omkr` blir metoder och inte vanliga funktioner. För det första är de definierade inne i en klass (lägg märke till att texten är indragen) och för det andra har de en första parameter som pekar på ett objekt de är kopplade till. Denna parameter brukar man i Python ge namnet `self`.

Definition av instansmetod

```
def namn (self, p1, p2, etc.):
        en eller flera satser
```

`self, p1, p2, ...` är metodens *parametrar.*

Det kan finnas godtyckligt många, men self måste alltid finnas först, `self` pekar på det objekt som metoden är kopplad till.

Definitionen av metoden måste ligga inne i en klass (indragen).

Det är naturligt att börja med att diskutera metoden `set_r`. Denna har till uppgift att sätta cirkels radie till ett tillåtet värde. Radien måste vara större än eller lika med noll. Kontrollen görs med hjälp av funktionen

---

`assert`. (Se sidan 186.) Vi lägger till följande rader efter definitionen av klassen `Cirkel`:

```
c = Cirkel()
e = float(input('Cirkels radie? '))
c.set_r(e)
```

Den sista raden är ett *anrop av en instansmetod.* Ett anrop av en instansmetod gäller alltid för ett *visst* objekt. I detta exempel är det objektet `c`. Det är ju radien för `c` som ska sättas. Inget annat objekt ska påverkas. För att ange vilket objekt det gäller skriver man i anropet objektets namn först, före punkten (på samma sätt som man gör när man vill komma åt en instansvariabel för ett visst objekt). Därefter skriver man instansmetodens namn och dess argument. Lägg märke till att man i anropet *inte* ska ge något argument för parametern `self`.

Anrop av en instansmetod

Ett anrop av en instansmetod har formen

objekt.metodnamn(a1, a2, ... an)

objekt är en referens till ett objekt.

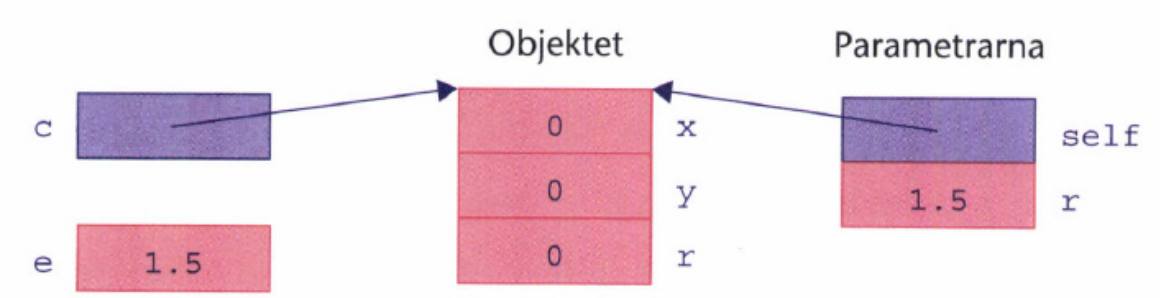
Man ska *inte* ge något argument för parametern `self`.

Det objekt som *objekt* refererar till kallas det ***aktuella objektet****,* *a1, a2, ... an* är ***argument.*** De får vara uttryck.

Nu är det dags att i detalj studera vad som händer vid anropet av instansmetoden `set_r`. Hur det ser ut precis när anropet påbörjas demonstreras figur 13.11. Variablerna längst till vänster ligger i main-modulen. Den referens till objektet som man skrivit före punkten i anropet, dvs. `c` i uttrycket `c. set_r (e)`, kopieras automatiskt till metodens första parameter `self`. Det betyder att `self` kommer att peka på samma objekt som `c` pekar på. Anta sedan att den som kört programmet har skrivit in värdet 1.5 och att detta värde därför ligger i variabeln `e`. I anropet av `set_r` anges `e` som argument. Därför kommer värdet av `e` att *kopieras* till parametern `r` i metoden. Figur 13.11 visar läget i just det ögonblick när detta skett.

Det som sedan sker är att satserna inne i metoden `set_r` utförs. Metoden hade utseendet:

Figur 13.11 Anrop av instansmetoden set_r (när anropet påbörjas).



Figuren visar variabeln c och e, samt ett objekt och flera parametrar. Variabeln c pekar på objektet som har de tre variablerna 0 x, 0 y, och 0 r. Parametrarna visar två variabler: self (som pekar på objektet), och 1.5 r. Variabeln e visar 1.5.

```
def set_r(self, r):
        assert r >= 0, ('Negativ radie', r)
        self.r = r
```

Eftersom parametern `r` i detta exempel är större än noll kommer den sista satsen att utföras. Där kopieras värdet 1.5 från parametern `r` till instansvariabeln `r` i objektet `c`. För att komma åt objektet `c` använder

metoden parametern `self`. Därefter avslutas metoden. Eftersom parametrarna bara existerar medan anropet pågår kommer parametrarna `self` och `r` att då upphöra att finnas till. Det som finns kvar är objektet och variablerna i main-modulen. Det enda som egentligen hänt är att instansvariabeln `r` för objektet `c` har ändrats till värdet 1.5.

Men hur "vet" metoden `set_r` att det är radien för `c` som ska ändras och inte radien för någon annan cirkel? Det står ju bara `self.r` i tilldelningssatsen. Det beror på att satsen finns *inne* i en instansmetod och att man har använt referensen `self`. Denna pekar alltid på det s.k. *aktuella objektet.* Vilket som är det aktuella objektet bestäms av vad som står före punkten vid *anropet* av instansmetoden. Här anropades metoden med satsen:

```
c.set_r(e)
```

Därför blev `c` det aktuella objektet. Hade vi i stället skrivit `c2.set_r(e)`, hade `c2` blivit det aktuella objektet.

Varför behövs allt detta? Kan man inte helt enkelt skriva:

```
c.r = e
```

Jovisst, det är precis så vi gjort tidigare i detta kapitel för att komma åt instansvariabler. Men för att undvika att cirkeln får en negativ radie bör man då, efter att man läst in det användaren skrivit, kontrollera att det inlästa värdet inte är negativt. Det blir ganska klumpigt, speciellt om man ska göra flera inläsningar. Om man använder metoden `set_r` görs

felkontrollen i stället inne i den metoden. Anta t.ex. att användaren skriver in värdet -1.5 i stället för 1.5 i exemplet. Då kommer programmet att avbrytas och man får utskriften:

```
AssertionError: ('Negativ radie', -1.5)
```

Nu när du sett hur instansmetoden `set_r` fungerar, kan du säkert förstå vad de två andra instansmetoderna `area` och `omkr` gör. För att räkna ut radien och omkretsen för cirkeln `c` kan man skriva

```
ar = c.area()
om = c.omkr()
```

Framför namnet på metoden skriver man, precis som för metoden `set_r`, namnet på det objekt det gäller, det aktuella objektet. Båda metoderna saknar parametrar. Därför står det ingenting innanför parenteserna, men de måste ändå alltid vara med.

Instansmetoden `area` hade utseendet:

```
    def area (self):
            return pi * self.r ** 2
```

Det enda metoden gör är att beräkna arean för den aktuella cirkeln. Här används referensen `self` för att metoden ska komma åt instansvariabeln `r` i det aktuella objektet.

Här kommer ett program som beräknar arean och omkretsen av två cirklar. Programmet förutsätter att definitionen av klassen `Cirkel` ligger i en fil som heter `cirkel.py`, så att denna kan importeras.

[fullständigt program]

```
from cirkel import Cirkel
r1 = float(input('Den första cirkels radie? '))
r2 = float(input('Den andra cirkels radie? '))
c1 = Cirkel()
c2 = Cirkel()
c1.set_r(r1)
c2.set_r(r2)
print('Den första cirkeln har arean', c1.area(), 'och
omkretsen', c1.omkr())
print('Den andra cirkeln har arean', c2.area(), 'och
omkretsen', c2.omkr())
```


Uppgift 13.6

Definiera en klass `Rektangel` som beskriver rektanglar. Lägg klassen i en fil som heter `rektangel.py`. Låt klassen innehålla instansvariabler som beskriver en rektangels startpunkt (dess övre vänstra hörn) samt dess höjd och bredd. Klassen ska också innehålla instansmetoder som sätter höjden och bredden samt beräknar arean och omkretsen.

Uppgift 13.7

Skriv ett program som testar klassen `Rektangel` i uppgift 13.6. Programmet ska läsa in höjden och bredden för ett godtyckligt antal rektanglar. För varje rektangel ska dess area och omkrets beräknas och skrivas ut. Ändra aldrig någon instansvariabel direkt från main-

modulen, utan anropa instansmetoderna. *Tips:* Använd en `while`-sats i programmet och skapa en ny rektangel på varje varv.

13.8 Inkapsling

Som nämndes i inledningen till detta kapitel är *inkapsling (encapsulation, information hiding)* en viktig princip inom den objektorienterade programmeringen. Den går ut på att man gömmer information inne i objekten, så att man bara kan komma åt informationen på ett kontrollerat sätt. Det finns två fördelar med detta. Den ena är att man utifrån inte kan komma åt känsliga instansvariabler direkt och ge dem felaktiga värden, så att ett objekt hamnar i ett otillåtet tillstånd. (Tänk t.ex. vad som skulle kunna hända om ett objekt av klassen `Hiss` skulle beskriva att en hiss befann sig på en våning som inte finns i verkligheten.) Den andra fördelen med inkapsling är att man kan välja ut precis de variabler och metoder som användare av klassen behöver känna till; man kan presentera ett snyggt och begripligt gränssnitt. Användare av en klass behöver inte belastas med onödig information.

En vanlig teknik för att åstadkomma inkapsling i de objektorienterade programspråken är att definiera vissa instansvariabler och instansmetoder som *privata.* Detta innebär att de bara är kända inne i själva klassen. De är kända i klassens instansmetoder, men inte utanför klassen. I Python finns inget verkligt stöd för att definiera privata variabler. Där

är alla attribut publika, dvs. åtkomliga utifrån. Någon riktig inkapsling går det därför inte att åstadkomma i Python, men genom att följa vissa rekommendationer kan man ändå komma en bit på vägen.

Låt oss som exempel fortsätta med klassen `Cirkel` från förra avsnittet. För att det inte skulle gå att ge ett negativt värde till radien definierade vi metoden `set_r`. Denna kontrollerade att radien man försökte ge var större än eller lika med noll. Problemet är att man inte behöver använda sig av metoden `set_r` för att ändra radien. Som klassen ser ut nu går det lätt att skapa en felaktig cirkel. Vi kan t.ex. ge en negativ radie när vi skapar cirkeln:

```
c = Cirkel(radie = -1)
```

eller så kan vi ändra instansvariabeln `r` direkt

```
c.r = -1
```

Det första av dessa sätt kan vi undvika genom att ändra i funktionen `_init_` i klassen `Cirkel`. I stället för att där på sista raden skriva

```
self.r = radie
```

kan vi anropa instansmetoden `set_r`:

```
self.set_r(radie)
```

Vi kan emellertid inte förhindra att någon ändrar instansvariabeln `r` direkt. Hade det funnits någon verklig inkapsling, hade vi kunnat göra detta omöjligt.

Hur gör man då i Python? Där får man lita på programmerarnas goda omdöme. Det finns en konvention som säger att man låter sådana instansvariabler och instansmetoder som ska vara privata ha namn som

börjar med ett understrykningstecken. Detta signalerar till dem som använder klassen att man inte ska röra dessa variabler eller anropa dessa metoder. I klassen `Cirkel` är det instansvariabeln `r` som borde varit privat. För att följa konventionen kan vi döpa om den till `_r`. Naturligtvis kan en programmerare som inte följer konventionerna komma åt instansvariabeln `_r` utifrån, men då sker det på egen risk.

Vi använder nu denna konvention i vår klass `Cirkel`. Vi låter alltså instansvariabeln som innehåller radien ha namnet `_r`:

```
from math import pi
class Cirkel:

        def _init_ (self, x=0, x=0, radie=0):
                self.x = x
                self.y = y
                self.set_r(radie)

        def get_r(self):
                return self._r

        def set_r(self, r):
                assert r >= 0, ('Negativ radie', r)
                self._r = r

        def area(self):
                return pi * self._r ** 2

        def omkr(self):
                return 2 * pi * self._r
```

De ändringar vi gjort jämfört med den tidigare definitionen på sidan 251 har markerats med rött. Förutom att ändra variabelnamnet till `_r` har vi också

lagt till en ny metod, `get_r`, som man kan använda för att avläsa radien. Lägg märke till att det inte längre finns någon instansvariabel som heter `r`. Det naturliga är nu att använda metoden `set_r` för att ändra radien och `get_r` för att avläsa den, t.ex.

```
c.set_r(5)
print(c.get_r())
```

Det är vanligt att man i de objektorienterade språken använder sig av metoder som heter `get_variabelnamn` och `set_variabelnamn` för att på ett kontrollerat sätt avläsa och ändra instansvariabler. Sådana funktioner brukar på engelska kallas "getters" och "setters". I vår klass `Cirkel` har vi nu sådana metoder för variabeln `_r`. Ibland vill man inte att man utifrån ska ändra en instansvariabel. Då kan man utesluta metoden `set_variabelnamn`. Som exempel kan vi ta klassen `Konto` på sidan 244. Vi kan utöka klassen med en instansvariabel som innehåller kontonumret. Detta nummer ska initieras när man skapar ett nytt konto, men det ska inte senare ändras. Vi löser det genom att använda en instansvariabel `_nr` och en get-funktion. Vi har lagt till kontonumret som parameter till funktionen `_init_` så att man måste ange det när man skapar ett nytt konto. Så här ser då klassen `Konto` ut:

```
class Konto:
        def _init_ (self, nr):
                self._nr = nr
                self.kontohavare = None
                self.saldo = 0
                self.intjänad_ränta = 0

        def get_nr(self):
```

```
        return self._nr
```

Uppgift 13.8

Ändra i klassen `Rektangel` från uppgift 13.6 så att instansvariablerna som beskriver höjden och bredden betraktas som privata. Lägg till get- och `set`metoder.

13.9 Privata attribut

[överkurs]

I klassen `Cirkel` på sidan 257 lät vi radien ha namnet `_r`. Understrykningstecknet först i namnet signalerade att variabeln ska betraktas som privat och att man inte ska röra den utifrån. Men som vi konstaterade i förra avsnittet är variabeln fullt synlig, och det finns inget som hindrar att man ändrar eller avläser den utifrån.

Det finns emellertid i Python ett sätt att göra instansvariabler och instansmetoder lite mer privata; man låter deras namn börja med *två* understrykningstecken. Då kommer Python-interpretatorn att automatiskt döpa om dem så att de får ett mer komplicerat namn. Detta kallas *name mangling.* Interpretatorn lägger till `_Klaassnamn` först till variabelnamnet. Om vi t.ex. döper om variabeln `_r` till `_r` överallt i definitionen av klassen `Cirkel`, kommer `_r` att ha namnet `_Cirkel_r` utanför klassen. Inne i klassen kan man fortfarande skriva `_r` för att komma åt radien.

Om man nu t.ex. skriver

```
c = Cirkel()
print(c._r)
```

kommer man att få utskriften

```
AttributeError: 'Cirkel' object has no attribute '_r'
```

Att använda två understrykningstecken först i namnet på sådana instansvariabler och instansmetoder som man vill ska vara privata kan alltså vara en bra idé, även om det inte följer Pythons konvention, som säger att man ska använda *ett* understrykningstecken.

13.10 Egenskaper – properties

[överkurs]

I klassen `Cirkel` är koordinaterna `x` och `y` för en cirkels mittpunkt publika. Om vi har en cirkel `c` kan vi därför enkelt ändra och avläsa mittpunktens koordinater genom att t.ex. skriva:

```
c.x = 3
print(c.y)
```

Anledningen till att vi inte markerat `x` och `y` som privata är att alla värden är tillåtna för dem, både positiva och negativa. De behöver alltså inte skyddas. Om klassen `Cirkel` ser ut som på sidan 257 bör vi använda metoderna `set_r` och `get_r` för att ändra och avläsa radien. För att det ska se ut på enhetligt sätt, skulle vi kanske också vilja att det gick att hantera radien lika enkelt som `x` och `y`. Vi skulle t.ex. vilja skriva

```
c.r = 5
print(c.r)
```

Detta är möjligt om vi definierar `r` som en *egenskap, property.* Detta kan enkelt göras i klassen `Cirkel` om vi lägger till följande rad sist i klassen:

```
r = property(get_r, set_r)
```

Här säger vi att `r` ska vara en egenskap och att funktionen `get_r` ska anropas för att avläsa egenskapen och funktionen `set_r` för att ändra den. Med detta tillägg kommer det att gå att tilldela värden till egenskapen `r` och att avläsa den lika enkelt som vi gjorde ovan. Vad som i själva verket sker är att funktionerna `get_r` och `set_r` anropas automatiskt. Detta betyder att instansvariabeln `_r` (eller `_r` om vi använt två understrykningstecken) ändras och avläses.

Man behöver inte ange både get- och set-metoder. I klassen `Konto` kan vi t.ex. lägga till raden

```
nr = property(get_nr)
```

Man kan emellertid definiera egenskaper ännu elegantare genom att använda s.k. *decorators.* I klassen `Cirkel` kan vi definiera en egenskap `r` genom att skriva på följande sätt:

```
@property
def r(self):
```

```
                return self._r
```

Detta säger att `r` är en egenskap. När man avläser den, utförs satsen `return self._r`. Denna definition ersätter alltså funktionen `get_r`. Vill man att det ska gå att ändra `r` kan man lägga till följande rader:

```
        @r.setter
                def r(self, r):
                        assert r >= 0, ('Negativ
radie', r)
                        self._r = r
```

Detta ersätter funktionen `set_r`. Vi kan nu lägga in de nya raderna i klassen `Cirkel`. Vi låter instansvariabeln som beskriver radien ha kvar namnet `_r`, men det går lika bra (eller bättre) att använda namnet `_r`.

```
from math import pi
class Cirkel:
        def _init_ (self, x=0, y=0, radie=0):
                self.x = x
                self.y = y
                self.r = radie

        @property
        def r(self):
                return self._r

        @r.setter
        def r(self, r):
                assert r >= 0, ('Negativ radie', r)
                self._r = r

        def area(self):
                return pi * self.r ** 2

        def omkr(self):
```

```
        return 2 * pi * self.r
```

Lägg märke till att vi i de andra metoderna i klassen, t.ex. i metoden `area`, kan använda egenskapen `r` i stället för instansvariabeln `_r`.

I `_init_` skriver vi `self.r = radie`. Då anropas "settern" för egenskapen `r` och där kontrolleras att radien inte är negativ.

I exemplen vi nu sett är egenskaperna direkt kopplade till en motsvarande instansvariabel. Men det är inte nödvändigt. Man kan mycket väl ha en egenskap vilkens värde beräknas. I klassen `Cirkel` kan vi t.ex. definiera egenskaperna `arean` och `omkretsen`:

```
    ©property
    def arean(self):
        return pi * self.r ** 2

    ©property
    def omkretsen(self):
        return 2 * pi * self.r
```

Om vi har en cirkel `c` kan vi nu skriva

```
print('Cirkeln har arean', c.arean,
      'och omkretsen', c.omkretsen)
```

Uppgift 13.9

[överkurs]

Ändra i din klass `Rektangel` så att den får egenskaper (properties) `h` och `b` som beskriver höjden och bredden. Använd decorators.

13.11 Metoden `_str_`

Ett objekt kan vara ganska komplicerat och innehålla många instansvariabler. Det går inte att utan vidare göra om ett sådant objekt till en vanlig text så att man t.ex. kan skriva ut det. Ett smart objekt "vet" emellertid hur den information som finns i objektet på bästa sätt ska presenteras i textform. Det görs med hjälp av en instansmetod med namnet `_str_`. Som exempel kan man ta följande klass som beskri ver tidpunkter. Objekt av denna klass har två instansvariabler, vilka innehåller timmar respektive minuter:

```
class Klockslag:
        tim = 0
        min = 0
```

Anta nu att man har skapat ett objekt `k1` av denna klass och initierat det nya objektet så att det innehåller tiden 08:30.

```
k1 = Klockslag()
kl.tim = 8
kl.min = 30
```

Nu vill man på ett enkelt sätt kunna få utskrifter som t.ex

```
Avgångstid 08:30
```

Det bästa sättet att göra detta är att i klassen `Klockslag` lägga till en instansmetod med namnet `_str_`. Så här ser den ut:

```
        def _str_ (self):
                return f'{self.tim:02}:{self.min:02}'
```

Metoden `_str_` ger som resultat en "textversion" av det aktuella objektet. Den anropas automatiskt om man försöker göra om ett objekt av typen `Klockslag` till typen `str`.

```
s = str(k1)
```

Variabeln `s` blir en `str` som innehåller texten `'08:30'`. I metoden `_str_` redigeras texten med hjälp av en *f-string* (Du minns väl från avsnitt 2.3 att formatet `02` betyder ett decimalt heltal som ska visas med två siffror och att utfyllnad ska ske med nollor?)

Standardmetoden `print` kan anropas direkt med en referens som parameter. Då anropas metoden `_str_` automatiskt. Anropet:

```
print('Avgångstid', k1)
```

ger t.ex. den önskade utskriften:

```
Avgångstid 08:30
```

För att kunna styra utskrifternas utseende måste man skriva en egen version av `_str_` i sin klass. Gör man inte det, får man en standard version, som ger en ganska ful text som bl.a. innehåller klassens namn.

Uppgift 13.10

Deklarera en klass `Datum`. Klassen ska innehålla de tre instansvariablerna `år, mån` och `dag`. Låt klassen innehålla en instansmetod `_str_` som returnerar ett datum som en text. Använd standardformatet *åååå-mm-dd.*

13.12 Att jämföra objekt

[överkurs]

Anta att vi har definierat tre objekt av klassen `cirkel`:

$$\mathrm{c1} = \mathrm{Cirkel}(1, 1, 3)$$

$$\mathrm{c2} = \mathrm{Cirkel}(2, 2, 3)$$

$$c3 = c1$$

Vi antar att klassen `cirkel` är definierad som på sidan 260. Då kan man ställa sig frågan: Är dessa objekt lika? Vi kan testa:

```
print(c1 == c2) # ger urskriften: False
print(c1 == c3) # ger utskriften: True
```

Cirklarna `c1` och `c2` är lika i den meningen att de har samma radie, men de uppfattas ändå som olika. Det beror på att operatorn == jämför referenserna, om man inte anger att något annat ska ske. Eftersom `c1` och `c2` pekar på två olika objekt uppfattas de därför som olika. Däremot uppfattas `c1` och `c3` som lika eftersom de pekar på samma objekt.

Om man själv vill bestämma hur jämförelser ska göras kan man lägga till en speciell metod med namnet `_eq_` i sin klass. Anta t.ex. att vi vill att två cirklar som har samma radie ska uppfattas som lika även om de har olika mittpunkt. Då kan vi utöka klassen `cirkel` från sidan 257 med följande metod:

```
def eq (self, other):
        return self.r == other._r
```

Om man nu använder operatorn == för att jämföra två objekt av typen `cirkel`, kommer metoden `_eq_` att anropas automatiskt. Metoden har två parametrar: `self` vilken kommer att peka på objektet till vänster om operatorn == och `other` som kommer att peka på objektet till höger om operatorn. Om vi t.ex. skriver

$$c1 == c2$$

, blir `self` en kopia av referensen `c1` och `other` en kopia av referensen `c2`. Inne i metoden jämförs sedan radierna för de två cirklarna. Vi kan nu testa igen:

```
print(c1 == c2) # ger urskriften: True
print(c1 == c3) # ger utskriften: True
```

Om man har definierat metoden `_eq_` får man också automatiskt en metod `_ne_` *(not equal).* Det går alltså att skriva:

```
print(c1 != c2) # ger urskriften: False
```

Det finns en operator med namnet `is`. Denna testar också om två objekt är lika, men den jämför alltid referenserna. Därför ger den värdet True bara om referenserna pekar på *samma* objekt, oberoende av om man definierat någon metod `_eq_` eller inte:

```
print(c1 is c2) # ger urskriften: False
print(cl is c3) # ger utskriften: True
```

Det finns möjlighet att definiera fler jämförelsemetoder, förutom `_eq_`. Som exempel visas här operatorn `_lt_` *(less than)* som undersöker om en cirkel är mindre än en annan:

```
def _lt_ (self, other):
        return self._r < other._r
```

Vi kan testa den nya metoden:

```
C4 = Cirkel(0, 0, 4)
print(c1 < c4) # ger utskriften: True
```

Andra jämförelsemetoder man kan definiera är `_le_` *(less than or equal),* `_gt_` *(greater than)* och `_ge_` *(greater than or equal).*

Uppgift 13.11

[överkurs]

Utöka klassen `cirkel` med jämförelsemetoder så att man kan använda operatorerna `<=, >` och `>=`. *Tips.* Om man har metoderna `_eq_` och `_lt_` går det alltid att utnyttja dessa för att definiera de övriga jämförelsemetoderna. Detta gäller för alla klasser.

13.13 Klassvariabler och klassmetoder

Alla de variabler du sett i detta kapitel och som har definierats i klasserna har varit s.k. instansvariabler. Dessa är, som du nu vet, sådana variabler som det finns en upplaga av för *varje* objekt som tillhör klassen. I klassen `Konto` på sidan 258 fanns t.ex. de fyra instansvariablerna `_nr, kontohavare, saldo` och `intjänad_ränta`. Det är självklart att varje bankkonto har ett unikt kontonummer, sin egen kontohavare, sitt eget saldo och sin egen intjänade ränta. Denna information är ju olika för olika konton.

Men ibland finns det sådan information som är gemensam för *alla* objekt som tillhör en viss klass. För bankkonton finns det t.ex. en räntesats

som man måste känna till för att kunna beräkna räntan för de olika kontona. Räntesatsen är normalt samma för *alla* bankkonton. Det räcker därför att lagra den på *ett enda* ställe. Den behöver inte finnas i en upplaga för varje konto. Därför ska man inte använda en instansvariabel för att komma ihåg räntesatsen. I stället ska en s.k. *klassvariabel* användas. Här nedan ser du hur klassen `Konto` ser ut efter det att man har lagt till en klassvariabel som ska innehålla räntesatsen:

```
class Konto:
        räntesats = 0 # klassvariabel
        def _init_ (self, nr):
                self.nr = nr
                self.kontohavare = None
                self.saldo = 0
                self.intjänad_ränta = 0

        def get_nr(self):
                return self._nr
```

En klassvariabel placeras och initieras direkt i klassen. Den initieras inte inne i funktionen `_init_` vilket instansvariabler gör. Om man nu skapar flera objekt av klassen `Konto` så kommer vart och ett av dem ändå att ha de fyra instansvariablerna `_nr, kontohavare, saldo` och `intjänad_ränt.` Variabeln `räntesats` lagras inte i något av objekten. I stället skapas ett separat utrymme för denna variabel i själva klassen. Om man t.ex. har utfört de två programraderna:

```
k1 = Konto(1111)
k2 = Konto(2222)
```

så får man bilden i figur 13.12.

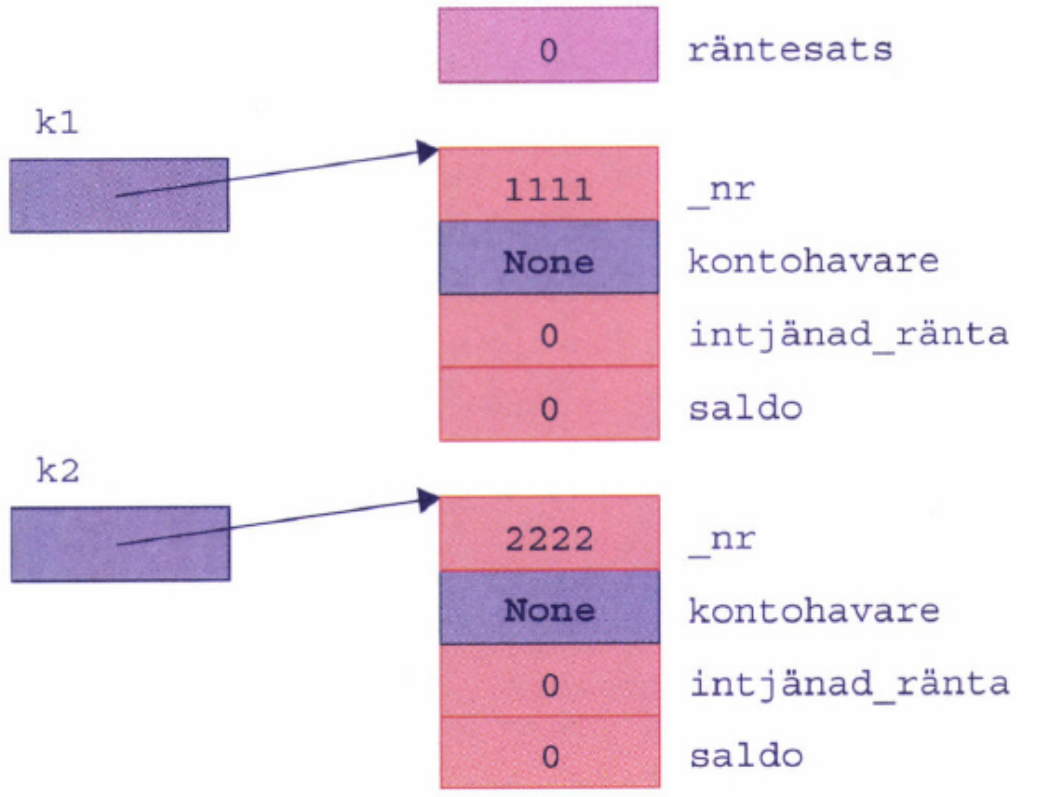
Utrymmet för klassvariabeln skapas automatiskt i och med att man använder klassen i sitt program. En klassvariabel existerar alltså t.o.m. även om man inte skapat några objekt av klassen.

För att komma åt en klassvariabel använder man punktnotation, precis som för instansvariabler. Man kan då skriva klassnamnet före punkten:

```
Konto.räntesats = 1.2
```

Detta fungerar även om det inte finns några objekt av klassen `Konto`.

Figur 13.12 Två Konto-objekt och en klassvariabel.



Figuren visar en räntesats på 0, samt konto k1 och k2. K1 visar de fyra variablerna 1111 _nr, None kontohavare, 0 intjänad_ränta, och 0 saldo. K2 visar de fyra variablerna 2222 _nr, None kontohavare, 0 intjänad_ränta, och 0 saldo.

En klassvariabel hör inte ihop med något speciellt objekt, men det går ändå att skriva en referens till ett visst objekt före punkten i stället för klassnamnet. Men man bör undvika detta skrivsätt eftersom det kan bli fel när man tilldelar ett värde till variabeln. I stället för att ändra klassvariabeln, kommer man då att skapa en ny instansvariabel med samma namn som klassvariabeln. Om vi t.ex. skriver

```
k1.räntesats = 0.9 # FEL!
```

skapas en instansvariabel med namnet `räntesats` i objektet `k1`. Klassvariabeln `räntesats` ändras inte.

Klassvariabler

Placeras och initieras direkt i klassen.

Finns bara i *en* upplaga, gemensam för alla objekt som tillhör klassen. För att komma åt dem skriver man *`Klassnamn. variabelnamn.`*

Du har i avsnitt 13.8 sett hur man kan ge en instansvariabel ett namn som börjar med ett understrykningstecken. Detta gjorde man för att markera att instansvariabeln är privat och att den inte ska ändras direkt utifrån. Vi kompletterade sådana instansvariabler med set- och getmetoder så att man kunde komma åt dem på ett säkert sätt. Man kan göra på samma sätt för klassvariabler. Anta t.ex. att man i klassen `Konto`

inte vill att räntesatsen ska kunna bli negativ. Då kan man döpa om klassvariabeln `räntesats` till `_räntesats` och definiera metoderna `set_räntesats` och `get_räntesats`. Eftersom dessa metoder är kopplade till själva klassen och inte till något speciellt objekt av klassen, kallas de *klassmetoder.* De ska kunna anropas på följande sätt:

```
Konto.set_räntesats(0.9)
print(Konto.get_räntesats() )
```

Man skriver alltså klassnamnet framför funktionsnamnet. (Det går också att skriva namnet på ett objekt framför punkten. Detta namn omvandlas i så fall automatiskt till en referens till klassen.)

Vi gör nu dessa ändringar och tillägg i klassen `Konto`:

```
class Konto:
        _räntesats = 0

        @classmethod
        def set_räntesats(cls, p) :
                assert p >= 0, ('Negativ ränta', p)
                cls._räntesats = p

        ©classmethod
        def get_räntesats(cls):
                return cls._räntesats

        resten som tidigare
```

Man markerar att en metod ska vara en klassmetod genom att skriva `@classmethod` på raden ovanför. På samma sätt som en instansmetod ska ha en extra första parameter som heter `self` ska en klassmetod ha en extra parameter som brukar kallas `cls`. När metoden anropas kommer

denna parameter automatiskt att referera till *klassen* (inte till något objekt av klassen som `self` gör). För att komma åt en klassvariabel inne i en klassmetod skriver man därför `cls` och en punkt före klassvariabelns namn, såsom vi gjort här.

En klassmetod kan inte komma åt instansvariabler, eftersom dessa hör ihop med enstaka objekt av klassen. Däremot kan en instansmetod komma åt en klassmetod genom att man skriver klassnamnet före variabelnamnet. Man kan t.ex. avläsa räntesatsen när man vill beräkna den intjänade räntan för ett visst konto. Du kan tänka dig att det finns en instansmetod `lägg_till_ränta` som ska anropas en gång per dygn.

Definition av klassmetod

```
®classmethod

def namn (cis, p1, p2, etc.):
    en eller flera satser
```

`cls, p1, p2,` ... är metodens *parametrar.*

Det kan finnas godtyckligt många, men `cls` måste alltid finnas först. `cls` pekar på den klass som metoden är definierad i.

En klassmetod kan bara komma åt klassvariabler, inte instansvariabler.

Anrop har formen `Klassnamn.metodnamn (argument)`

Man kan t.ex. skriva

```
k1.lägg_till_ränta()
```

Metoden beräknar hur mycket ränta kontot har tjänat det senaste dygnet och adderar detta till den intjänade räntan. Så här ser metoden ut:

```
def lägg_till_ränta(self):
        self.intjänad_ränta = self.intjänad_ränta + \
                self.saldo*Konto._räntesats/100/365
```

(Divisionen med 100 beror på att räntesatsen anges i procent och divisionen med 365 beror på att det gäller den ränta man har fått för en enda dag.) Lägg märke till att vi har skrivit `Konto._räntesats` för att avläsa räntesatsen. (Det hade också gått att skriva `self._räntesats` här eftersom vi bara avläser klassvariabeln. Hade vi däremot skrivit `self._räntesats` och försökt ändra klassvariabeln hade det blivit fel. Då hade det skapats en ny instansvariabel.)

Uppgift 13.12

Varje år måste ägaren till en bil betala fordonsskatt. Hur mycket man ska betala beror på flera olika saker, bl.a. på bilens tjänstevikt och vilken typ av fordon det är fråga om. Anta emellertid att det är så enkelt att fordonsskatten bara beror på tjänstevikten och att man får betala ett visst belopp per kg tjänstevikt. Komplettera din klass `Bil` från de tidigare uppgifterna i detta kapitel med en klassvariabel som anger skattebeloppet per kg. Initiera klassvariabeln med ett lämpligt värde, t.ex. 4.0. Komplettera sedan ditt program med satser som beräknar och visar fordonsskatten för dina bilar.

13.14 Arv

En sak som är speciell för objektorienterade språk är att klasser kan *ärva* egenskaper från andra klasser. När man konstruerar en ny klass kan man utgå från en klass som redan finns och lägga till ytterligare variabler och metoder. Man säger då att den nya klassen är en *subklass* till den klass man utgick från och att den klass man utgick från är en *superklass* till den nya klassen. Ibland kallar man subklassen *derived class* eller *child class* och superklassen *base class* eller *parent class.*

Som exempel ska du få se klasser som beskriver olika sorters hus. Alla klasserna ska vara subklasser (direkt eller indirekt) till en klass `Hus` som beskriver hus i största allmänhet. Klassen `Hus` ska ha instansvariablerna `längd` och `bredd`. Metoden `_str_` (se avsnitt 13.11) är definierad så att man kan få en utskrift av ett hus. Klassen `Hus` har också metoden `kvadratiskt` som undersöker om längden och bredden är lika och metoden `yta` som beräknar husets totala golvyta.

```
class Hus:
        def _init_ (self, l, b):
                self.längd = l
                self.bredd = b

        def kvadratiskt(self) :
                return self.längd == self.bredd

        def yta(self):
                return self.längd * self.bredd
```

Instansvariablerna `längd` och `bredd` skapas som vanligt i initierings-funktion `_init_` som automatiskt anropas när man skapar ett objekt av

klassen `Hus`. Instansvariablernas värden ska då ges som argument till parametrarna `l` och `b`. (Som du ser har vi inte här markerat instansvariablerna som privata genom att ge dem namn som börjar med ett understrykningstecken. Det borde vi gjort, men vi kan avstå från det i detta exempel. Det blir då lite enklare och vi slipper lägga till metoder för att avläsa och ändra variablerna.) Vi kan enkelt skapa ett objekt av klassen `Hus`:

```
h = Hus(20, 10)
```

Här får instansvariabeln `längd` värdet `20` och `bredd` värdet `10`.

270

Anta nu att vi vill beskriva flervåningshus. Vi kan då definiera en ny klass `Flervåningshus` som är subklass till klassen `Hus`. I den nya klassen vill vi lägga till en instansvariabel som anger hur många våningar ett hus har. Den nya klassen `Flervåningshus` ser ut så här:

```
class Flervåningshus(Hus):
        def _init_ (self, l, b, v):
                super(). _init_ (l, b)
                self.antal = v # antalet våningar

        def höghus(self):
                return self. antal >= 10

        def yta(self):
                return self.längd * self.bredd *
self.antal
```

På första raden står det `Hus` inom parentes. Det är detta som talar om att `Flervåningshus` är en subklass till klassen `Hus`. Precis som i andra klasser har vi definierat en funktion `_init_`. Denna anropas automatiskt när vi skapar ett objekt av klassen `Flervåningshus`, t.ex.

```
f = Flervåningshus(30, 15, 3)
```

Här har vi gett tre argument eftersom `_init_` har tre parametrar `l, b` och `v`, förutom `self`. På första raden i `_init_` anropas funktionen `_init_` i superklassen. Detta sker eftersom det står `super()` framför punkten. Som argument till superklassens `_init_` ges parametrarna `l` och `b`. Vad som händer i anropet av superklassens `_init_` är att de två instansvariablerna `längd` och `bredd` skapas och initieras. Det är viktigt att man anropar superklassens `_init_` på detta sätt i en subklass. Gör man inte det, skapas inte de instansvariabler som ska finnas i superklassen. Man bör helst lägga anropet av superklassens `_init_` först i subklassens `_init_`.

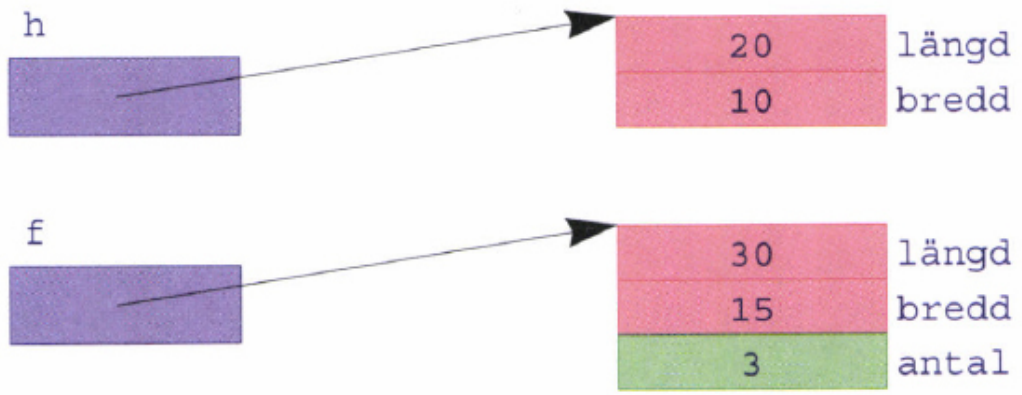
På sista raden i `_init_` i subklassen `Flervåningshus` skapar och initierar vi en tredje instansvariabel som ges namnet `antal`.

Hur de två objekten `h` och `f` som vi skapat här ser ut illustreras i figur 13.12. Lägg märke till att det objekt som `h` refererar till bara har en `Hus`-del (den del som har markerats med rött), medan det objekt som `f` refererar till har både en `Hus`-del och en `Flervåningshus`-del (den del som markerats med grönt).

Definition av subklass

```
class Klassnamn (superklass):
        def _init_ (self, parametrar):
                super(). _init_ (parametrar till
super())
                self.varl = initieringsvärde
                self.var2 = initieringsvärde
                ...

        Definitioner av metoder
                ...
```

Figur 13.13 Objekt av klasserna Hus och Flervåningshus.



Figuren visar variabeln h och f, samt varsitt objekt med flera parametrar. Variabeln h pekar på ett objekt som har de två variablerna 20 längd, och 10 bredd. Variabeln f pekar på ett objekt som har de tre variablerna 30 längd, 15 bredd, och 3 antal.

En subklass ärver alla de instansvariabler som finns i superklassen och kan dessutom ha sina egna instansvariabler. Därför kan vi t.ex. skriva

```
print(f.längd) # ger utskriften: 30
print(f.antal) # ger utskriften: 3
```

Däremot är det fel att försöka skriva

```
print(h.antal) # FEL!
```

eftersom ett objekt av klassen `Hus` inte har någon instansvariabel som heter `antal`.

En subklass ärver alla metoder från superklassen. Klassen `Flervåningshus` ärver därför metoden `kvadratiskt` från klassen `Hus`.

```
print(f.kvadratiskt()) # ger utskriften: False
```

I en subklass kan man också lägga till nya metoder. I klassen `Flervåningshus` har vi lagt till metoden `höghus` som undersöker om ett hus har tio våningar eller fler. Vi kan t.ex. skriva

---


```
if f.höghus():
        print('Högt')
```

Det går förstås inte att göra anropet `h. höghus()` eftersom objektet `h` har typen `Hus` och därför saknar metoden `höghus`.

I en subklass kan man lägga in en ny definition av en metod som annars skulle ha ärvts från superklassen. I objektorienterade sammanhang brukar man använda ordet *override* för att beskriva detta. (A *method in the subclass overrides the method in the superclass.)* Det finns inget bra ord på svenska för *override,* men det närmaste man kan komma är *överskugga.* I klassen `Flervåningshus` har vi lagt in en ny definition av metoden `yta`. Den såg ut så här:

```
def yta(self):
        return self.längd * self.bredd * self.antal
```

Lägg märke till att vi i denna metod kan komma åt de ärvda instansvariablerna `längd` och `bredd` via referensen `self`. Det hade också gått att anropa metoden `yta` i klassen `Hus` för att räkna ut bottenytan:

```
def yta(self):
        return super().yta() * self.antal
```

Metoden `yta` i klassen `Flervåningshus` överskuggar den metod som finns i klassen `Hus`. Det ser vi t.ex. om vi gör följande:

```
a = Hus(25, 10)
b = Flervåningshus(25, 10, 2)
print(a.yta()) # ger utskriften: 250
print(b.yta()) # ger utskriften: 500
```

På den nästa sista raden anropas den version av `yta` som finns i klassen `Hus` och på sista raden den version som finns i klassen `Flervåningshus`. Detta är ett exempel på en mekanism som finns i de objektorienterade språken och som kallas *dynamisk bindning.* När man anropar en metod via en referens letar Python-interpretatorn efter

metoden. Den söker då först i den understa subklassen i det objekt som referensen pekar på. Det är därför som metoden `yta` i klassen `Flervåningshus` kommer att anropas på sista raden ovan. Om metoden inte finns i den understa subklassen, söker interpretatorn i närmaste superklass. Finns metoden inte där, söker den i nästa superklass osv. Det är alltid det objekt som referensen pekar på just när anropet sker som avgör vilken metod som anropas. Det ser vi om vi skriver

---


```
b = a
print(b.yta()) # ger utskriften: 250
```

Dynamisk bindning

En metod `m` i en subklass som har samma namn som en metod i superklassen *överskuggar (overrides)* metoden i superklassen.

Om man gör anropet `r.m(argument)`, avgör klassen på det objekt som `r` refererar till vilken metod som anropas. Sökning efter första metod med namnet `m` sker nerifrån och uppåt i klasshierarkin.

I Python tillämpas dynamisk bindning även *inne* i instansmetoder när man skriver `self.m (argument).` Dynamisk bindning tillämpas då också för instansvariabler.

I en subklass kan man skriva `super() .m(argument)` för att anropa metoden `m` i superklassen.

En subklass kan i sin tur vara superklass till en subklass. Vi definierar en ny klass som beskriver hyreshus med flera lägenheter:

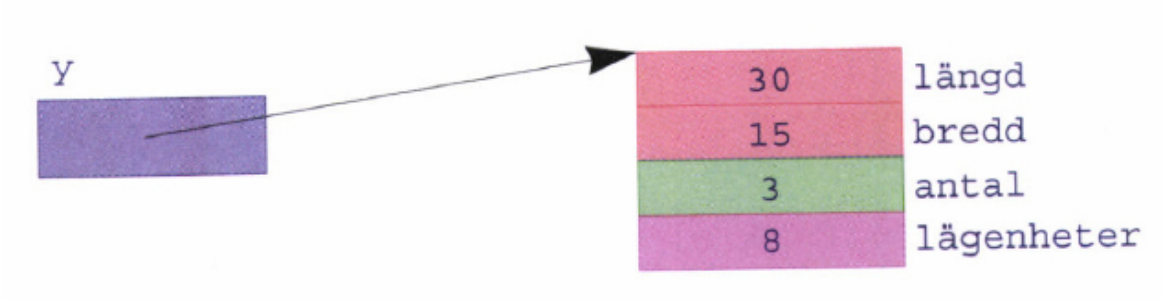
```
class Hyreshus(Flervåningshus):
        def _init_ (self, 1, b, v, n):
                super(). init (1, b, v)
                self.lägenheter = n # antalet
lägenheter

        def yta(self):
                return super().yta() * 0.95
```

I denna klass har vi lagt till en instansvariabel `lägenheter` som innehåller antalet lägenheter i huset. När man skapar ett objekt av klassen `Hyreshus` ska man ange fyra parametrar, t.ex.

```
y = Hyreshus(25, 10, 2, 8)
```

Funktionen `_init_` i klassen `Hyreshus` börjar med att anropa `_init_` i superklassen `Flervåningshus`. Den skickar då med de första tre argumenten. Funktionen `_init_` i klassen `Flervåningshus` anropar i sin tur `_init_` i klassen `Hus` med de två första argumenten. På detta sätt kommer instansvariablerna `längd, bredd` och `antal` att skapas och initieras. Instansvariabeln `lägenheter` skapas i klassen `Hyreshus`. Hur det objekt som `y` pekar på ser ut visas i figur 13.14.

Figur 13.14 Ett objekt av klassen Hyreshus.



Figuren visar variabeln y som pekar på ett objekt med fyra variabler: 30 längd, 15 bredd, 3 antal, och 8 lägenheter.

I klassen `Hyreshus` har vi definierat ytterligare en version av metoden `yta`. Denna beräkar den yta som kan användas när man ska bedöma hur mycket hyra som lägenhetsinnehavarna ska betala. I metoden har vi dragit av 5 % för trappuppgångar etc. Den nya metoden kommer att överskugga metoderna `yta` i superklasserna. Vi kan göra anropet

```
print(y.yta()) # Ger utskriften: 475
```

Uppgift 13.13

Konstruera klassen `Skola`. Låt den vara en subklass till klassen `Flervåningshus`. Låt klassen `Skola` ha en instansvariabel `antal_klassrum` som anger hur många klassrum det finns. Lägg också till en metod som beräknar det genomsnittliga antalet klassrum per våning.

Skapa sedan två objekt: ett av klassen `Flervåningshus` och ett av klassen `skola`. Försök att sätta bredden för de båda objekten till värdet 15. Försök sedan att avläsa antalet klassrum för de båda objekten. Vad händer?

Dynamisk bindning är en kraftfull mekanism som är central i de objektorienterade programmeringsspråken. Man använder ofta dynamisk bindning när man har samlingar av flera objekt. Objekten i en samling kan vara av olika klasser, men dessa klasser brukar ha en gemensam superklass.

Som exempel ska återigen klasserna `Hus, Flervåningshus` och `Hyreshus` användas. Tänk dig att man vill hålla reda på alla hus som finns i ett visst kvarter. Man kan då skapa en lista med olika slags hus:

```
kvarter = [Hus(25, 10), Flervåningshus(25, 10, 2),
        Hyreshus(25, 10, 2, 8)]
```

Anta nu att man vill skriva ut information om kvarterets alla hus. Man kan då konstruera en repetitionssats som löper igenom listan `kvarter`:

```
for e in kvarter:
        print (type (e)._name_, 'med ytan', e.yta())
```

Anropet `type (e)` ger som resultat ett objekt som innehåller information om klassen för det objektet `e` refererar till. Man kan få reda på klassens namn genom att avläsa attributet `_name_`.

Vi får utskriften:

```
Hus med ytan 250
Flervåningshus med ytan 500
Hyreshus med ytan 475.0
```

För att beräkna ytan för ett hus anropas för varje objekt metoden `yta`. Anropet sker med dynamisk bindning, vilket innebär att olika versioner av `yta` kommer att anropas för olika typer av hus.

Lägg märke till att om man använder dynamisk bindning behöver man aldrig testa vilken typ ett viss objekt har. Valet av rätt metod är ju inbyggt i mekanismen för dynamisk bindning. Detta gör att programmeringen blir både enklare och mer flexibel än om man hade lagt in tester. Det går t.ex. utmärkt att lägga till fler subklasser till klassen `Hus` utan att den `for`-sats som visats ovan behöver ändras.

Men om man vill kan man testa vilket objekt en klass tillhör. För att testa om ett objekt tillhör en klass eller någon av dess subklasser använder man en funktion som heter `isinstance`. Uttrycket

```
isinstance(y, Hus) # Ger värdet: True
```

får värdet `True` eftersom `y` tillhör en subklass till `Hus`. Vill man testa exakt vilken typ det är kan man använda operatorn `is`:

```
type(y) is Hus # Ger värdet False
type(h) is Hus # Ger värdet: True
```

Uppgift 13.14

Utöka din klass `Skola` från uppgift 13.13 med en överskuggande metod `yta`. Låt metoden beräkna ytan som antalet klassrum multiplicerat med 50. Skapa sedan en lista med ett objekt av typen `Hus` och ett av typen `Skola`. Initiera de två objekten så att de har samma längd och bredd. Initiera också variabeln `antal_klassrum` i `Skola`-objektet. Löp sedan igenom listan och anropa metoden `yta` för de objekt den innehåller. Skriv ut resultaten och studera vad som händer.

Det bör här nämnas att man i Python kan ha s.k. *multipelt arv.* Det innebär att en klass kan ha flera superklasser. Användningsområdet för multipelt arv är begränsat och mekanismerna är ganska komplexa. Det ligger därför utanför ramarna för denna bok att behandla multipelt arv.

13.15 Statisk bindning

[överkurs]

Dynamisk bindning innebär, som vi sett, att det inte bestäms förrän vid anropet av en metod vilken metod som verkligen kommer att anropas. Ett exempel på detta var metoden `yta` för klasserna som beskrev olika slags hus. Detsamma gäller när man refererar en instansvariabel. Vilken variabel man kommer åt bestäms också precis när man refererar till den. Motsatsen till dynamisk bindning är *statisk bindning.* Lite förenklat kan man säga att statisk bindning innebär att det i förväg är bestämt vilken metod som kommer att anropas eller vilken variabel man kommer att nå. Det spelar då alltså ingen roll exakt vilken typ det objekt som referensen pekar på har.

Name mangling (att använda namn som börjar med två understrykningstecken) som vi beskrev i avsnitt 13.9 kan också användas för att åstadkomma statisk bindning. Vi kan då att undvika oönskade effekter av Pythons sätt att implementera dynamisk bindning vid arv. Vi ska visa två exempel som illustrerar detta.

Som första exempel ska vi diskutera klassen `Hyreshus` på sidan 273. Den hade en instansvariabel med namnet `lägenheter`. Anta nu att vi i stället hade gett denna variabel namnet `antal`:

```
class Hyreshus(Flervåningshus):
        def _init_ (self, 1, b, v, n):
                super(). _init_ (l, b, v)
                self.antal = n # antalet lägenheter

        def yta(self):
                return super().yta() * 0.95
```

Detta är tillåtet, men problemet är att superklassen `Flervåningshus` också har en instansvariabel med namnet `antal`. Anta nu att vi skapar ett `Hyreshus` med tre våningar och 20 lägenheter och försöker beräkna dess yta:

```
a = Hyreshus(20, 10, 3, 12)print ('Ytan är', a.yta())
```

Ytan ska beräknas som 95 % av den yta som beräknas i superklassen `Flervåningshus`. Där beräknas den som längden gånger bredden gånger antalet våningar., vilket borde bli 600. Multiplicerar vi detta med 0.95 får vi 570. Resultatet borde alltså bli 570, men vi får utskriften:

```
Ytan är 2280.0
```

Vad hände? Svaret hittar vi i metoden `yta` i klassen `Flervåningshus`. Den såg ut så här:

```
def yta(self):
```

```
        return super().yta() * self.antal
```

Först beräknas husets basyta genom att man anropar metoden `yta` i superklassen `Hus`. Denna ger som resultat längden gånger bredden. Sedan multiplicerar man med instansvariabeln `antal`. Här finns problemet! Meningen är förstås att man ska multiplicera med antalet våningar. Pythons sätt att implementera dynamisk bindning innebär emellertid att den tillämpas även *inne* i superklasser. Eftersom variabeln `self` refererar till ett objekt av typen `Hyreshus` kommer därför instansvariabeln `antal` i klassen `Hyreshus` att användas i stället för den variabel som finns i klassen `Flervåningshus`. I vårt exempel multiplicerar man alltså med 12 i stället för med 3.

Det bästa är naturligtvis att aldrig använda samma namn på instansvariabler i en subklass som i superklassen. Men när man konstruerar en subklass är det inte rimligt att man ska behöva ha full kännedom om alla variabler som finns i superklassen, dess superklasser etc. Ett sätt att lösa problemet är att använda *name mangling.* I klassen `Flervåningshus` döper vi om variabeln `antal` till `_antal`:

```
class Flervåningshus(Hus):
        def _init_ (self, l, b, v):
                super(). init (1, b)
                self._antal = v # antalet våningar

        def höghus(self) :
                return self._antal >= 10

        def yta(self) :
                return super().yta() * self. antal
```

Detta garanterar att vi inne i metoden `yta` använder den egna instansvariabeln, även om man i någon subklass skulle råka definiera en instansvariabel med samma namn. Vi får en statisk bindning. Om vi nu utför satserna ovan får vi den korrekta utskriften:

```
Ytan är 570.0
```

Vårt andra exempel gäller statisk bindning vid anrop av metoder. Den egentliga anledningen till att man införde *name mangling* i Python var att man skulle få en möjlighet att hindra att en överskuggande metod anropades internt från en superklass. Anta t.ex. att vi vill lägga till metoden `_str_` i klassen `Hus` så att man kan få en textversion av ett `Hus`-objekt. (Se avsnitt 13.11.) Vi lägger till följande rader i klassen `Hus`:

```
def str (self):
        y = self.yta()
        return f'Hus med basytan {y:.lf}'
```

Vi skapar sedan ett `Hus` och ett `Flervåningshus`:

```
h = Hus(25, 10)
f = Flervåningshus(25, 10, 3)
```

Vi anropar sedan funktionen `print` för båda objekten:

```
print(h)
print(f)
```

Här kommer automatiskt metoden `_str_` att anropas. Eftersom vi inte definierat någon metod `_str_` i subklassen Flervåningshus, ärvs denna

metod från superklassen. Metoden `_str_` i klassen `Hus` kommer därför att anropas för båda objekten. Vi får utskrifterna:

```
Hus med basytan 250.0
Hus med basytan 750.0
```

Här är det något som inte stämmer. Eftersom båda husen har samma längd och bredd borde deras basyta vara densamma, men basytan för `f` har blivit tre gånger så stor. Felet ligger som du nog anar i anropet `self.yta()` i metoden `_str_`. Där kommer dynamisk bindning att tillämpas, vilket betyder att det blir metoden `yta` i klassen `Flervåningshus` som anropas i stället för motsvarande metod i klassen `Hus`. Det vill vi inte här. Vi använder därför *name mangling* och gör ett par justeringar i klassen `Hus` så att den kommer att se ut på följande sätt. Justeringarna har markerats med rött.

```
class Hus:
def _init_ (self, l, b):
self.längd = l
self.bredd = b
def kvadratiskt(self):
return self.längd == self.bredd
def yta(self):
return self.längd * self.bredd
_yta = yta # privat klassvariabel
def _str_ (self):
y = self._yta()
return f'Hus med basytan {y:.lf}'
```

Vi definierar en klassvariabel som heter `_yta`. Eftersom namnet inleds med två understrykningar blir variabeln privat i klassen. *(Name mangling* tillämpas.) Som du minns från avsnitt 8.6 kan man skapa referenser till funktioner. Variabeln `_yta` initieras så att den refererar till metoden `yta` i klassen `Hus`. Variabelns värde påverkas inte av att dynamisk bindning eventuellt sker när man anropar metoden `yta`. I metoden `_str_` anropar vi sedan metoden som beräknar ytan via referensen `_yta_` i stället för via namnet `yta`. Då får vi statisk bindning och den version av metoden `yta` som finns i klassen `Hus` kommer att anropas. Om vi nu gör samma två anrop av `print` som tidigare får vi de korrekta utskrifterna:

```
Hus med basytan 250.0
Hus med basytan 250.0
```

Vi kan lägga till en metod `_str_` också i klassen `Flervåningshus`:

```
_yta = yta

def _str_ (self):
        y = self._yta()
        return super()._str_() + \
                f' och Flervåningshus med golvytan
{y:.lf}'
```

Om vi nu skriver

```
print(f)
```

får vi utskriften

```
Hus med basytan 250.0 och Flervåningshus med golvytan
750.0
```

som säger att `f` är både ett `Hus` och ett `Flervåningshus`.

Uppgift 13.15

[överkurs]

Utöka klassen `Hyreshus` så att den innehåller en metod `_str_`. Låt metoden ha samma uppbyggnad som motsvarande metod i klassen `Flervåningshus`. Skapa sedan ett `Hyreshus`-objekt och skriv ut det.

13.16 Ett objektorienterat exempel

För att knyta ihop allt det vi diskuterat i detta kapitel kommer här ett större exempel, där vi bygger upp programmet på ett objektorienterat vis. Det är samma exempel som vi beskrev i avsnitt 9.3 på sidan 172, kortspelet Tjugoett, men denna gång ska vi använda oss av klasser och objekt. Vi börjar med att definiera en modul `kortlek2`. Du kan jämföra denna modul med den som vi visade på sidan 167.

[fullständigt program]

```
I # Modulen kortlek2
import random
```

```
class Kort:
        def _init_ (self, färg, valör):
                self._färg = färg
                self._valör = valör

        def get—färg(self) :
                return self._färg

        def get_valör(self) :
                return self._valör
        ftab = '\N{BLACK club suit}\n(white diamond
SUIT}'\ '\n{white heart suit}\n{black spade suit}'
        vtab = ('E', '2', '3', '4', '5', '6', '7',
'8', '9', '10', 'Kn', 'D', 'K')

        def str (self):
                return Kort.ftab [self._färg-l] + ' '
+\
                        Kort.vtab [self._valör-1]
```

```
class Kortlek:
        def _init_ (self): # skapar en ny, blandad
kortlek
                self._lek = []
                for i in range(1, 5) :
                        for j in ranged, 14):

self._lek.append(Kort(i, j))
                random.shuffle(self,_lek) # blanda
leken

        def ge(self): # ger det översta kortet
                if len(self._lek) > 0 ::
                        return self._lek.pop()
                else :
```

```
        return None
```

Modulen `kortlek2` innehåller två klasser: `Kort` och `Kortlek`. Klassen `Kort` beskriver ett enstaka spelkort. Den har två instansvariabler som beskriver kortets färg och valör. Eftersom det inte är meningen att man ska kunna ändra ett enstaka kort, har dessa instansvariabler markerats som privata genom att namnen inleds med ett understreck. Färgen och valören ska initieras när man skapar ett kort och ska sedan inte ändras. Vi har definierat get-metoder så att färgen och valören kan avläsas.

I klassen `Kort` har metoden `_str_` definierats. Denna ger som resultat en text som beskriver kortet. Om man anropar funktionen `print` för ett kort kommer denna text att skrivas ut. Vi har använt samma teknik för att bilda texten som vi gjorde i funktionen `visa` på sidan 167, fast tabellerna `ftab` och `vtab` har nu blivit klassvariabler i klassen `Kort`.

Den andra klassen i modulen, `Kortlek`, beskriver en hel kortlek. I funktionen `_init_` skapas en lista med namnet `_lek` som innehåller alla 52 korten i en kortlek. Anropet av `shuffle` blandar korten i listan på ett slumpmässigt. sätt. I klassen finns metoden `ge` som delar ut det översta kortet i kortleken. Klasserna `Kort` och `Kortlek` är generella och kan användas i vilket program som helst där man konstruerar olika kortspel.

Vår andra modul är speciell för kortspelet Tjugoett som vi diskuterade på sidan 172. Modulen innehåller tre klasser: `Spelare` som beskriver en generell `Tjugoett`-spelare, `Användare` som beskriver en mänsklig Tjugoett-spelare och `Dator` som beskriver en dator som spelar Tjugoett. Dessutom innehåller modulen sist en main-del med de satser som skapar objekten och kör spelet. Kommentarer följer efter programtexten.

[fullständigt program]

```
# Modulen tjugoett2
import kortlek

class Spelare():
        def init (self, leken, namn ='Spelaren'):
                self._lek = leken # referens till
kortleken
                self._namn = namn
                self._hand = []
                self.__poäng = 0
                self._antal_ess = 0

        def nytt_kort(self):
                k = self._lek.ge() # dra ett nytt kort
                self._hand.append(k) # lägg till på
handen
                if k.get_valör() == 1: # Ess?
                        self._poäng += 14 # Räkna ess som
14
                        self._antal_ess += 1
                else:
                        self._poäng += k.get_valör()
                if self._poäng > 22 and self._antal_ess > 0:
                        self._poäng -= 13 # Räkna Ess som 1
                        self._antal_ess -= 1

        def korten(self):
                return ', '.join([str(k) for k in
self._hand])

        def utskrift (self) :
                print(self._namn, 'har:',
self.korten(),
                        'och', self._poäng, 'poäng')

class Användare(Spelare):
        def spela(self):
                while True:
                        self.nytt_kort()
                        self.utskrift()
                        if self._poäng >= 21\
                                or not fortsätta('Ett
```

```
kort till'):
                                    break
                return self._poäng
```



```
class Dator(Spelare) :
        def spela(self) :
                while self._poäng <= 16:
                        self.nytt_kort()
                self.utskrift()
                return self._poäng

        def fortsätta(fråga): # Ställer en fråga
                s = input(fråga + '? ')
                return len(s) > 0 and (s[0] =='j' or s[0] =='J')

        # Här börjar exekveringen
        print('Välkommen til Tjugoett')
        while True:
                lek = kortlek.Kortlek()
                du = Användare(lek, 'Du')
                datorn = Dator(lek, 'Datorn')
                p1 = du.spela() # Låt användaren spela
                if p1 > 21:
                        print('Du förlorade')
                elif p1 == 21:
                        print('Du vann')
                else:
                        p2 = datorn.spela() # Låt datorn
spela
                        if p2 <= 21 and p2 >= p1:
                                print('Du förlorade')
                        else :
                                print('Du vann')
                if not fortsätta('Nytt parti'):
                        break
```

När man kör programmet fungerar det på samma sätt som i exemplen på sidan 173, fast utskrifterna blir lite annorlunda. Det kan se ut så här:

```
Du har: ♣ 10 och 10 poäng
Ett kort till? j
Du har: ♣ 10, ♠ 8 och 18 poäng
Ett kort till? n
Datorn har: ♣ 9, ♠ 3, ♠ Kn och 23 poäng
Du vann
```

Låt oss först diskutera main-delen av programmet. När programmet startar skapas först en kortlek:

```
lek = kortlek.Kortlek()
```

Variabeln `lek` kommer då att innehålla en referens till en ny blandad kortlek. Denna referens ger vi sedan som parameter när vi skapar de två spelarna. Detta betyder att de båda spelarna kommer att dra kort från samma kortlek.

```
du = Användare(lek, 'Du')
datorn = Dator(lek, 'Datorn')
```

Sedan anropas metoden `spela` för objektet `du`:

```
p1 = du.spela()
```

Denna metod genomför en omgång för den aktuella spelaren och returnerar de poäng spelaren fick. Om spelaren fick mer en 21 poäng, har han eller hon förlorat. Om spelaren fick färre än 21 poäng, ska datorn också spela. Detta görs i anropet

```
p2 = datorn.spela()
```

Sedan jämförs de två spelarnas poäng och man avgör vem som vunnit.

För att förstå hur main-delen av programmet fungerar behöver man inte veta hur klasserna `Kortlek`, `Användare` och `Dator` ser ut internt. Det räcker att veta hur man skapar objekt av klasserna och att det finns en metod som heter `spela`. Detta är ett exempel på en av styrkorna med objektorienterad programmering; man kan dela upp sina program i separata delar som har ett minimalt och rent gränssnitt mot varandra.

Låt oss nu studera klasserna `Användare` och `Dator`. De är båda subklasser till en superklass `Spelare`. Eftersom ingen av subklasserna innehåller någon egen konstruktör kommer `_init_` i klassen `Spelare` att anropas när man skapar objekt av klasserna `Användare` och `Dator`. Där initieras en instansvariabel `_lek` så att den innehåller referensen till den gemensamma kortleken och `_namn` så att den innehåller det namn som ges som parameter. I klassen `Spelare` finns tre variabler till: `_hand` som är en lista med de kort spelaren har för tillfället, `_poäng` som anger hur många poäng korten är värda och `_antal_ess` som innehåller det antal ess spelaren har fått.

När en spelare ska genomföra en omgång anropas metoden `spela`. Denna metod finns definierad i subklasserna. Varje subklass har sin egen version av den. Detta innebär att dynamisk bindning kommer att ske när metoden `spela` anropas. Eftersom referensen `du` pekar på ett objekt av klassen `Användare` kommer metoden `spela` i klassen

`Användare` att anropas när man skriver `du. spela ()` och metoden `spela` i klassen `Dator` att anropas när man skriver `datorn.spela`. I båda versionerna av metoden `spela` finns en while-sats där spelaren drar ett kort i taget genom att anropa metoden `nytt_kort` i superklassen `Spelare`. Det som skiljer de två versionerna av metoden `spela` åt, är villkoret som avgör när man inte ska dra fler kort.

I klassen `Användare` drar man nya kort tills poängen överskrider 20 eller spelaren själv väljer att avsluta. Efter varje nytt kort anropas metoden `utskrift` som visar vilka kort användaren har och hur många poäng dessa är värda. I klassen `Dator` drar man nya kort så länge kortsumman är 16 eller mindre. I klassen `Dator` skriver man inte ut information om vilka kort och poäng spelaren fått förrän alla korten dragits.

Metoden `nytt_kort` i superklassen `Spelare` är alltså gemensam för båda typerna av spelare. Där dras det översta kortet från kortleken och kortet läggs i listan `_hand`. Sedan adderas poängen på det nya kortet till de poäng spelaren redan har. Om poängen överskrider 21 och spelaren har något ess, räknas esset valör om från 14 till 1.

Metoden `korten` i klassen `Spelare` ger en text som visar de kort som finns i listan `_hand`. Deluttrycket

```
[str(k) for k in self._hand]
```

bildar en lista där varje element är en text. (För varje kort anropas metoden _str_ i klassen Kort automatiskt.) Denna lista ger man sedan som argument till funktionen `join` som slår ihop texterna till en enda text med kommatecken mellan varje kort.

Metoden `utskrift` i klassen `Spelare` skriver ut korten och vilken poäng spelaren har.

Uppgift 13.16

Ändra i main-delen av programmet så att två mänskliga spelare kan spela mot varandra.

13.17 Sammanfattning

Efter att ha läst detta kapitel bör du:

- förstå skillnaden mellan en klass och ett objekt,
- förstå skillnaden mellan en instansvariabel och en klassvariabel,
- förstå skillnaden mellan en instansmetod och en klassmetod,
- kunna definiera klasser i Python,
- kunna skapa objekt och hantera referenser till objekt,
- kunna initiera instansvariabler och definiera instansmetoder,
- kunna anropa instansmetoder och komma åt instansvariabler,
- veta vad som menas med inkapsling och hur man kan markera att attribut ska vara privata,
- (om du läst avsnitt 13.9) veta hur man kan använda *name mangling* för att göra attribut mer privata,

- (om du läst avsnitt 13.10) veta vad egenskaper *(properties)* är och hur man definierar och använder dem,
- kunna skapa klassvariabler och definiera klassmetoder,
- kunna anropa klassmetoder och komma åt klassvariabler,
- kunna skapa subklasser och initiera deras instansvariabler,
- kunna skapa överskuggande metoder,
- förstå dynamisk bindning,
- (om du läst avsnitt 13.15) förstå hur kan använda *name mangling* för att åstadkomma statisk bindning.

13.18 Övningar

De flesta av dessa uppgifter går ut på att skriva en klass. För att kunna testa dina lösningar måste du skriva testprogram, trots att det inte alltid står något om det i övningarna.

13.1 Definiera en klass `Bok` som beskriver en bok. Varje bok har en titel, en författare, ett sidantal och ett pris. Skriv sedan ett program i vilket du skapar två objekt av klassen `Bok`. Det ena objektet ska initieras så att det beskriver denna bok och det andra så att det beskriver någon annan av dina böcker. Låt slutligen programmet skriva ut informationen i `Bok`-objekten.

13.2 Vid navigering anger man positioner genom att ange latitud och longitud. Latituden anger var man befinner sig i nord/syd-riktning och longituden var man befinner sig i öst/väst-riktning. Latituden kan vara

antingen nord (N) eller syd (S) och anges i grader mellan 0 och 90. Longituden kan vara antingen Ö eller V och anges i grader mellan 0 och 180. Både latituden och longituden kan anges i delar av grader. Traditionellt anges därför latituden och longituden som grader, minuter och sekunder. (Det går 60 minuter på en grad och 60 sekunder på en minut.)
Ett exempel: Göteborg/Landvetter flygplats ligger på latitud 57°39'47" N, longitud 12°16'58" Ö.
Skriv en definition av en klass som beskriver positioner. Skriv sedan ett program som skapar två positioner: en för Göteborg/Landvetter flygplats och en annan för någon annan plats som du väljer själv.
Tips: Använd instansvariabler av typen `bool` för att hålla reda på om det är nord eller syd resp. öst eller väst.

13.3 Definiera en klass `Punkt` som anger en punkt i ett tvådimensionellt koordinatsystem. Låt klassen innehålla en metod som anger hur långt punkten ligger från origo. *Tips:* Avståndet från origo till en punkt *(x, y)* kan beräknas med formeln $d = \sqrt{x^2 + y^2}$. Lägg också till de två metoderna `flytta_horisontellt` och `flytta_vertikalt` som flyttar den aktuella punkten. Metoderna ska ha en parameter som anger hur långt punkten ska flyttas. Ett positivt värde anger flyttning år höger resp. uppåt och ett negativt värde flyttning åt vänster resp, neråt.

13.4 Konstruera en klass `Räknare` som beskriver heltal som bara får anta värden inom ett visst intervall. Funktionen `_init_` ska ha två parametrar som anger det minsta och det största värdet räknaren får anta. Det ska också finnas metoderna `öka` och `minska`, vilka räknar upp resp. ner en räknare med ett. Om räknaren

då får ett otillåtet värde, ska en felsignal genereras. Markera att klassens instansvariabler ska vara privata.

13.5 Skriv en klass `c` som innehåller en klassvariabel `total_antal` som automatiskt håller reda på hur många objekt av klassen `C` som har skapats. Klassen ska dessutom innehålla en klassmetod som returnerar värdet av klassvariabeln. *Tips:* Initiera variabeln `total_antal` till noll från början. Se sedan till att du i metoden `_init_` ökar variabeln `total_antal` med ett varje gång ett nytt objekt initieras.

13.6 Utöka föregående uppgift så att varje objekt av klassen `c` har en instansvariabel som automatiskt tilldelas ett unikt id-nummer. *(Tips:* Avläs klassvariabeln `total_antal` i konstruktorn.) Det ska finnas en metod som gör det möjligt att avläsa id-numret, men man ska inte kunna ändra det.

13.7 Olika sorters djur kan beskrivas med hjälp av klasser och arv. Definiera några olika djur. Utgå från en superklass `Djur` och låt de olika djuren vara subklasser till denna. Använd gärna mellanliggande klasser som t.ex. `Däggdjur, Kräldjur, Fågel` etc. Låt klassen `Djur` ha en metod `läte` som beskriver hur ett djur låter. Implementera denna metod i de olika subklasserna.
Deklarera sedan en lista med djur och skriv satser för att löpa igenom listan och låta alla djur i den ge ifrån sig ett läte.

13.8 I objektorienterade sammanhang är det vanligt med exempel som beskriver geometriska figurer. Använd klassen `Punkt` i övning 13.3. Deklarera sedan en klass `Figur` som innehåller en startpunkt. Definiera därefter ett antal subklasser till klassen `Figur`, t.ex. `Cirkel, Triangel` och `Rektangel`. Ge dem lämpliga instansvariabler. Låt varje subklass ha en egen version av en metod `area` som beräknar figurens area.
Deklarera slutligen en lista som beskriver en samling figurer och skriv de satser som behövs för att beräkna och skriva ut arean för samtliga figurer i samlingen.

1 Egentligen kan den heta vad som helst. I andra språk brukar den heta `this`. Men det är lämpligt att man följer konventionerna i Python och använder namnet `self`.

Gå tillbaka till notreferensen.

Sakregister

Sidnummer som är skrivna med fet stil hänvisar till faktarutor.

A

B

C

D

E

F

H

I

J

K

M

N

O

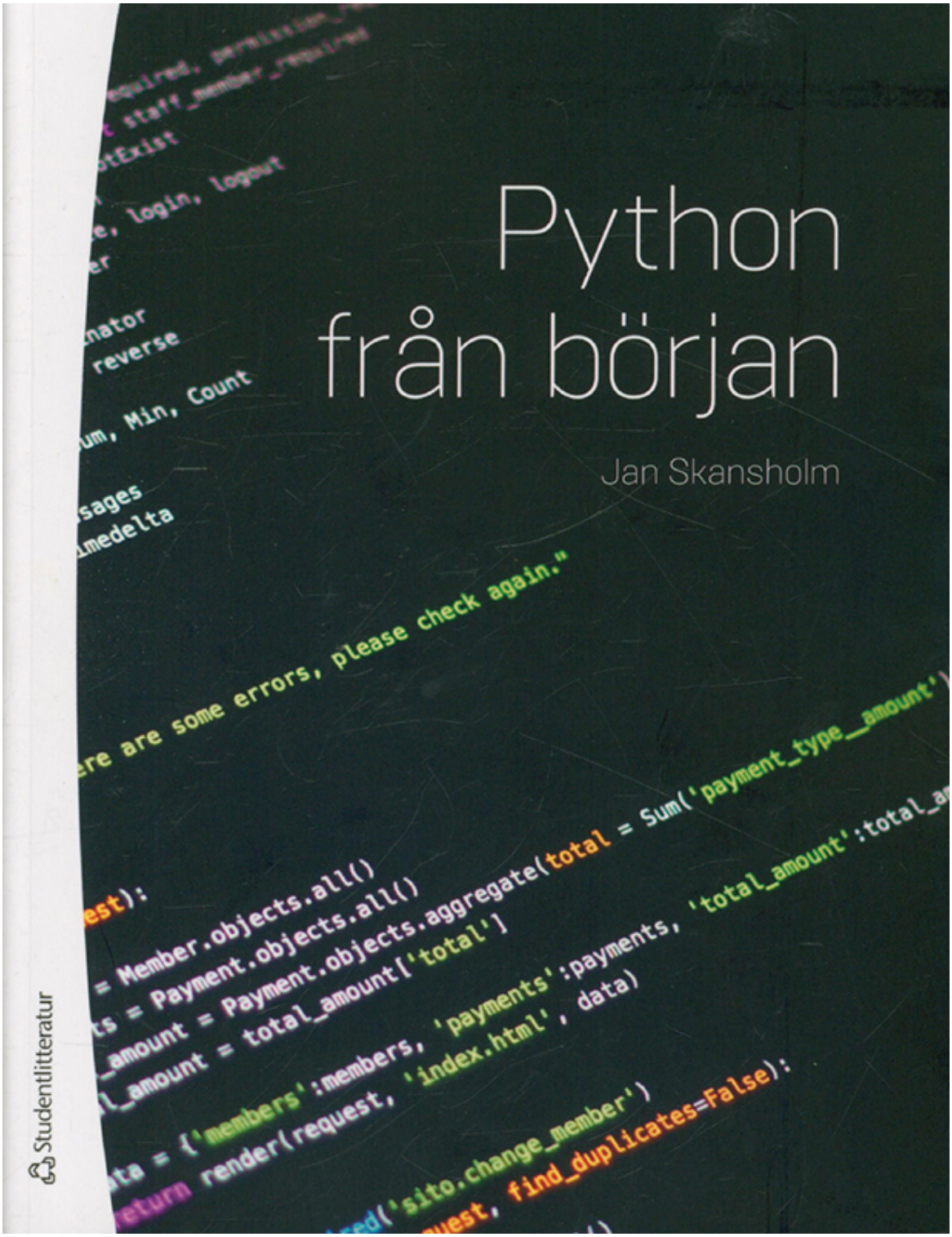
P

R

S

T

U



Jan Skansholm *är tekn.dr och har under lång tid varit verksam som universitetslektor vid Chalmers tekniska högskola och Göteborgs*

universitet. Han har mångårig erfarenhet av programmeringsundervisning och är författare till ett stort antal böcker om programmering för universitetet och gymnasiet samt för grundskolans högstadium.

Python från början

Denna bok är tänkt att passa som kurslitteratur i grundläggande programmeringskurser, till exempel i en första kurs på universitet eller högskolan. På gymnasiet kan den användas i kurserna *Programmering 1* och *Programmering 2* och i de matematikkurser som innehåller programmering som ett moment.

Boken passar också för var och en som på egen hand vill lära sig grunderna i programmering, eller för den som redan kan programmera i något annat språk och vill lära sig Python.

Python från början

- kräver inga förkunskaper
- är lättläst och pedagogisk
- behandlar i stort sett alla konstruktioner i språket i Python
- beskriver de viktigaste av de moduler som ingår i standarddistributionen
- lär ut programmering på ett praktiskt sätt, med en mängd exempel och övningar
- ger en stabil grund för att gå vidare med mer avancerad programmering, till exempel utveckling av program för matematiska och vetenskapliga beräkningar eller program för webbapplikationer
- går igenom funktioner, moduler och paket
- beskriver hur man kan använda sig av texter, listor, mängder och avbildningstabeller

- behandlar objektorienterad programmering med klasser, objekt och arv
- visar hur man kan läsa och skriva data till och från filer
- behandlar algoritmer
- diskuterar felhantering.

På bokens webbsida som nås via studentlitteratur.se/40543 finns kompletterande material, till exempel lösningar till alla övningar.