

Copyright © 2018 Krister Trangius och Emil Hall

UTGIVEN AV THELIN FÖRLAG

Samtliga varumärken som förekommer i boken tillhör innehavaren av varumärket. Observera att all kopiering eller annat mångfaldigande av denna bok eller delar av den är förbjudet enligt lag.

Thelin Förlag, Lidköping

Tel. 0510-66100, www.thelinforlag.se

utskriven 2018-05-20

Beställningsnummer J200 4941

Tryckeri: JustNu

ISBN: 978-91-7379-391-9

Omslagsfoto: Mikael Carlsson

Förord

Hösten 2018 kom programmering in som en del i kurserna Ma1c, Ma2c och Ma3c. Att använda programmering i matematiken kan vara till stor hjälp. Med hjälp av programmering kan vi visualisera sådant som annars ofta upplevs som abstrakt och svårt att få grepp om. Det är också möjligt att göra många beräkningar (och göra dem om och om igen med olika värden) som skulle vara mer eller mindre omöjliga med bara papper och penna.

Samtidigt är det många som har ganska begränsade (eller inga) erfarenheter av programmering när de möter det i matematiken. Syftet med den här boken är främst att du ska få öva dig i att räkna med programmering som hjälp, men vi går också igenom de viktigaste grunderna i programmering, så att du kan göra vilka matematiska beräkningar som helst.

Programmering är också roligt! Vi hoppas att den här boken ska vara till stor hjälp under din mattekurs och att du, med hjälp av att använda programmering i matematiken, kommer att få ett nytt förhållningssätt till räknande. Ett sätt som kan vara både spännande, utmanande och givande.

Vi vill tacka Mikael och Rebecka som har varit till stor hjälp vid framtagandet av denna bok. Vi vill också ge ett stort tack till vår käre förläggare Jan-Eric Thelin på Thelin Förlag.

Krister Trangius och Emil Hall, april 2018.

A	Om denna bok	v
A.1	Att använda detta läromedel	v
A.1.1	Till lärare	v
A.1.2	Konventioner som används i boken	vi
A.1.3	Övningar	vi
A.2	Del- och kapitelöversikt	vii
B	Python, Anaconda och Spyder	xi
B.1	Installera Anaconda och Spyder	xi
B.2	Spyder	xii
B.2.1	Filer i Spyder	xiii
B.3	import	xiii
B.4	Andra alternativ	xiv
B.4.1	Bara Python	xiv
B.4.2	Onlinelösningar	xiv

I Grunderna i programmering

1	Datorn som miniräknare	1
1.1	Kommentarer	1
1.2	Aritmetik: de fyra räknesätten	2
1.2.1	Decimaltal	2
1.3	Operatorer	3
1.4	Kvadratrot	5
1.5	Funktioner i programmering	5
1.6	Mer matte	6
1.6.1	Potenser	6
1.6.2	Trigonometri	7
1.6.3	Logaritmer	7
1.6.4	Avrundning	8

1.6.5	Slumptal	8
1.6.6	Förvandla negativa tal till positiva (abs)	9
1.6.7	Mer då?	9
2	Variabler	11
2.1	Att skapa en variabel	12
2.2	Tilldelningsoperatorn =	12
2.3	Att läsa en variabels innehåll	13
2.4	Styra utskrifterna med print	14
2.5	Variabeltyper och rutan "Variable explorer"	15
2.5.1	Konvertera variabler	16
2.5.2	Övningar för variabler	16
3	Listor	19
3.1	Indexering av listor	21
3.2	Listans längd	22
3.3	Listans sista element	23
3.4	Förenklat sätt att skapa listor	23
3.5	Fylla en lista med nummer i ordning	23
3.6	Söka i en lista	24
3.7	Matematiska operationer på varje element i listan	25
3.8	Övningar för listor	26
4	Grafer och diagram	27
4.1	Plot	27
4.1.1	Rita punkter och rutnät	28
4.1.2	Flera grafer på samma gång	28
4.1.3	Plotta med två listor, i två dimensioner	29
4.1.4	Plotta en funktion av x	31
4.1.5	Övningar för plot	32
4.2	Histogram	32
5	Selektion (med if)	35
5.1	Läsa in variabler	35
5.2	if-satsen	36
5.2.1	else	38
5.2.2	if-satser med mindre än-operatorn <	39
5.2.3	Input med bokstäver	39
5.3	Övningar	40

6	Iteration (med while)	41
6.1	while-loopen	41
6.2	Oändliga loopar	43
6.3	for-loopen	43
6.4	Gissa talet!	43
6.5	Övningar	45
7	Problemlösning	47
7.1	Nedbrytning och webbsökning	47
7.2	Hur ett program växer fram	49
II	Övningar och facit	
8	Övningar	57
8.1	Primtal, delbarhet och faktorisering	57
8.2	Sannolikhet och statistik	59
8.3	Numerisk lösning av linjära ekvationer	64
8.4	Andragradsekvationer	67
8.5	Potensfunktioner och exponentialfunktioner	70
8.6	Derivata	74
9	Facit och lösningsförslag	77
10	Sakregister	101



OM DENNA BOK

I detta kapitel kommer vi dels att titta på hur detta läromedel är tänkt att användas, dels kommer en kapitelöversikt där vi går igenom bokens innehåll.

A.1 Att använda detta läromedel

Detta läromedel är tänkt som ett komplement till redan befintliga matteböcker för gymnasiet som saknar programmering. Boken riktar sig främst till kurserna Ma1c, Ma2c och Ma3c men går bra att använda som komplement i övriga mattekurser också.

Boken är indelad i två delar. Del I lär ut de viktigaste grunderna i programmering och Python. Den innehåller också en del övningar. Del II innehåller bara övningar som är riktade till specifika mattekurser. De övningarna är ofta mer komplexa än de i del I.

A.1.1 Till lärare

Skolverkets tanke med programmering i matematiken är att det ska vara ett hjälpsamt verktyg som *stödjer* matematikundervisningen. Det finns många intressanta matematiska problem som lämpar sig väldigt väl att lösa med hjälp av programmering medan andra problem lämpar sig bättre att lösa med ”traditionella” metoder (läs: papper och Penna) eller andra digitala verktyg.

En svårighet idag är att många elever inte behärskar programmering när de behöver arbeta med det i matematiken. Därför har vi i denna bok valt att ha en del som ändå ger en introduktion till programmering. Den delen kan med fördel behandlas i eller tillsammans med Programmering 1-kursen. Delen kan också vara till hjälp, även om eleverna redan är bekanta med programmering

men tidigare inte stött på just Python. Vi författare har försökt hålla denna introduktion till ett absolut minimum - ett minimum som ändå gör att man kan arbeta med programmering för att lösa meningsfulla uppgifter i matematiken.

A.1.2 Konventioner som används i boken

Låt oss nu titta på vissa konventioner som används i boken och vad de betyder.

Pronomen i boken (vi och du)

Genomgående i denna bok används orden *vi* och *oss* - det som syftas då är vi (författarna) och du läsaren. När det står *du* eller *dig*, åsyftas dig, läsaren.

Nya termer och namn

När nya termer och namn på olika saker dyker upp för första gången är de oftast *kursiverade*. Då det inte är självklart vad som är en "term" så gäller detta inte alltid. Ibland har vi författare också funnit det lämpligt att kursivera en term mer än bara första gången.

Källkod

Stora delar av boken innehåller källkod (programmeringskod) som exempel. Det anges på följande sätt:

Exempel A.1: Exempel på källkod

```
1 temperature = float(input("Ange temperatur: "))
2 if temperature == 100:
3     print("Nu kokar vattnet!")
4 else:
5     print("Vattnet är inte exakt 100 grader...")
```

Ett problem med att ange källkod i en bok, är att källkoden inte alltid rymms på bredden. Detta har vi författare försökt lösa genom att formatera koden för att passa boksidan så gott det går. Ibland bryts dock en kodrad upp på två rader, vilket tydliggörs då den andra raden har ett indrag och saknar eget radnummer:

```
1 hist(temperaturPerDag , arange(min(temperaturPerDag)-0.5,
2                                     max(temperaturPerDag)+1.5))
```

Källkod i löptext

När källkod skrivs som en del av löptext, kan det se ut på följande sätt: Anropa metoden `print("något skrivas ut")`.

A.1.3 Övningar

I boken finns det övningar som är indelade i tre olika svårighetsgrader. Svårighetsgrad anges med färg och bokstav (E)nkel, (M)edel och (S)vår:

Övning (E), A.1: Enkel övning

Detta är ett exempel på hur en enkel övning ser ut.

Övning (M), A.2: Medelsvår övning

Detta är ett exempel på hur en medelsvår övning ser ut.

Övning (S), A.3: Svår övning

Detta är ett exempel på hur en svår övning ser ut.

Tekniska detaljer

På olika ställen i boken finns tekniska detaljer i en ruta. Dessa detaljer kan vara intressanta för dem som vill, men är inget man måste kunna eller förstå för att hålla på med programmering i matematiken.

Länkar

Det finns mängder av information på nätet. I denna bok har vi författare därför valt att ha länkar till platser där läsaren kan fördjupa sig i vissa aspekter av det som tas upp. Länkar anges i en sån här ruta.

Och här är ett exempel på en länk: <http://www.trangius.se>

A.2 Del- och kapitelöversikt

Här kommer en kapitelöversikt:

Kapitel B - Python, Anaconda och Spyder

Vi går igenom hur du kommer igång med de verktyg/program som du behöver för att programmera matematik i Python.

Del I - Grunderna i programmering

I den här delen lär vi oss det viktigaste om programmering för att kunna arbeta med det i matematiken.

Kapitel 1 - Datorn som miniräknare

Vi går igenom hur man använder Python som en miniräknare. Detta är en viktig grund innan vi går vidare till grafritande räknare och programmering.

Kapitel 2 - Variabler

Vi går igenom hur variabler funkar i programmering.

Kapitel 3 - Listor

Vi går igenom hur listor funkar i programmering.

Kapitel 4 - Grafer och diagram

Vi går igenom hur man ritar grafer och histogram.

Kapitel 5 - Selektion (med if)

Vi går igenom hur man kan göra val mellan olika alternativ, beroende på sådant som värden på olika tal och variabler.

Kapitel 6 - Iteration (med while)

Vi går igenom hur man kan köra vissa kodstycken om och om igen, utifrån önskade villkor.

Kapitel 7 - Problemlösning

Vi tar två exempel på hur programmerare tänker när de bygger upp ett längre program, från början till slut.

Del II - Övningar och facit

I denna del finns övningar med facit för kurserna Ma1c, Ma2c och Ma3c. Många av övningarna lämpar sig för flera av de olika mattekurserna. Här kommer en lista över vilka underrubriker i *kapitel 8 - Övningar* som är relaterade till det centrala innehållet i de olika kursplanerna:

Ma1c

- [...]begreppen primtal och delbarhet.
 - ▷ [8.1 - Primtal, delbarhet och faktorisering](#)
- Algebraiska och grafiska metoder för att lösa linjära ekvationer och olikheter samt potensekvationer[...]
 - ▷ [8.3 - Numerisk lösning av linjära ekvationer](#)
- [...M]etoder för beräkning av sannolikheter vid slumpförsök i flera steg med exempel från spel[...].
 - ▷ [8.2 - Sannolikhet och statistik](#)
- Begreppen förändringsfaktor och index. Metoder för beräkning av räntor och amorteringar för olika typer av lån [...]
 - ▷ [8.5 - Potensfunktioner och exponentialfunktioner](#)
- egenskaper hos [...] exponentialfunktioner.
 - ▷ [8.5 - Potensfunktioner och exponentialfunktioner](#)

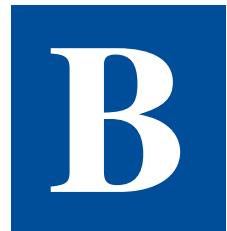
Ma2c

- Statistiska metoder för rapportering av observationer och mätdata från undersökningar.
 - ▷ [8.2 - Sannolikhet och statistik](#)
- Metoder för beräkning av olika lägesmått och spridningsmått inklusive standardavvikelse, med digitala verktyg.
 - ▷ [8.2 - Sannolikhet och statistik](#)
- Konstruktion av grafer till funktioner [...] med digitala verktyg.
 - ▷ [8.2 - Sannolikhet och statistik](#)
 - ▷ [8.5 - Potensfunktioner och exponentialfunktioner](#)
 - ▷ [8.4 - Andragradsekvationer](#)

- Egenskaper hos andragradsekvationer
 - ▷ 8.3 - Numerisk lösning av linjära ekvationer [som förkunskapskrav till andragradsekvationer].
 - ▷ 8.4 - Andragradsekvationer
- Algebraiska och grafiska metoder för att lösa exponential-, andragrads- och rotekvationer [...]
 - ▷ 8.4 - Andragradsekvationer
 - ▷ 8.5 - Potensfunktioner och exponentialfunktioner

Ma3c

- Algebraiska och grafiska metoder för lösning av extremvärdesproblem.
 - ▷ 8.4 - Andragradsekvationer
- Orientering när det gäller kontinuerlig och diskret funktion samt begreppet gränsvärde.
 - ▷ 8.6 - Derivata
- Begreppen [...] ändringskvot och derivata för en funktion.
 - ▷ 8.6 - Derivata
- Algebraiska och grafiska metoder för bestämning av derivatans värde för en funktion.
 - ▷ 8.6 - Derivata
- Introduktion av talet e och dess egenskaper.
 - ▷ 8.3 - Numerisk lösning av linjära ekvationer [som förkunskapskrav till talet e].
 - ▷ 8.6 - Derivata



P Y T H O N , A N A C O N D A O C H S P Y D E R

I det här kapitlet ska vi gå igenom hur man kommer igång med Python. Det finns flera olika sätt att arbeta med Python och det sätt som vi rekommenderar i denna bok är att använda ett program som kallas *Spyder*. Spyder är (enkelt förklarat) speciellt utvecklat för att arbeta med matematik i Python - precis vad vi behöver.

B.1 **Installera Anaconda och Spyder**

Det enklaste sättet att komma igång med Python och Spyder är genom att installera ett annat program som heter Anaconda. Anaconda innehåller flera olika viktiga python-komponenter som vi behöver. Anaconda finns för Windows, OS X och Linux.

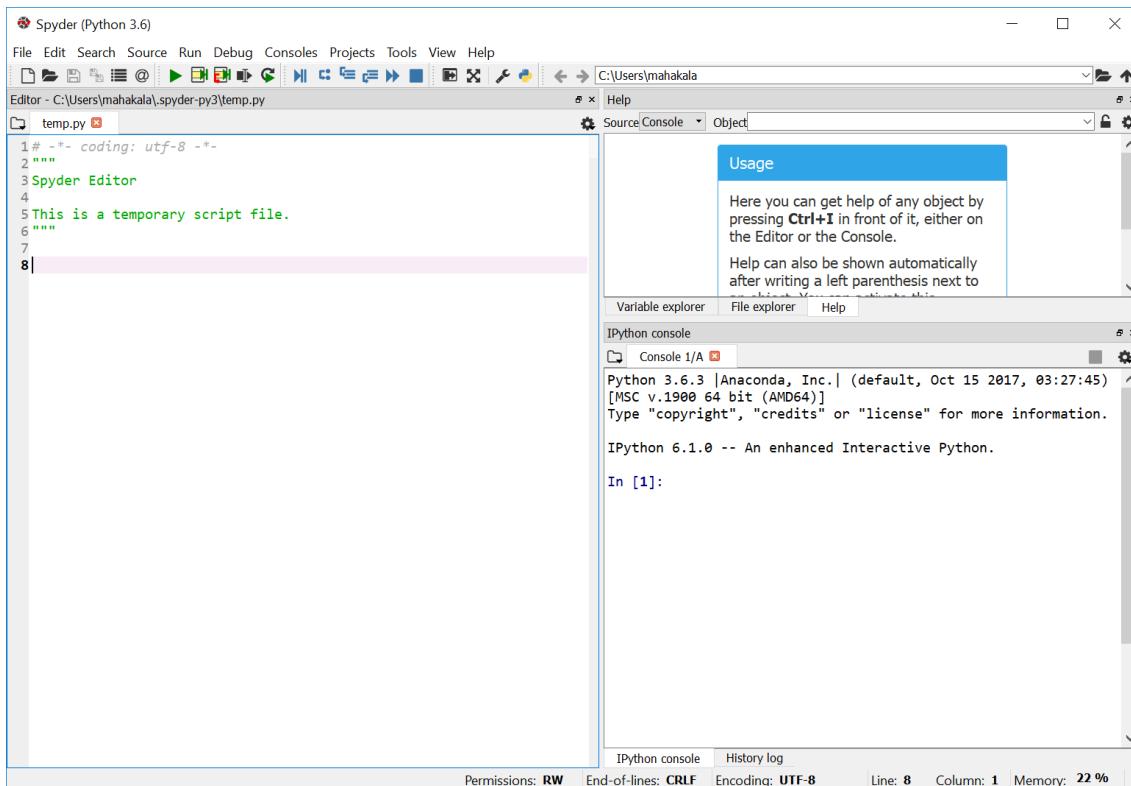
Du hittar Anaconda här: <https://www.anaconda.com/download/>

Installationen av Anaconda kan ta ganska lång tid men när det väl är installerat så ska du också hitta programmet Spyder installerat. Starta det!

B.2 Spyder

När du först startar Spyder så ser det ut ungefär såhär:

Figur B.1: Programmet Spyder



Som du ser finns det ett antal olika rutor. Den som kommer vara mest intressant för oss i början av boken är rutan ”Console” som ligger till höger. Eftersom man kan ha flera sådana rutor öppna samtidigt, heter den första ”Console 1/A”. I den finns en del text som berättar vilken version av Python etc. vi använder oss av. Där finns också följande rad:

Exempel B.1: Tom kommando-rad

In [1]:

Klicka i den rutan. Då hamnar en blinkande markör där, så du kan skriva text. Testa att skriva in texten $1+1$, så att det ser ut såhär:

Exempel B.2: Skrivit in lite matte

In [1]: $1+1$

och tryck sedan på Enter-tangenten. Vad tror du kommer hända?

Exempel B.3: Hurra, datorn kan räkna!

In [1]: $1+1$
Out[1]: 2

Du kan testa att skriva in några olika beräkningar där. Därefter kan du trycka på uppåtpilen på tangentbordet; på så vis kan vi köra gamla kommandon igen, utan att behöva skriva in dem på nytt.

B.2.1 Filer i Spyder

I början av boken kommer vi bara behöva använda ”Console”, men i kapitel 4 behöver vi börja arbeta med filer, för att kunna skriva längre kodstycken.

Till vänster i Spyder finns en ruta med en öppen fil. Första gången man startar programmet heter filen *temp.py*. Det är i vanlig ordning möjligt att skapa nya filer och spara via File uppe i menyn.

Det är viss skillnad på vilken kod ”Console” accepterar och vilken kod man kan skriva in i filen. För att kunna se vad $1+1$ blir i filen, behöver vi använda `print` (vi kommer att lära oss mer om `print` längre fram i boken):

Exempel B.4: print

```
1 | print(1+1)
```

Överst finns en grön play-knapp som du nu kan trycka på för att se resultatet i rutan ”Console”. Det är också möjligt att trycka F5 på tangentbordet.

B.3 import

För att all kod i den här boken ska fungera, så måste vi skriva följande:

Exempel B.5: import för matematikräkning

```
1 | from pylab import *
```

Gör detta *varje* gång du startar programmet och ska använda ”Console” eller längst upp i filen. Det är viktigt att du kommer ihåg detta!

Programmeringsspråket Python innehåller viss basfunktionalitet redan från början. Därutöver har många smarta människor skrivit kod för annan funktionalitet också, och paketerat ihop den till så kallade *bibliotek* som vi kan *importera*. När vi ska arbeta med matematik i Python så skriver vi `from pylab import *` för att kunna använda funktionalitet som är gjord just för matematik. Denna kodrad importerar egentligen tre bibliotek som heter `numpy`, `scipy` och `matplotlib`. Sök på nätet om dem om du är nyfiken!

B.4 Andra alternativ

B.4.1 Bara Python

Det finns möjlighet att installera Python på datorn utan Anaconda och Spyder. Hur man gör det beror på vilket operativsystem man kör och vilken funktionalitet man behöver. Om man gör det, så behöver man installera bibliotek på egen hand, vilket kan vara krångligt. Därför rekommenderar vi Anaconda och Spyder.

Du kan läsa mer om Python och hur man installerar här: <https://www.python.org>

B.4.2 Onlinelösningar

Det finns flera olika onlinelösningar med webbsidor som man enkelt kan surfa in på och börja koda direkt. Samtliga onlinelösningar har stöd för själva språket Python, men de skiljer sig åt i vilka olika bibliotek som stödjs. Eftersom man inte kan påverka vilka bibliotek en onlinelösning har, så finns det stora fördelar med att köra Python lokalt på sin egen dator. Därför rekommenderar vi Anaconda och Spyder som från början har stöd för de bibliotek som krävs i denna bok.

Trinket

En av de bästa onlinelösningarna heter Trinket och finns på <https://trinket.io/>. Det går inte att fullt ut följa denna bok med Trinket, men för att få så mycket funktionalitet som möjligt kan vi importera dessa bibliotek istället för `from pylab import *`:

Exempel B.6: import på trinket.io

```
1 from math import * # för olika matematiska funktioner
2 from random import * # för att skapa slumptal
3 from numpy import * # för arrayer
4 from matplotlib.pyplot import * # för plot
```

Exempel på funktioner som saknas eller har ett annat beteende i Trinket jämfört med Spyder: `randint`, `grid`, `append`, `argwhere`.

I

Grunderna i programmering

```
22
23
24
25
26 # print the result
27 for i in xrange(1, limit)
28     print i, ", isPrime[i]"
29
30 # factor a number
31 number = 10010
32 maybeFactor = 2
33 factors = []
34 print number, ' =',
35 while number > 1:
36     # is there no remainder after division?
37     # in other words, is "maybeFactor" a factor?
38     if number % maybeFactor == 0:
39         # maybeFactor är en primtalsfaktor
40         factors.append(maybeFactor)
41         number = number / maybeFactor
42     else:
43         maybeFactor = maybeFactor + 1
44 print "*".join(str(x) for x in factors)
45 # todo: multiply all factors, doublecheck that they
46 #       add up to the original number
47
48
```


1

DATORNS SOM MINIRÄKNARE

I det här kapitlet kommer vi att lära oss att använda Python som en miniräknare. Detta är en viktig grund innan vi går vidare till att använda Python för att rita grafer och programmera. All kod i detta kapitlet körs i ”Console”.

1.1 Kommentarer

När vi skriver in tal och matteberäkningar i Python så utför datorn dessa beräkningar. Men ibland kan det vara användbart för vår egen skull att skriva in text som datorn *inte* bryr sig om. Som minnesanteckningar till oss själva. Om vi skriver en så kallad fyrkant/brädgård/hashtagg, # så kommer datorn att strunta i resten av raden, vad det än står där. Detta kallas inom programmering för *kommentarer*:

Exempel 1.1: Vår första kommentar

```
1 In: 1+1 # allt efter fyrtanten är bara en kommentar
2 Out: 2
```

Våra kodexempel i boken kommer häданefter ofta att innehålla små kommentarer som förklrar detaljer i koden, när vi inte skriver förklaringarna i brödtexten ovanför eller nedanför kodexemplet.

1.2 Aritmetik: de fyra räknesätten

I föregående kapitel så testade vi kommandofönstret i Python genom att skriva $1+1$. Låt oss testa de fyra räknesätten. Vi börjar med bara heltal (så tar vi decimaltal senare):

Exempel 1.2: De fyra räknesätten

```

1 In: 1-1
2 Out: 0
3 In: 1+5-3
4 Out: 3
5 In: 122-300
6 Out: -178
7 In: 3*5
8 Out: 15
9 In: 20+2*7
10 Out: 34
11 In: (20+2)*7
12 Out: 154
13 In: 21/7
14 Out: 3.0
15 In: 10+6/2
16 Out: 13.0
17 In: (10+6)/2
18 Out: 8.0

```

Som du kanske ser ovan, så gäller de vanliga reglerna för de fyra räknesätten även i Python; Multiplikation och division sker före addition och subtraktion och det är, i vanlig ordning, möjligt att styra detta med parenteser. Som du också ser är multiplikationstecknet en asterisk $*$.

1.2.1 Decimaltal

Vad händer om vi gör en division som inte går jämnt upp? I Python får vi faktiskt decimaltal av alla divisioner, oavsett om de går jämnt upp eller ej - det är därför vi får svaret 3.0 istället för bara 3.

Exempel 1.3: Decimaltal

```

1 In: 17 / 2
2 Out: 8.5

```

Jo, vi får ett decimaltal med en punkt mellan heltalsdelen och decimalerna. Svensk standard är att skriva kommatecken där, men Python följer engelsk standard där man använder punkt istället. Om vi själva vill skriva in ett decimaltal så måste vi också använda punkt:

Exempel 1.4: Decimaltal skrivs med punkt

```

1 In: 8.5 * 2
2 Out: 17.0

```

Eftersom kommatecken har andra betydelser inom programmering så kommer Python att missförstå och ger oss ett helt annat resultat än vi ville:

Exempel 1.5: Varning för kommatecken

```
1 In: 8,5 * 2  
2 Out: (8, 10)
```

Övning (E), 1.1: De fyra räknesätten

Testa de fyra räknesätten i Python med olika siffror, både heltal och decimaltal.

1.3 Operatorer

Inom programmering talar man om något som kallas *operatorer*. Vi har redan använt några operatorer, nämligen de fyra räknesätten (+, -, *, /) men det finns fler som du redan känner till från matematiken (och några som du kanske inte känner igen).

I den här boken kommer vi att arbeta med lite olika operatorer och du kommer att lära dig dem allt eftersom. Några operatorer som vi kan titta på redan nu, är de så kallade *jämförelseoperatorerna*:

En jämförelseoperator används för att jämföra två tal. Vi kan se det som att vi frågar datorn om jämförelsen stämmer (t.ex. ifall ett tal är mindre än ett annat) och datorn svarar ja eller nej. Låt oss testa:

Exempel 1.6: Mindre än-operatorn

```
1 In: 3 < 17 # är 3 mindre än 17?  
2 Out: True
```

Det stämmer ju att 3 är mindre än 17. Som du ser, så svarar datorn True. Det är datorns sätt att säga ”ja”. Om det inte hade stämt, hade datorn svarat False:

Exempel 1.7: Mindre än-operatorn igen

```
1 In: 17 < 3 # är 17 mindre än 3?  
2 Out: False
```

Här kan du se de jämförelseoperatorer som finns i Python:

Tabell 1.1: Jämförelseoperatorer

Tecken	Betydelse
<code>==</code>	lika med
<code><</code>	mindre än
<code>></code>	mer än
<code><=</code>	mindre än eller lika med
<code>>=</code>	mer än eller lika med
<code>!=</code>	inte lika med

Notera att jämförelseoperatorn `==` består av två lika med-tecken på rad. Det kanske verkar märkligt, men blir begripligt senare, i delkapitel 2.2 där vi talar om tilldelningsoperatorn som skrivs med endast ett lika med-tecken.

Låt oss testa några av dessa:

Exempel 1.8: Test av jämförelseoperatorer

```

1 In: 3 > 17 # är 3 mer än 17?
2 Out: False
3 In: 5 >= 5 # är 5 mer än eller lika med 5?
4 Out: True
5 In: 8 == 8 # är 8 lika med 8?
6 Out: True

```

Du kanske undrar vad det här ska vara bra för. Är det inte självklart att 3 är mindre än 17? Varför ska vi fråga datorn om det? Jämförelseoperatorer används oftast tillsammans med selektion (som vi går igenom i kapitel 5) och iteration (som vi går igenom i kapitel 6).

Det finns också fler operatorer i Python än de vi går igenom i den här boken. Se här för en lista: https://www.quackit.com/python/reference/python_3_operators.cfm

Övning (E), 1.2: Jämförelseoperatorer

Testa själv med samtliga jämförelseoperatorer och lite olika tal till höger och vänster!

1.4 Kvadratrot

Kom ihåg: vi måste köra `from pylab import *` innan vi använder matematiska funktioner i Spyder. Utan importen kommer `sqrt` och många andra funktioner inte att funka.

Vi antar att du redan känner till begreppet ”roten ur”, eller *kvadratrot* som det också kallas. Med papper och penna brukar vi skriva till exempel $\sqrt{9} = 3$. Men det finns ingen tangent på datorns tangentbord för att skriva ett sådant rottecken. I Python räknar vi istället ut kvadratrötter med ordet `sqrt` - förkortning av engelskans *square root*. Sen direkt efter ordet `sqrt` ska vårt tal stå inom parentes:

Exempel 1.9: Kvadratrot, roten ur 9

```
1 In: sqrt(9)
2 Out: 3.0
```

1.5 Funktioner i programmering

Begreppet *funktion* är kanske något du känner igen från din vanliga mattebok? T.ex. kanske du har hört att y är en funktion av x , alltså: $y = f(x)$. Här händer något med x inne i funktionen f och y har värdet av $f(x)$. Inom programmering används det begreppet lite annorlunda - varje funktion har ett visst namn. Vi kan se det som att namnet är en förkortning som representerar ett längre stycke kod.

Det finns en massa inbyggda funktioner i Python och i Pythons olika matematiska bibliotek. Vi har redan lärt oss en av dem, nämligen `sqrt`, och vi ska strax lära oss några till. Det går även att skapa egna funktioner, men det kommer vi inte lära oss i denna bok. Att skapa egna funktioner gör programmerare nämligen mest för att strukturera stora program, och vi kommer inte att skapa så stora program.

Om du inte riktigt förstår allt detta så är det ingen fara - du kommer kunna använda funktioner ändå.

För att använda en funktion skriver vi alltid dess namn, sedan en inledande parentes, sedan så kallade *argument*, och sist en avslutande parentes. Till exempel:

Exempel 1.10: Repetition av funktionsanvändning

```
1 In: sqrt(81)
2 Out: 9.0
```

Vissa funktioner har fler än ett argument. Här är exempel på funktioner som tar två argument. Argumenten står inom paranteserna och separeras med kommatecknen:

Exempel 1.11: Funktioner med två argument

```

1 In: min(3.5, 17) # ger det minsta av två tal
2 Out: 3.5
3 In: max(3.5, 17) # ger det största av två tal
4 Out: 17
5 In: fmod(26, 10) # ger rest efter division
6 Out: 6

```

Nu förstår du kanske varför det inte går så bra att använda kommatecknen för decimaltal?

Notera att ett argument i sin tur kan vara ett resultat av en uträkning, t.ex.:

Exempel 1.12: Resultat av uträkning som argument

```

1 In: min(sqrt(81), 8+2)
2 Out: 9.0

```

Och det går förstås att göra hur långa kedjor som helst:

Exempel 1.13: Människor från ytter rymden

```

1 In: min(sqrt(sqrt(81)*3*3), sqrt((8+2)*22))
2 Out: 9.0

```

Övning (E), 1.3: Testa funktioner

Testa att använda alla de ovanstående funktionerna med några olika argument. Testa även att använda funktioner som argument till andra funktioner.

1.6 Mer matte

Det finns förstås många fler funktioner inbyggda i Python. Vi kommer nu att lista andra operatorer och funktioner som brukar användas i mattekurserna Ma1c, Ma2c och Ma3c. Vi kommer också titta på ett par konstanter.

1.6.1 Potenser

I den rena matematiken brukar vi skriva potenser ("upphöjt till") med små siffror, till exempel 3^2 . Åter igen finns det inget sätt på datorns tangentbord att skriva sådana små siffror. Istället använder vi två multiplikationstecken på rad:

Exempel 1.14: Potenser

```

1 In: 3 ** 2
2 Out: 9

```

1.6.2 Trigonometri

Det speciella talet pi finns inbyggt i Python. Det finns inget specialtecken π utan vi skriver helt enkelt:

Exempel 1.15: pi

```
1 In: pi
2 Out: 3.141592653589793
```

De trigonometriska funktionerna sinus, cosinus, tangens och deras släktingar, finns också inbyggda. Låt oss testa sinus-funktionen:

Exempel 1.16: Trigonometri

```
1 In: sin(pi/4)
2 Out: 0.7071067811865475
```

I följande tabell kan bokstaven v mellan parenteserna ersättas med valfritt tal:

Tabell 1.2: Trigonometriska funktioner

Matteformel	Funktion i Python
$\sin v$	$\sin(v)$
$\cos v$	$\cos(v)$
$\tan v$	$\tan(v)$
$\sin^{-1} v$	$\arcsin(v)$
$\cos^{-1} v$	$\arccos(v)$
$\tan^{-1} v$	$\arctan(v)$

Argumentet som vi skickar in i de trigonometriska funktionerna \sin , \cos och \tan är en vinkel. Men den funkar kanske inte som du förväntar dig? Vinkeln mäts nämligen i *radianer*, inte i *grader*. Det går 360 grader på ett varv, men det går $2\pi \approx 6.28$ radianer på ett varv.

För att omvandla mellan grader och radianer kan vi använda funktionerna `deg2rad` (*from degrees to radians*), respektive `rad2deg` (*from radians to degrees*). T.ex: `sin(deg2rad(30))` eller `rad2deg(arcsin(0.5))`.

Övning (E), 1.4: Trigonometriska funktioner

Testa själv med samtliga trigonometriska funktioner och lite olika tal mellan parenteserna!

1.6.3 Logaritmer

Exempel 1.17: Logaritmer

```

1 In: e
2 Out: 2.718281828459045
3 In: log10(10**3)
4 Out: 3.0
5 In: log(e**3)
6 Out: 3.0

```

1.6.4 Avrundning

Som vanligt ska vårt tal stå inom parentes.

Exempel 1.18: Avrundning

```

1 In: round(3.6) # avrundar till närmaste heltal
2 Out: 4
3 In: floor(3.6) # avrundar nedåt
4 Out: 3.0
5 In: floor(-3.6) # nedåt även för negativa tal. ej mot noll
6 Out: -4.0
7 In: ceil(3.2) # avrundar uppåt
8 Out: 4.0

```

1.6.5 Slumptal

Det är möjligt att slumpha fram tal i Python. Vi kan be om att få ett slumptal från och med 0 till ett valfritt tal:

Exempel 1.19: Slumptal

```

1 In: randint(6) # ger ett slumptal fr.o.m 0 t.o.m 5
2 Out: 2
3 In: randint(6) # nästa gång kan det bli ett annat tal
4 Out: 0
5 In: randint(6)
6 Out: 5

```

Notera att om vi vill ha ett slumptal till och med 5, så skriver vi *inte* `randint(5)` utan `randint(6)`. Om du har programmerat Python tidigare så kanske detta låter fel. Saken är att det finns två olika `randint`. Det finns en inbyggd i Python i biblioteket `random`. Men eftersom vi i den här boken istället skriver `from pylab import *` så får vi den andra `randint` som beter sig annorlunda. Förvirrande? Ja, tyvärr.

Om vi vill ha ett slumptal inom ett intervall som *inte* börjar på 0, t.ex. om vi vill efterlikna ett tärningsslag mellan 1 och 6, så gör vi såhär:

```
1 randint(1, 7) # ger ett slumptal fr.o.m 1 t.o.m 6
```

Det går förstås att ha vilket intervall som helst:

```
1 randint(7, 42) # ger ett slumptal fr.o.m 7 t.o.m 41
```

1.6.6 Förvandla negativa tal till positiva (abs)

En annan funktion som kan vara användbar är `abs`, som förvandlar negativa tal till positiva:

Exempel 1.20: `abs`

```
1 In: abs(-26) # förvandlar negativa tal till positiva
2 Out: 26
```

1.6.7 Mer då?

I det här kapitlet gick vi igenom sånt som gör Python till en vanlig miniräknare. Det finns såklart sådant som gör att vi kan använda datorn som en grafräknare också. Det kommer vi att gå igenom i kapitel 4 men först ska vi lära oss några viktiga grunder i programmering.

2

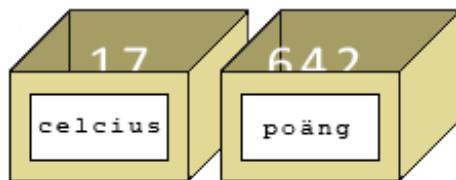
VARIABLER

I det här kapitlet ska vi gå igenom variabler och hur de funkar i programmering. Vi kommer också lära oss hur man själv kan styra utskrift på ett tydligare sätt. All kod i detta kapitlet körs i ”Console”.

Du känner kanske redan till begreppet variabler? Traditionellt inom matematiken talar man ofta om okända variabler som x och y eller a och b . Variabler i Python är något liknande, men de används lite annorlunda.

En variabel i programmering ses kanske enklast som en låda, med en etikett på. På etiketten står det ett namn och i lådan ligger det ett tal.

Figur 2.1: Variabler kan ses som lådor med etikett och innehåll



I matematikböcker är variablers namn oftast bara en bokstav långa, men i programmering brukar namnet vara ett helt ord, vilket hjälper oss att hålla reda på dem då vi har många.

En annan skillnad är att variabler i programmering alltid har ett visst värde. I traditionell matematik kan man t.ex. säga att $a^x * a^y = a^{x+y}$ som en generell regel som gäller för alla värden. I Python så arbetar man inte med generella värden på variablerna, utan de ”innehåller” alltid ett visst tal.

2.1 Att skapa en variabel

För att skapa en variabel hittar man först på ett namn till den, sen skriver man namnet, likamed-tecken, och värdet. Till exempel:

Exempel 2.1: Skapa variabeln celsius

```
In: celsius=17
```

Det är även okej att ha mellanslag om man tycker att det blir mer lättläst när det är mindre tätt:

Exempel 2.2: Skapa variabeln celsius

```
In: celsius = 17
```

Namnet kan inte innehålla åäö, mellanslag eller andra konstiga specialtecken. Om vi vill att namnet ska bestå av flera ord så kan vi som sagt inte använda mellanslag, utan istället brukar vi separera orden med hjälp av understreck; min_eigen_variabel.

2.2 Tilldelningsoperatorn =

Lika med-tecknet = som vi använde ovan kallas för *tilldelningsoperatorn*. I delkapitel 1.3 tittade vi på jämförelseoperatorn ==. Inom programmering skiljer man på *jämförelse* och *tilldelning*. Tilldelningsoperatorn skrivs med bara *ett* lika-med tecken och används, som vi såg ovan, när man vill ge en variabel ett värde. Det är mycket viktigt att man inte blandar ihop tilldelningsoperatorn = och jämförelseoperatorn ==.

Traditionellt inom matematiken har du kanske lärt dig att det inte spelar någon roll på vilken sida ”lika med”-tecknet olika tal står. När man programmerar i Python är det dock annorlunda. Variabeln, som ska få ett värde, står till vänster om tilldelningsoperatorn. Värdet som variabeln ska få, står till höger. Detta är alltså inte okej, men testa gärna själv och se vad som händer:

Exempel 2.3: Man får inte sätta variabelnamnet på fel sida

```
In: 17 = celsius # ej ok!
```

Det som ligger på högersidan om tilldelningsoperatorn händer först. Det innebär att vi kan göra beräkningar på högersidan. När det väl har räknats ut, tilldelas variabeln värdet. Vi kan t.ex. plussa ihop en massa siffror och sedan lägga det i variabeln. Här får variabeln nr värdet 110:

Exempel 2.4: Beräkningar görs på högersidan om tilldelningsoperatorn

```
In: nr = 100 + 3 + 7
```

Med andra ord sker det i följande två steg:

1. Talen 100, 3 och 7 summeras till 110.
2. Variabeln `nr` skapas och tilldelas värdet 110.

2.3 Att läsa en variabels innehåll

Efter att en variabel har skapats så kan man läsa innehållet i den och göra beräkningar med den. Om man vill se vilket värde en variabel har just nu, så kan man skriva dess namn:

Exempel 2.5: Se en variabels värde

```
1 In: nr = 100 + 3 + 7  
2 In: nr  
3 Out: 110
```

Man kan addera två variabler med varandra:

Exempel 2.6: Addera två variabler

```
1 In: nr1 = 100  
2 In: nr2 = 555  
3 In: total = nr1 + nr2  
4 In: total  
5 Out: 655
```

Men, kanske du undrar nu, sa vi inte nyss att man inte får sätta variabelnamn på högersidan? Nja, det får man visst, om de variablerna redan finns sedan tidigare och därmed faktiskt har ett värde (100 och 555 i exemplet ovan). Vad vi egentligen menade var att den variabel som vi vill tilldela ett värde måstestå på vänstersidan.

Viktigt att notera är att koden körs uppifrån och ned. Så först skapar vi två variabler (på rad 1 och rad 2). Därefter adderar vi dem (på rad 3). Det hade inte gått att göra tvärtom, att addera två variabler innan de skapats.

Efter att man har skapat och använt variabeln, kan man fortsätta att använda den. Man kan t.ex. ändra variabelns värde. I följande kodstycke ändrar vi en variabel från att ha värdet 100 till att ha värdet 555.

Exempel 2.7: Ändra värdet på en variabel

```
1 In: nr = 100  
2 In: nr  
3 Out: 100  
4 In: nr = 555  
5 In: nr  
6 Out: 555
```

Faktum är att ordet *variabel* kommer från latinets *variare*, vilket betyder ”ändra”. Jämför svenska variera!

Man kan öka en variabels värde. Det gör man genom att lägga på variabelns (tidigare) värde till sig själv och addera ett nytt värde. Detta kanske är lite förvirrande om du fortfarande tänker på tilldelningsoperatorn som ett lika med-tecken. Men det finns inget som hindrar att vi använder en variabel i en beräkning på högersidan, och skriver samma variabel på vänstersidan. På första raden i följande kodstycke får variabeln `nr` värdet 100, på andra raden får den värdet 150:

Exempel 2.8: Öka värdet på variabeln

```

1 In: nr = 100
2 In: nr = nr + 50
3 In: nr
4 Out: 150

```

Om detta vore en matematisk ekvation så vore det förstås omöjligt. Det finns ju inget tal som är lika med sig självt plus 50.

2.4 Styra utskrifterna med `print`

Hittills har vi fått ut värdet på variabler genom att bara skriva deras namn. Vi kommer dock att behöva ha mer kontroll över vår utskrift framöver. Då kan vi använda funktionen `print` som funkar såhär:

Exempel 2.9: Skriv ut på kommando

```

1 In: nr1 = 100
2 In: nr2 = 555
3 # vi kan skriva ut text med citattecken ,
4 # så kallade dubbelfnuttar:
5 In: print("nu ska vi räkna")
nu ska vi räkna
6 In: print(nr1) # vi kan skriva ut en variabels värde ,
7 100
8 In: print(nr1+nr2+12) # ...och resultatet av en uträkning
9 667
10

```

Det är också möjligt att använda två eller fler argument med `print`. De skrivs då ut på samma rad, med mellanslag emellan.

Exempel 2.10: Skriv ut på kommando

```

1 In: nr1 = 100
2 In: print("Värdet är:", nr1)
3 Värdet är 100

```

2.5 Variabeltyper och rutan ”Variable explorer”

Hittills har vi bara arbetat med heltalsvariabler. Det går såklart att arbeta med decimaltal också. Faktum är att Python gör skillnad på dessa typer av variabler. Varför? Förklaringen är teknisk och ligger utanför ramen för denna bok.

För att se detta tydligt, låt oss titta på en till av rutorna i Spyder. De variabler vi skapar går att se i den lilla rutan ”Variable explorer” som ligger till höger.

Figur 2.2: Rutan ”Variable explorer” efter att vi skapat två variabler

Name	Type	Size	Value
celsius	int	1	17
nr	int	1	110

I rutan finner vi dels kolumnerna ”Name” och ”Value”. De kan vara till hjälp när vi behöver dubbelkolla värdet på variabler vi har skapat. Kolumnen ”type” berättar för oss vilken typ som våra variabler har. De ord Python använder är *int* (från engelskans *integer*) för heltal och *float* för decimaltal. Kolumnen ”size” berättar hur mycket plats variabeln tar i datorns minne.

Du kommer att kunna se variabler av andra typer än *float* och *int*. Det finns många andra variabeltyper i Python (ett exempel är typen *bool* som bara kan ha ett av två värden, `True` eller `False`). I den här boken räcker det dock med att du kommer ihåg *float* och *int*.

När man startar Spyder så är rutan oftast helt tomt. När vi skriver `from pylab import *` fylls dock rutan på med en del variabler som kan göra rutan mer överskådlig. Dessa variabler finns för att kunna göra vissa viktiga matematiska beräkningar. T.ex. kan du se värdet på variablerna `e` och `pi` i rutan.

2.5.1 Konvertera variabler

Oftast behöver vi inte tänka på vilken typ en viss variabel har. Det finns dock tillfället då vi måste konvertera från decimaltal till heltal (eller tvärtom). Då gör vi såhär:

Exempel 2.11: Konvertera variabler

```

1 In: x = 1.7 # skapa en variabel av typen float, decimaltal
2 In: x = int(x) # konvertera den till int, heltal
3 In: print(x)
4 Out: 1
5 In: x = float(x) # konvertera tillbaka den till float
6 In: print(x)
7 Out: 1.0

```

Testa denna kod och se hur kolumnen "type" i rutan "Variable explorer" ändras. Notera att decimaltalen avrundas om vi konverterar dem till heltal (nedåt om talet är positivt och uppåt om talet är negativt).

Vi kommer att behöva konvertera till heltal i Övning 8.11.

2.5.2 Övningar för variabler

Övning (E), 2.1: Höger eller vänster?

Om du kör dessa tre rader kod:

```

1 a = 1
2 b = 2
3 a = b

```

Vad är nu värdet på a och b? Fundera själv innan du testar och läser facilit.

Övning (E), 2.2: Kopplas variabler ihop för all framtid?

Om du kör dessa tre rader kod:

```

1 a = 1
2 b = a
3 a = 2

```

Vad är nu värdet på b? Fundera själv innan du testar och läser facilit.

Övning (E), 2.3: Summan och medelvärdet av tre tal

Låt säga att du redan har dessa tre variabler inmatade i Python:

```
1 | a = 23  
2 | b = 45  
3 | c = 67
```

Skriv en rad kod som beräknar summan av dessa variabler och skriver ut summan.

Skriv en till rad kod som skriver ut medelvärdet. Kom ihåg att det är datorn som ska göra uträkningen, inte du!

Övning (E), 2.4: Decimaltal till heltal

Låt säga att du redan har denna variabel inmatad i Python:

```
1 | a = 11.534
```

Skriv en rad kod som tar denna variabel, omvandlar decimaltalet till närmsta heltal, och skriver ut det.

3

L I S T O R

I det här kapitlet ska vi gå igenom listor och se hur de funkar i programmering. Man kan se listor som en speciell typ av variabler (som vi lärde oss i föregående kapitel). En lista kan, till skillnad från en vanlig variabel, innehålla flera värden som ligger på rad. Listor är användbara då man vill hålla reda på många saker samtidigt.

Den formella beteckningen för en lista är *array* eller *vektor*. Termen vektor kan vara lite förvirrande, då den används på olika sätt i olika sammanhang. Ibland syftar man på en riktning eller position i en rymd (då har den x-, y- och kanske z-koordinater). Termen array används bara på engelska men inte på svenska. I denna bok kommer vi att använda termen lista.

Ett konkret exempel på en lista kan vara temperaturer olika dagar. Den första dagen var det 15 grader, den andra 13, den tredje 16 osv. Varje temperatur ligger var för sig lagrad i något som kallas *element*. Vi kan komma åt varje element i en lista med något som kallas för *index*.

Här är en illustration av en lista som innehåller fem olika element. Vi kan se varje elements index. Notera att det första elementet har index 0:

Figur 3.1: En lista

<i>index</i>	0	1	2	3	4
	17	-65	-20	9	42

Att skapa listor i Python är ganska enkelt, man gör bara såhär:

Exempel 3.1: Skapa lista

```

1 # Skapa listan:
2 lista = []
3
4 # Fyll på listan med olika värden med funktionen append:
5 lista.append(17)
6 lista.append(-65)
7 lista.append(-20)
8 lista.append(9)
9 lista.append(42)

```

Tyvärr behöver vi göra det lite krångligare för oss själva i denna bok. Anledningen är teknisk men beror kortfattat på att vi vill använda viss matematisk funktionalitet på våra listor, som finns i biblioteket `pylab`. Kom ihåg: För att använda denna typ av listor i Spyder behöver vi köra `from pylab import *`.

Såhär behöver vi skriva istället:

Exempel 3.2: Skapa lista

```

1 # Skapa listan:
2 temperature = array([])
3
4 # Fyll på listan med olika värden med funktionen append:
5 temperature = append(temperature, 17)
6 temperature = append(temperature, -65)
7 temperature = append(temperature, -20)
8 temperature = append(temperature, 9)
9 temperature = append(temperature, 42)
10
11 # Skriv ut listan:
12 print(temperature)

```

Vi får resultatet:

```
[17. -65. -20. 9. 42.]
```

Om du undrar varför talen får en punkt efter sig, så beror det på att de är lagrade som decimaltal, se delkapitel 2.5. Men för att spara plats så skrivs 0:an inte ut. Vissa funktioner som vi använder konverterar heltal till decimaltal, vilket ger de lite lustiga utskrifterna.

För att sedan använda listan kan vi t.ex. göra så här:

Exempel 3.3: Använda lista

```

1 # Lägg samman värdet på de olika elementen i listan och
2 # skriv ut medelvärdet:
3 summa = temperature[0] + temperature[1] + temperature[2]
   + temperature[3] + temperature[4]
4 print(summa / 5)

```

Då får vi följande utskrift:

```

1 -3.4

```

3.1 Indexering av listor

I exempel 3.3 läser vi innehållet i listans fem olika element med hjälp av deras index. Genom en siffra indexeras ett specifikt element i listan. Index är det som står inne i hakparenteserna [] efter variabelnamnet. Vi kan använda index både för att skriva och läsa ett specifikt element.

När man arbetar med listor och indexering av dem gäller det dock att se upp! Det är lätt hänt att man försöker att använda ett element som inte finns. Det kommer bli fel då man kör programmet. Här är ett exempel:

Exempel 3.4: Felaktig indexering av en lista

```

1 temperature = array([])
2
3 temperature = append(temperature, 17)
4 temperature = append(temperature, -65)
5 temperature = append(temperature, -20)
6 temperature = append(temperature, 9)
7 temperature = append(temperature, 42)
8
9 # Skriv ut elementen med index 0 och 5
10 print("Första elementet: ")
11 print(temperature[0])
12 print("Sjätte elementet: ")
13 print(temperature[5]) # fel!

```

Då får vi en utskrift som ser ut något i stil med det här:

```

1 Första elementet:
2 17.0
3 Sjätte elementet:
4 Traceback (most recent call last):
5
6   File "<ipython-input-20-0b23519ca590>", line 1, in
     <module>

```

```

7     print(temperature[5])
8
9 IndexError: index 5 is out of bounds for axis 0 with size
  5

```

Först skrivs värdet på det första elementet, `temperature[0]` ut. Därefter får vi som väntat ett felmeddelande. Felet är "*IndexError: index 5 is out of bounds for axis 0 with size 5*". Vi försöker helt enkelt indexera ett element som ligger utanför vår lista.

Det går alltså inte att läsa ett index som inte finns. Det går inte heller att skriva till ett index som inte finns. Om vi ändå skriver till ett index som redan finns så kommer det gamla värdet att ersättas, precis som det funkar för vanliga variabler.

3.2 Listans längd

En lista har ett antal element. Antalet kallas även listans längd. När vi lägger till ett element så växer listan - den blir längre. Vi kan mäta listans längd genom att skriva `.size` efter namnet på listan. Så här:

Exempel 3.5: Listans längd

```

1 temperature = array([])
2
3 # Stoppa in tre element i listan,
4 # och skriv ut listans längd efter varje steg:
5 temperature = append(temperature, 17)
6 print(temperature.size)
7 temperature = append(temperature, -65)
8 print(temperature.size)
9 temperature = append(temperature, -20)
10 print(temperature.size)

```

Vi får resultatet:

```

1 1
2 2
3 3

```

3.3 Listans sista element

Vår listas första element är alltid `temperature[0]`, men vad är dess sista element? Det sista elementets index varierar ju. Vi kan alltid komma åt det sista elementet genom att mäta längden, så här:

```
1 | temperature[temperature.size - 1]
```

Notera att vi måste skriva `-1`. En lista med t.ex. tre element har ju index `0, 1` och `2` men längden `3`.

Men eftersom det är långt och tyatigt att skriva så finns det ett kortare sätt. Man skriver bara `-1` som index för att få det sista elementet:

```
1 | temperature[-1]
```

3.4 Förenklat sätt att skapa listor

Det finns också ett enklare sätt att skapa listor på:

Exempel 3.6: Förenklat sätt att skapa listor

```
1 | temperature = array([17, -65, -20, 9, 42])
```

Eller om vi vill skapa en lista med ett visst antal element, och alla element ska ha samma värde:

Exempel 3.7: Massproduktion

```
1 | temperature = repeat(0, 5) # ger listan [0, 0, 0, 0, 0]
2 | temperature = repeat(99, 3) # ger listan [99, 99, 99]
```

Detta kan vara praktiskt om man vill skapa väldigt många element och slippa skriva in dem ett i taget. Också praktiskt om det önskade antalet element ligger i en variabel.

3.5 Fylla en lista med nummer i ordning

Ofta kommer vi vilja fylla en lista med tal på rad, t.ex. `[-2 -1 0 1 2]`. Då finns det ett koncisare sätt att skriva. Vi kan använda en funktion som heter `arange` som funkar så här:

Exempel 3.8: `arange`

```
1 | print(arange(-2, 3))
```

Vilket ger resultatet:

Exempel 3.9: `arange`

```
1 | [-2 -1 0 1 2]
```

Notera att `arange` ger oss värden upp till det andra argumentet (3) men inte *till och med* det värde. Jämför med `randint` i delkapitel 1.6.5.

Med hjälp av `arange` kan vi alltså skapa en lista så här:

Exempel 3.10: Skapa en lista som innehåller alla heltalet från -2 till 2

```
1 | x = arange(-2, 3)
```

Med hjälp av ett tredje argument till funktionen `arange` kan vi dessutom ta kortare ”steg” mellan elementen:

Exempel 3.11: Anpassa steglängd mellan element

```
1 | x = arange(-2, 3, 0.5)
2 | print(x)
```

Vi får resultatet:

```
1 | [-2. -1.5 1. -0.5 0. 0.5 1. 1.5 2. 2.5]
```

Det här blev kanske inte exakt som vi ville? Vi ville egentligen att listan skulle gå upp till och med 2. För att få listan att sluta med 2 får vi sätta 2.5 som stopp-värde:

Exempel 3.12: Anpassa steglängd mellan element

```
1 | x = arange(-2, 2.5, 0.5)
```

3.6 Söka i en lista

Vi har lärt oss att läsa elementet med ett visst index. Ibland kan vi istället vilja göra samma sak ”baklänges” och få svar på frågan: På vilket index ligger ett visst element? T.ex. vilken dag var temperaturen 9 grader? Då kan vi använda funktionen `argwhere`.

```
1 | temperature = array([17, -65, -20, 9, 42])
2 | argwhere(temperature==9)[0][0] # ger resultatet 3
```

Varför behöver vi skriva `[0][0]` efter `argwhere()`? Anledningen är mer avancerad än vad vi tar upp i denna bok, men har med flexibilitet och matriser att göra.

3.7 Matematiska operationer på varje element i listan

Vi kan enkelt göra samma matematiska operation på varje element i en lista, t.ex. multiplicera alla element med 2, eller med sig själva. Vi använder de vanliga operatorerna + - * /:

Exempel 3.13: Matematiska operationer på varje element i listan

```
1 x = array([1, 2, 3, 4])
2 y = x + 1
3 z = x * 2
4 w = x * x
5 print(y)
6 print(z)
7 print(w)
```

Vi får resultatet:

```
1 [2 3 4 5]
2 [2 4 6 8]
3 [1 4 9 16]
```

Det är också möjligt att köra funktioner på varje element i en lista.

Exempel 3.14: Köra funktioner på varje element i en lista

```
1 x = array([4, 9, 16, 25])
2 print(sqrt(x))
```

Vi får resultatet:

```
1 [2. 3. 4. 5.]
```

3.8 Övningar för listor

Övning (E), 3.1: Olika sätt att skapa samma lista

Skapa en lista som innehåller 8 element. Det första elementet ska ha värdet 20, det andra ska ha värdet 30, det tredje 40, och så vidare upp till och med 90. Hur många olika sätt kan du komma på för att skapa en sådan lista? Vilket tycker du känns lämpligast?

Övning (E), 3.2: print med lista

Tänk dig följande kod:

```
1 temperature = array([])
2 temperature = append(temperature, -14)
3 temperature = append(temperature, 22)
4 temperature = append(temperature, 7)
5 print(temperature[1])
6 print(temperature.size)
```

Utan att skriva in koden själv i Python, vad tror du att vi får för utskrift?

4

GRAFER OCH DIAGRAM

En stor anledning till att vi programmerar i Spyder är att det där är väldigt lätt att rita olika sorters grafer och diagram. I det här kapitlet kommer vi gå igenom en funktion som heter `plot` - den använder vi för att rita grafer. Vi kommer också lära oss funktionen `hist` som används för att rita ut histogram.

Hittills har vi bara skrivit in vår kod i "Console" men nu kommer våra program att växa och då blir det mycket enklare om vi använder filer. Om du inte redan är bekant med det, se kapitel B för hur du ska göra.

4.1 Plot

Tänk dig att vi med en termometer har mätt temperaturen en gång per dag, fr.o.m. måndag t.o.m. fredag, så att vi har 5 mätvärden. Om vi har våra mätvärden i en lista så kan vi ge listan till Python med kommandot `plot` och då kommer vi få upp en graf på skärmen.

Kom ihåg: För att rita grafer i Spyder behöver vi köra `from pylab import *`. För att vara säker på att grafen ska visas, använd funktionen `show`. Skriv följande i din fil och kör koden:

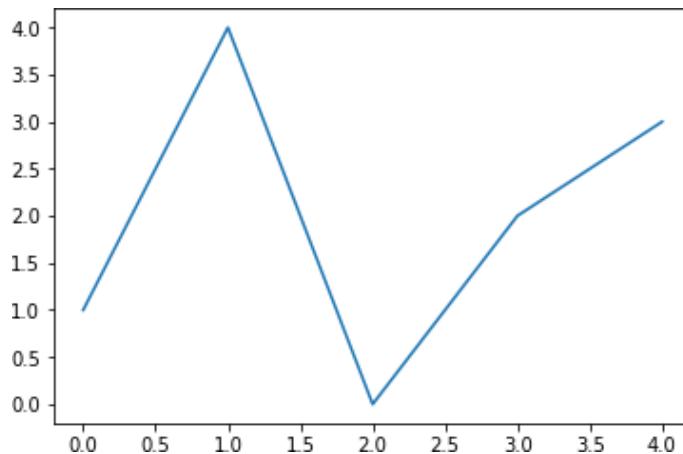
Exempel 4.1: Vår första graf

```
1 temperaturPerDag = array([1, 4, 0, 2, 3])
2 plot(temperaturPerDag)
3 show()
```

Du ser resultatet här nedanför. Notera att linjen börjar på 1 längst till vänster, sen går den upp till 4, ner till 0, och så vidare, precis i samma ordning som vår lista. Notera också att den horisontella axeln går från 0 till 4, vilket motsvarar

elementens index i vår lista. Den vertikala axeln går från 0 till 4, vilket beror på att den lägsta temperaturen i vår lista är 0 och den högsta är 4.

Figur 4.1: Vår första graf



4.1.1 Rita punkter och rutnät

För att göra grafen lite tydligare så kan vi be Python att rita små cirklar vid varje mätpunkt. Det gör vi genom att lägga till `, "-o"` innan slutparentesen efter `plot`. För att göra grafen ännu tydligare kan vi visa ett rutnät i bakgrunden, och det gör vi genom att lägga till `grid(True)` på en egen rad.

Exempel 4.2: Vår andra graf

```

1 temperaturPerDag = array([1, 4, 0, 2, 3])
2 plot(temperaturPerDag, "-o")
3 grid(True)
4 show()

```

Det finns många fler möjligheter att ställa in och anpassa hur grafer ska ut. T.ex. färger, storlek, var de vertikala och horisontella axlarna ska börja och sluta, med mera. Detta kan vara krångligt men för den som är intresserad finns det mycket att läsa om `plot` här: https://matplotlib.org/api/pyplot_api.html

4.1.2 Flera grafer på samma gång

Vad händer om vi använder `plot` flera gånger på rad, i samma fil, fast med olika listor? Jo, vi får se flera resultat i samma graf. Varje `plot`-linje får då en egen färg, för att vi lättare ska kunna se skillnad på dem:

Exempel 4.3: Två grafer i en

```

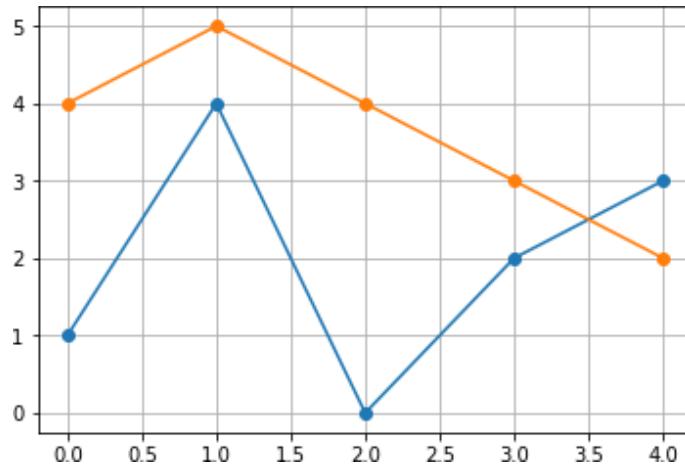
1 plot(array([1, 4, 0, 2, 3]), "-o")
2 plot(array([4, 5, 4, 3, 2]), "-o")

```

```

3 | grid(True)
4 | show()

```

Figur 4.2: Två grafer i en

4.1.3 Plotta med två listor, i två dimensioner

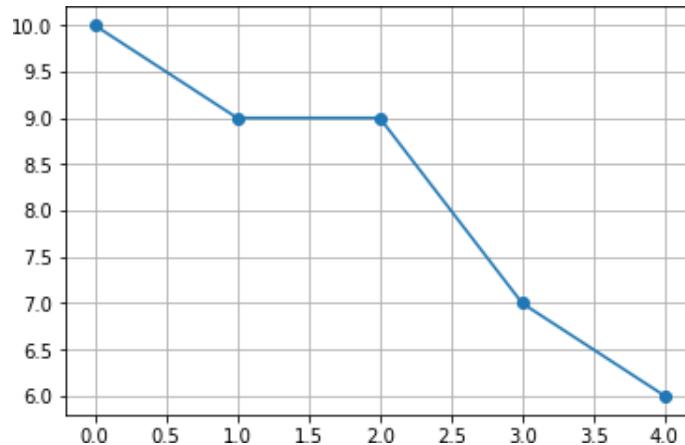
Tänk om vi hade tänkt mäta temperaturen varje dag i en vecka, men vi glömde mäta på torsdag och fredag. Hur vill vi att vår graf ska se ut då? Det vore bra om det syntes i grafen att två dagar saknas i mitten. Vi kan förstås plotta som vanligt...

Exempel 4.4: Syns inte att torsdag och fredag saknas

```

1 | temperaturPerDag = array([10, 9, 9, 7, 6])
2 | plot(temperaturPerDag, "-o")
3 | grid(True)
4 | show()

```

Figur 4.3: Syns inte att torsdag och fredag saknas

... men då ser det ut som vi mätte fem dagar på rad. Det vore bättre om vi fick ett ”glapp” i grafen, så att det inte finns någon punkt på $x = 4$ och $x = 5$,

men att det sedan finns punkter igen på $x = 6$ och $x = 7$. Kan vi åstadkomma detta i Python? Såklart vi kan! Men då måste vi använda `plot` på ett nytt sätt. Istället för att bara skicka in en lista så skickar vi in två listor:

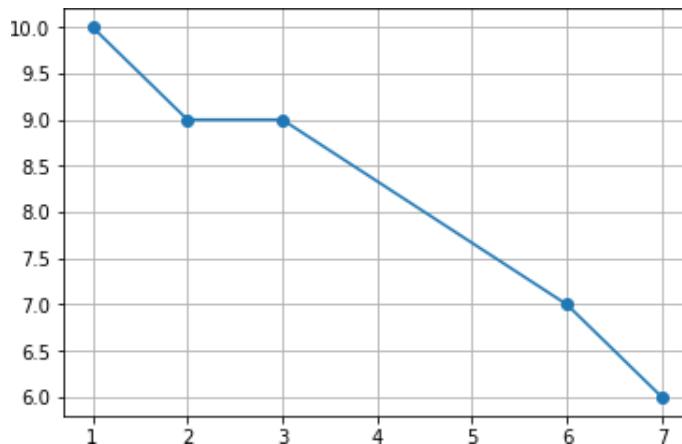
Exempel 4.5: Syns att torsdag och fredag saknas

```

1 dagar = array([1, 2, 3, 6, 7]) # vi hoppar över 4 och 5
2 temperaturPerDag = array([10, 9, 9, 7, 6])
3 plot(dagar, temperaturPerDag, "-o")
4 grid(True)
5 show()

```

Figur 4.4: Syns att torsdag och fredag saknas



Nu fick vi det glapp som vi ville ha, hurra!

Notera att vi på detta sätt kan rita vilka former som helst, även linjer som vänder tillbaka, korsar sig själv, osv:

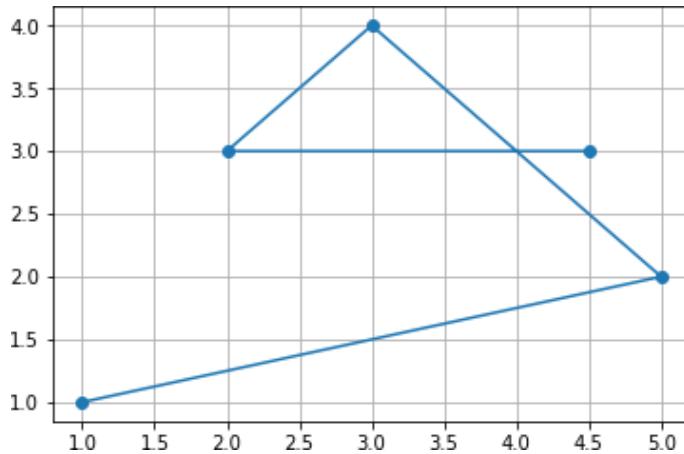
Exempel 4.6: Plotta vilken form som helst

```

1 plot(array([1, 5, 3, 2, 4.5]), array([1, 2, 4, 3, 3]),
      "-o")
2 grid(True)
3 show()

```

(Se graf på nästa sida...)

Figur 4.5: Kors och tvärs

Om vi enkelt vill rita en rät linje, så kan vi använda ovanstående metod och bara ange linjens startpunkt och slutpunkt, såhär: `plot(array([xstart, xend]), array([ystart, yend]))`. Exempel:

Exempel 4.7: Rät linje

```

1 plot(array([0, 10]), array([7, 5]))
2 show()

```

4.1.4 Plotta en funktion av x

Nu har vi lärt oss allt vi behöver för att kunna rita en graf av en funktion av x. Som vi såg i delkapitel 3.7 så är det möjligt att köra en funktion på varje element i en lista. Låt oss testa att rita ut en graf av funktionen `sqrt`:

Exempel 4.8: Plotta kvadratrotsfunktion

```

1 x = arange(0, 100) # fyll lista med x-värden att visa
2 y = sqrt(x) # kör funktionen sqrt på varje element
3 plot(x, y) # rita ut grafen!
4 show()

```

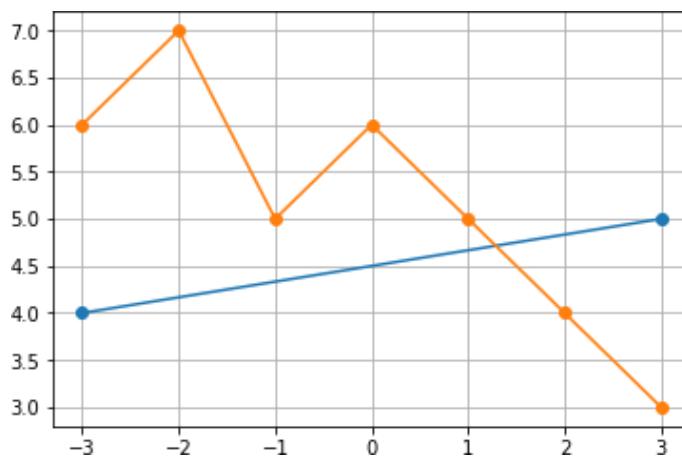
Notera att vi också använder det vi lärde oss i delkapitel 3.5 där vi fyllde en lista med alla nummer mellan -2 och 2.

4.1.5 Övningar för plot

Övning (E), 4.1: Återskapa kod efter bild

Skriv koden för att rita följande bilden:

Figur 4.6



Övning (E), 4.2: Plotta given funktion

Här är några ofullständiga rader kod:

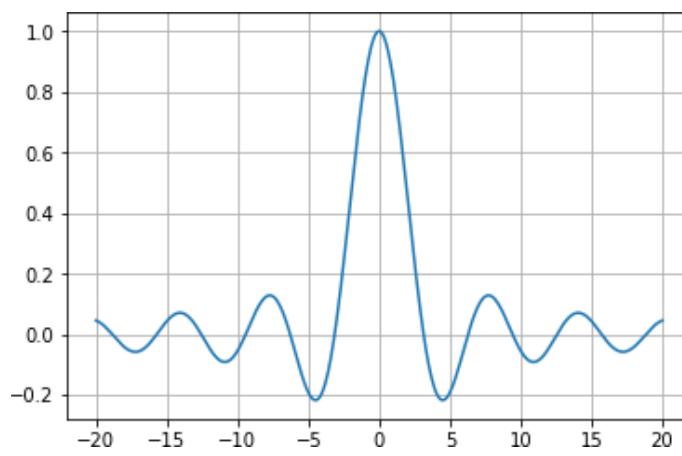
```

1 x = # här ska det stå något
2 y = sin(x) / x
3 # här ska det stå kod för att rita ut grafen

```

Ersätt kommentarerna med kod för att rita följande bild:

Figur 4.7



4.2 Histogram

Om vi istället har temperaturen från många dagar så kanske vi hellre vill se en slags sammanfattning - hur många dagar var det 3 grader varmt? Hur många

dagar var det 4 grader? Då passar det bra med ett så kallat histogram, även känt som stapeldiagram eller stolpdiagram.

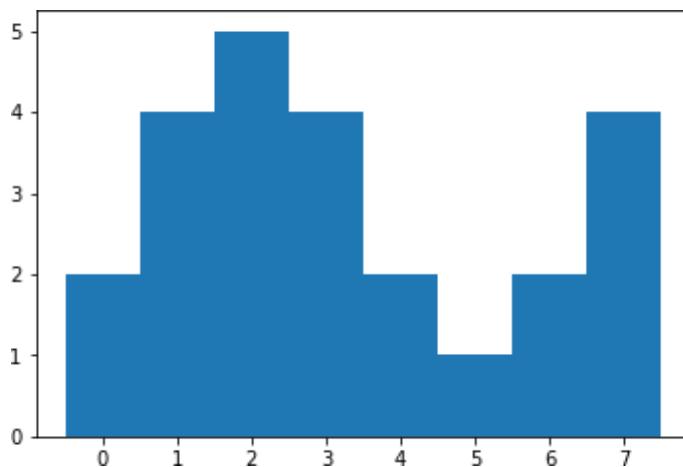
Exempel 4.9: Vårt första histogram

```

1 temperaturPerDag = array([1, 4, 0, 2, 3, 4, 3, 6, 7, 7,
2   7, 6, 5, 7, 3, 2, 3, 2, 2, 1, 1, 2, 1, 0])
3 hist(temperaturPerDag, array([-0.5, 0.5, 1.5, 2.5, 3.5,
4   4.5, 5.5, 6.5, 7.5]))
5 show()

```

Figur 4.8: Vårt första histogram



I histogrammet kan vi se att det var 3 grader varmt fyra dagar och 4 grader varmt i två dagar. Funktionen `hist` tar alltså två argument. Det första är en lista av tal att sammanfatta, som kan vara hur lång som helst. Det andra argumentet berättar hur vi vill sammanfatta listan, närmare bestämt var vi vill ha gränserna mellan staplarna. Den första stapeln har sin vänsterkant vid -0.5 och högerkant vid 0.5 , och där börjar nästa stapel, och så vidare.

När vi vill visa en sån här lista, där vi vet att den längsta temperaturen är 0 och den högsta är 7, så passar det förstås bäst att ha just de staplarna i diagrammet. Mer generellt, när vi har en lista som bara innehåller heltal (inga decimaltal), och har ett ganska litet antal olika heltal, så passar det bäst att ha en stapel för varje heltal. Vi kan automatisera det så här:

Exempel 4.10: Välja rätt antal staplar automatiskt

```

1 temperaturPerDag = array([1, 4, 0, 2, 3, 4, 3, 6, 7, 7,
2   7, 6, 5, 7, 3, 2, 3, 2, 2, 1, 1, 2, 1, 0])
3 hist(temperaturPerDag, arange(temperaturPerDag.min()-0.5,
4   temperaturPerDag.max()+1.5))
5 show()

```

Med ovanstående kod kan vi lätt lägga till nya temperaturer utan att behöva komma ihåg att ändra på `hist`-raden. Du behöver inte tänka på hur `arange`-biten

funkar om du inte vill.

Men om vi istället har en lista som innehåller väldigt många olika heltal, eller decimaltal, så passar det inte bäst med något särskilt antal staplar utan är mer av en smaksak.

5

SELEKTION (MED IF)

Ett program behöver ofta göra olika val beroende på olika värden på saker och ting (t.ex. olika variablers värden). Då använder man något som kallas för *selektion*. I det här kapitlet kommer vi att gå igenom `if`-satsen. Den utför selektion men brukar i sig kallas för *villkorssats*.

5.1 Läsa in variabler

Innan vi går vidare med selektion så ska vi ta ett kort sidospår och lära oss hur man kan läsa in variabler medan en kodsnutt körs. Det är smidigt om man vill köra samma kodsnutt flera gånger, men testa olika värden på variablerna. Hittills när vi har skrivit kod har vi ju bara gjort det för oss själva. Men det är ju möjligt att låta andra personer köra vår kod.

Vi skapar en ny fil och skriver följande kod:

Exempel 5.1: Läsa in variabler

```
1 a = float(input("Ange variabeln a: "))
2 b = float(input("Ange variabeln b: "))
3 print(a+b)
```

Här blev det mycket nytt på en gång. Vi går igenom det bit för bit.

Funktionen `input` skriver alltså ut en text på skärmen, och låter användaren (den som kör vår kod) skriva in något. När användaren tryckt på Enter-tangenten, så fortsätter vår kod köras.

En användare skulle ju kunna skriva in vad som helst, t.ex. ordet ”morot”. Det är ju inget tal! Men om användaren skriver in ett tal, så kan Python ändå inte

automatiskt anta att det är ett tal. Därför måste vi, när vi skriver programmet, använda funktionen `float` för att tala om för datorn att vi bara är intresserade av decimaltal. Om vi ändå bara vore intresserade av heltal skulle vi skriva `int` istället.

Övning (E), 5.1: input

Testa funktionen `input`. Prova lite olika utskrifter och olika namn på variabler. Testa med `int` istället för `float`.

Testa att köra exemplet ovan, men istället för att skriva in en siffra så skriver du in ditt namn. Vad får du för felmeddelande? Vad tror du det beror på?

Men nu ska vi gå vidare och lära oss selektion, där vi kommer att använda `input`.

5.2 if-satsen

En `if`-sats jämför alltså två värden med varandra. En `if`-sats på svenska blir alltså en om-sats:

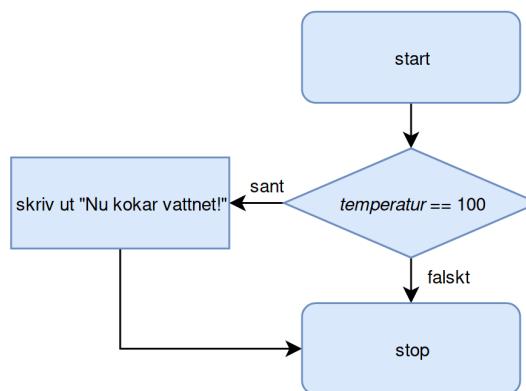
```
OM något SÅ
    gör detta.
SLUT OM
```

T.ex. så kan man kontrollera hur varmt vatten är:

```
OM temperatur är 100 SÅ
    skriv ut "Nu kokar vattnet!" på skärmen.
SLUT OM
```

Detta kan illustreras visuellt som ett flödesschema som ser ut så här:

Figur 5.1: If-sats som flödesschema



Låt oss prova detta i Python. Vi lägger till lite kod för att användaren ska få mata in värdet på variabeln temperature:

Exempel 5.2: Vår första if-sats

```
1 temperature = float(input("Ange temperatur: "))
2 if temperature == 100:
3     print("Nu kokar vattnet!")
```

Du minns väl att det är skillnad på jämförelseoperatorn och tilldelningsoperatorn? Om du inte minns, se delkapitel 1.3 och delkapitel 2.2. När vi arbetar med if-satser använder vi jämförelseoperatorn just för att jämföra två olika tal (eller i exemplet ovan, värdet av variabeln temperature och talet 100).

Lägg märke till kolonet : efter if-raden. Efter if temperature == 100: ligger den kod som vi vill utföra, ifall villkorssatsen visar sig stämma.

Raden under if-raden är indragen åt höger. Låt oss tydliggöra vad detta indrag innebär med följande exempel:

Exempel 5.3: Vår andra if-sats

```
1 temperature = float(input("Ange temperatur: "))
2 if temperature == 100:
3     print("Nu kokar vattnet!")
4     print("Kok kok kok!")
5 print("Programmet avslutas")
```

De två första raderna ligger i rak linje med varandra längst åt vänster, medan rad 3 och 4 ligger med ett indrag åt höger. *Detta indrag är av yttersta vikt!* Indraget görs med hjälp av TAB på tangentbordet. Alla indragna rader körs om (och endast om) det som står i if-satsen är sant. Om detta verkar oklart nu, så hoppas vi att det ska bli tydligare när du läst klart detta kapitel och nästföljande.

Om vi anger att vattnet är 100 grader, får vi alltså följande resultat:

```
1 Ange temperatur: 100
2 Nu kokar vattnet!
3 Kok kok kok!
4 Programmet avslutas
```

5.2.1 else

Att använda en `if`-sats utan något mer, gör att vi kör ett stycke kod om villkors-satsen visar sig stämma. Annars gör vi ingenting speciellt utan programmet fortsätter bara att köra. Låt oss fortsätta med temperaturer. Om vi i körningen av exempel 5.3 angav något annat än 100, slutar programmet abrupt:

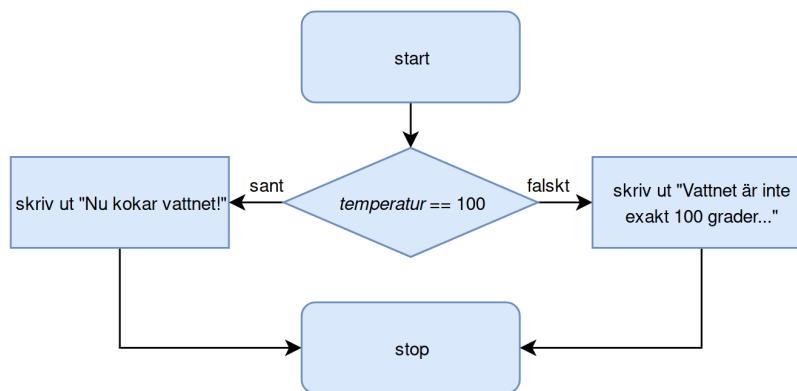
```
1 Ange temperatur: 89
2 Programmet avslutas
```

Men ofta vill man ju faktiskt göra något annat, om det visar sig att villkorssatsen *inte* stämmer. Låt oss fortsätta med det kokande vattnet, först på svenska:

```
OM temperatur är 100 SÅ
    Skriv ut "Nu kokar vattnet!" på skärmen.
ANNARS
    Skriv ut "Vattnet är inte exakt 100 grader..."
SLUT OM
```

Detta kan illustreras som flödesschema så här:

Figur 5.2: if och else som flödesschema



Låt oss programmera detta i Python:

Exempel 5.4: Vår första else-sats

```
1 temperature = float(input("Ange temperatur: "))
2 if temperature == 100:
3     print("Nu kokar vattnet!")
4 else:
5     print("Vattnet är inte exakt 100 grader...")
```

Nu får vi i alla fall ett meddelande, om vi anger att temperaturen är annat än 100 grader:

```
1 Ange temperatur: 89
2 Vattnet är inte exakt 100 grader...
```

5.2.2 if-satser med mindre än-operatorn <

Låt oss testa if-satser med en annan jämförelseoperator. Vi tar mindre än-operatorn <.

Vi tar det först på svenska:

```
OM temperatur är mindre än 100 SÅ
    Skriv ut "Vattnet är inte tillräckligt varmt än..." på
    skärmen.
ANNARS
    Skriv ut "Vattnet kokar!"
```

Kodat i Python blir det:

Exempel 5.5: Mindre än-operatorn

```
1 temperature = float(input("Ange temperatur: "))
2 if temperature < 100:
3     print("Vattnet är inte tillräckligt varmt än...")
4 else:
5     print("Vattnet kokar!")
```

På samma sätt som med operatorerna == och <, kan du använda if-satser med de övriga jämförelseoperatorerna som finns listade i delkapitel 1.3.

5.2.3 Input med bokstäver

Ibland känns det lite tråkigt att vi bara kan prata siffror med datorn. Det vore roligare att kunna säga små ord till den, i alla fall ”j” och ”n” för att symbolisera ja/nej. Vi skapar ett program som ställer frågan ”Är det fint väder?”. Om användaren svarar ”j” skriver programmet ut ”Vi går på picknick!”. Annars händer ingenting. Men hur ska datorn kunna förstå svaret ”j”? Vi kan använda följande trick:

Exempel 5.6: Kontrollera vädret

```
1 svaret = input("Är det fint väder? ") # obs, utan
   float/int
2 if svaret == "j": # observera citattecknen
3     print("Vi går på picknick!")
```

5.3 Övningar

Övning (E), 5.2: Kontrollera vädret (fortsättning)

Arbata vidare på exempel 5.6 men lägg till att användaren kan svara ”n”. Då skriver programmet ut ”Vi stannar inne och läser en bok”.

Övning (E), 5.3: Var är det kallast?

Skapa ett program där man får mata in temperaturen i Östersund och Göteborg. Programmet ska sedan berätta var det är kallast. Men om det är lika kallt i båda städerna så ska programmet berätta detta istället.

Övning (E), 5.4: Felaktig if-sats

Något stämmer inte riktigt med följande if-sats:

```
1 x = 9
2 if x = 10
3     print("den är 10!")
```

När vi försöker köra koden så får vi ett felmeddelande - vad är det som inte stämmer? Skriv om koden så att det blir rätt!

6

ITERATION (MED WHILE)

Iteration är att köra ett kodstycke om och om igen, utan att behöva skriva kodstycket flera gånger. Ofta brukar ett sånt kodstycke kallas loop. Kodstycket upprepas så länge ett visst villkor är sant. Ofta vill man upprepa något ett visst antal gånger. Om man t.ex. vill gå igenom en lista med femtio element (saker i listan) i och göra något med varje element, så loopar man ett kodstycke femtio gånger.

I Python finns det flera typer av loopar med olika syften. Den enda loop vi kommer att arbeta med i denna bok är `while`, eftersom den är enklast att förstå, och går att använda till allt.

6.1 while-loopen

Man kan beskriva `while`-loopen på svenska så här:

MEDAN någonting SÅ
 Gör detta
SLUT MEDAN

Vi kan t.ex. be någon ange temperatur. Så länge temperaturen är mindre än 100, så kör vi ett varv i loopen. För varje varv i loopen, så ökar vi temperaturen med en grad och skriver ut den. När temperaturen är 100 så går vi vidare i programmet och skriver ut ett meddelande:

(Se nästa sida...)

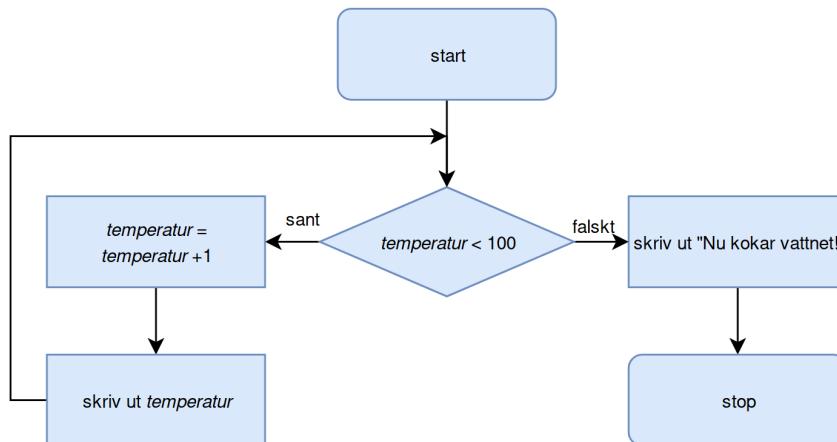
```

MEDAN temperatur är mindre än 100 SÅ
    Plusa på temperaturen med ett
    Skriv ut temperaturen
SLUT MEDAN
Skriv ut "Nu kokar vattnet!"

```

Detta kan illustreras som flödesschema så här:

Figur 6.1: En while-loop som flödesschema



Precis som med selektion (if-satser), så kan vi använda samtliga jämförelseoperatorer (som finns listade i delkapitel 1.3) när vi jobbar med iteration och while-loopar. Nu testar vi med mindre än-operatorn <.

I Python blir det:

Exempel 6.1: Vår första while-loop

```

1 temperature = float(input("Ange temperatur: "))
2 while temperature < 100:
3     temperature = temperature + 1
4     print("Temperaturen är nu:", temperature)
5 print("Nu kokar vattnet!")

```

Om vi anger den nuvarande temperaturen som 92 får vi följande resultat:

```

1 Ange temperatur: 92
2 Temperaturen är nu: 93.0
3 Temperaturen är nu: 94.0
4 Temperaturen är nu: 95.0
5 Temperaturen är nu: 96.0
6 Temperaturen är nu: 97.0
7 Temperaturen är nu: 98.0
8 Temperaturen är nu: 99.0
9 Temperaturen är nu: 100.0
10 Nu kokar vattnet!

```

Om vi istället anger temperaturen 100 (eller mer), kommer kodstycket i loopen aldrig att köras. Programmet hoppar då direkt till ”Nu kokar vattnet!”-meddelandet och vi får följande resultat:

```
1 Ange temperatur: 100
2 Nu kokar vattnet!
```

6.2 Oändliga loopar

När vi skriver koden för en loop så gäller det att se upp. Om vår jämförelse aldrig slutar vara sann, så fortsätter loopen att snurra i all oändlighet. Det kan t.ex. hända om vi glömmer att skriva koden för att plussa på temperaturen inuti loopen, så här:

Exempel 6.2: Oändlig loop, varning!

```
1 temperature = float(input("Ange temperatur: "))
2 while temperature < 100:
3     print("Temperaturen är nu:", temperature)
```

Om vi kör en oändlig loop så går det inte att köra någon mer kod efteråt. Vi kan då trycka Ctrl+C på tangentbordet för att avbryta vår loop.

6.3 for-loopen

Som sagt finns det flera typer av loopar. Den så kallade `for`-loopen har fördelen att den inte kan råka bli oändlig. Den är även mer koncis. Här är ett exempel på en `for`-loop:

```
1 start_temperature = 95
2 for t in range(start_temperature, 100):
3     print("Temperaturen är nu: ")
4     print(t)
```

Det finns ingen situation där man *måste* använda `for`-loopen, men det finns ändå situationer där man *måste* använda `while`-loopen. För att hålla denna bok enkel kommer vi inte att arbeta något mer med `for`-loopen.

6.4 Gissa talet!

Låt oss tillverka ett enkelt litet spel. Spelaren ska få gissa på ett tal mellan 1 och 100, ett tal som vi programmerare redan bestämt innan. Talet är 42. Så länge spelaren gissar fel, ska den få fortsätta att gissa. När spelaren har gissat rätt skriver vi ut ett grattis-meddelande. Vi tar det först på svenska:

```
Skriv ut "Gissa ett tal mellan 1 och 100"
Mata in tal
MEDAN talet inte är 42 SÅ
```

```
| Skriv ut "Fel. Gissa igen"  
| Mata in tal  
| SLUT MEDAN  
| Skriv ut "Grattis! Du gissade rätt!"
```

Kodat i Python blir det:

Exempel 6.3: Gissa talet

```
1 guess = int(input("Gissa ett tal mellan 1 och 100: "))  
2 while guess != 42:  
3     guess = int(input("Fel! Gissa igen: "))  
4 print("Grattis! Du gissade rätt!")
```

Här har min kompis försökt spela spelet:

```
1 Gissa ett tal mellan 1 och 100: 50  
2 Fel! Gissa igen: 25  
3 Fel! Gissa igen: 37  
4 Fel! Gissa igen: 44  
5 Fel! Gissa igen: 41  
6 Fel! Gissa igen: 42  
7 Grattis! Du gissade rätt!
```

Det här spelet är inte så roligt att spela mer än en gång. Vi kommer att återkomma till spelet i kapitel 7 där du också kan vidareutveckla det i en klurig övning.

6.5 Övningar

Övning (E), 6.1: Tal mellan 1 och 20

Skapa ett program som använder iteration för att skriva ut alla tal mellan 1 och 20.

Övning (E), 6.2: Tal mellan 1 och 100

Skapa ett program där användaren får mata in valfritt tal upp till 100. Programmet skriver sedan ut alla tal, från talet som användaren matade in upp till och med 100. Om man matar in ett tal som är större än 100 så stängs programmet av direkt.

Exempel:

```
1 Mata in ett tal: 93
2 93
3 94
4 95
5 96
6 97
7 98
8 99
9 100
```

Övning (E), 6.3: Singla slant

Be användaren mata in hur många gånger denne vill singla slant. Programmet ska sedan slumpvis mata ut om det blir krona eller klave, lika många gånger som användaren angett.

För att implementera övningen i Python behöver du använda funktionen `randint` för att slumpa fram nummer. Kom ihåg att du behöver som vanligt raden `from pylab import *` överst i din kod.

Övning (E), 6.4: Yatzy

Skapa ett program som fem gånger slumpar fram tärningsslag (tal mellan 1 och 6).

För att implementera övningen i Python behöver du använda `randint` för att slumpa fram nummer.

Övning (E), 6.5: Väderstationen

Skapa en lista som ska innehålla temperaturmätningar från en väderstation. I programmets början ska användaren få ange hur många mätningar som har gjorts. Därefter får användaren mata in olika temperaturer. Programmet ska sedan skriva ut de olika temperaturerna och medeltemperaturen.

Övning (M), 6.6: Multiplikationstabellen

Skapa ett program som skriver ut multiplikationstabellen, dvs skriv ut resultatet av att multiplicera $1 * 1$, sedan $1 * 2$, och så vidare hela vägen upp till $10 * 10$.

Tips: Du behöver använda två loopar - den ena inuti den andra.

7

PROBLEM LÖSNING

När man programmerar är det viktigt att kunna dela upp problemet i mindre delar. Det är faktiskt en av de allra viktigaste färdigheterna som en programmeare har. Ett mindre program kanske kan brytas ner till några få beståndsdelar, medan ett stort och komplext program kan innehålla tusen och åter tusen beståndsdelar. Man får bryta ner problemen i olika delar. Man börjar med det största problemet och bryter ner det i mindre underproblem. Dessa underproblem får sedan egna underproblem osv.

En tumregel för när man brytt ner problemen till lagom stora delar, är då det är möjligt söka efter lösningar med hjälp av en sökmotor på nätet. För att de ska vara sökbara, bör problemen kunna formuleras i några få nyckelord.

I det här kapitlet kommer vi att titta på två problem och hur man kan gå till väga för att lösa dem. Det första problemet är ganska kort och tanken är att du ska få lära dig hur du bryter ner problemet i så små delar att de är sökbara på nätet. Det andra problemet börjar från noll och byggs upp till en färdig lösning och du får se varje steg på vägen.

7.1 Nedbrytning och webbsökning

Låt oss igen ta gissa talet-spelet som exempel (från delkapitel 6.4). Istället för att alltid använda talet 42, så ska spelet slumpvis hitta på ett heltal mellan 1 och 100. Användaren ska sedan gissa talet. Gissar man fel ska programmet nu svara ”*Fel. Mitt tal är lägre*” respektive ”*Fel. Mitt tal är högre.*”. I slutet skriver programmet ut antalet försök.

Här är ett exempel på hur det kan se ut vid körning:

```
1 Gissa ett tal mellan 1 och 100: 15
2 Fel. Mitt tal är högre. Gissa igen: 50
3 Fel. Mitt tal är lägre. Gissa igen: 40
4 Fel. Mitt tal är högre. Gissa igen: 45
5 Fel. Mitt tal är högre. Gissa igen: 48
6 Rätt! Du behövde 5 gissningar.
```

Hur gör vi nu för att lösa detta? Vi kan kanske inte söka efter och hitta hela lösningen på webben. Däremot kan vi dela upp det i mindre delar och söka efter dem enskilt:

1. Vi kan söka information om hur man slumpar fram ett tal, genom att söka på "python random".
2. Vidare kan vi söka reda på hur man tar emot ett tal ifrån en användare, genom att söka på "python user input".
3. För att jämföra två tal med varandra kan vi söka på "python compare numbers"

Nu kanske vi redan vet hur man löser flera av dessa problem utan till. Då behöver vi såklart inte söka efter informationen på webben - men tankesättet är ändå detsamma när man bryter ner problem i mindre delar. På det här sättet har vi kommit närmare en lösning. Nu är det bara att sätta ihop dessa delar och skapa ett litet spel. Prova själv!

När man söker är det viktigt att inte ge upp i första taget. Men generellt gäller att om man inte hittar det man söker på något av de första 10 resultaten, är det oftast bättre att försöka med några nya sökord, än att leta vidare i lägre prioriterade resultat.

Övning (E), 7.1: Gissa talet

Skapa spelet "gissa talet" (se ovan) där datorn slumpar fram ett tal mellan 1 och 100 och användaren får gissa vilket tal det är.

När du spelar spelet så finns det en strategi som är den optimala för att alltid behöva göra så få gissningar som möjligt. Kan du hitta den strategin?

Övning (M), 7.2: Datorn gissar talet

Låt oss göra ungefär samma övning som ovan, fast denna gång är det användaren som bestämmer sig för ett tal (det räcker om den gör det tyst i huvudet) och datorn som gissar. Användaren får svara `r`, `h` eller `l` för ”rätt”, ”högre” eller ”lägre”. Så här skulle en exempelkörning kunna se ut:

```

1 Jag gissar på: 50
2 Är det [r]ätt? Eller är ditt tal [h]ögre eller
   [l]ägre? h
3 Jag gissar på: 75
4 Är det [r]ätt? Eller är ditt tal [h]ögre eller
   [l]ägre? l
5 Jag gissar på: 62
6 Är det [r]ätt? Eller är ditt tal [h]ögre eller
   [l]ägre? h
7 Jag gissar på 68
8 Är det [r]ätt? Eller är ditt tal [h]ögre eller
   [l]ägre? r
9 Hurra! Jag klarade det på 4 försök.
```

Kan du programmera datorn så att den använder samma optimala gissnings-strategi som du kom på när du gissade själv?

7.2 Hur ett program växer fram

Låt oss ta ett exempel där vi grundligt förklrarar hur vi tänker medan vi bygger upp programmet och löser problemet. Vi bygger upp programmet gradvis, steg för steg. När du ska lösa egna problem så kan det vara bra att tänka på samma steg-för-steg-sätt.

Vi gör ett program som skriver ut alla primtal mellan 1 och 100. Idén kommer från en grekisk herre vid namn Erastóthenes som levde för några tusen år sedan, långt innan det fanns datorer.

Ett sätt att skriva ut alla tal mellan 1 och 10 är med hjälp av en loop:

```

1 i = 1
2 while i <= 10:
3     print(i)
4     i = i + 1
```

Tänk dig att vi på något magiskt sätt redan vet vilka tal mellan 1 och 10 som är primtal och inte. Hur skriver vi ut bara dem, och inte resten? Vi behöver någon sorts `if`-sats (som vi lärde oss i kapitel 5):

```
1 i = 1
```

```

2 | while i <= 10:
3 |   if should_print[i]:
4 |     print(i)
5 |   i = i + 1

```

Ovanstående kod funkar inte i verkligheten- om du försöker köra den så klagar förstår Python på att `should_print` inte existerar. Men vi kan förstår skriva in den manuellt som en lista (som vi lärde oss i kapitel 3):

```

1 should_print = array([False, True, True, True, False,
2                      True, False, True, False, False])
3 i = 1
4 while i <= 10:
5   if should_print[i]:
6     print(i)
7   i = i + 1

```

Detta är förstår ingen lösning - vårt mål är ju att datorn ska räkna ut vilka tal som är primtal, inte att vi ska behöva skriva in det manuellt. Men det är en början.

En allmänt bra strategi för att lösa ett svårt problem när vi inte har någon aning om lösningen, är att börja med att lösa ett relaterat men mycket enklare problem. Så istället för att räkna ut alla primtal, låt oss börja med att räkna ut vilka tal som *inte* är delbara med 2, förutom talet 2 självt:

```

1 # skapa lista med 11 element. Varje element är True
2 should_print = repeat(True, 10+1)
3 # för alla tal som är en multipel av 2,
4 # sätt should_print till False
5 j = 4
6 while j <= 10:
7   should_print[j] = False
8   j = j + 2
9 # skriv ut
10 i = 1
11 while i <= 10:
12   if should_print[i]:
13     print(i)
14   i = i + 1

```

Förstår du while-loopen? Först sätter den `should_print[4]=False`, sen sätter den `should_print[6]=False`, och så vidare med 8 och 10.

Tänk om vi vill gå högre, alltså öka 10 till 25. Då måste vi ändra på tre olika ställen. Det är jobbigt i längden. Låt oss istället lägga tian i en variabel så att vi lätt kan ändra den:

```
1 limit = 25
2 should_print = repeat(True, limit+1)
3 # för alla tal som är en multipel av 2,
4 # sätt should_print till False
5 j = 4
6 while j <= limit:
7     should_print[j] = False
8     j = j + 2
9 # skriv ut
10 i = 1
11 while i <= limit:
12     if should_print[i]:
13         print(i)
14     i = i + 1
```

På samma sätt kan vi ju skriva ut alla tal som varken är delbara med 2 eller 3:

```
1 limit = 25
2 should_print = repeat(True, limit+1)
3 # för alla tal som är en multipel av 2,
4 # sätt should_print till False
5 j = 4
6 while j <= limit:
7     should_print[j] = False
8     j = j + 2
9 # för alla tal som är en multipel av 3,
10 # sätt should_print till False
11 j = 6
12 while j <= limit:
13     should_print[j] = False
14     j = j + 3
15 # skriv ut
16 i = 1
17 while i <= limit:
18     if should_print[i]:
19         print(i)
20     i = i + 1
```

Börjar du se mönstret? Tänk om vi gör detta inte bara för 2 och 3 utan för *alla* tal upp till och med 25. Då blir det ju bara primtalen kvar som skrivs ut. Men vi vill ju inte behöva skriva nästan samma kod 25 gånger på rad, så det är bättre att lägga in den koden i en till, yttre loop:

```

1 limit = 25
2 should_print = repeat(True, limit+1)
3
4 i = 2
5 while i <= limit:
6     # för alla tal som är en multipel av i,
7     # sätt should_print till False
8     j = i*2
9     while j <= limit:
10         should_print[j] = False
11         j = j + i
12     i = i + 1
13
14 # skriv ut
15 i = 1
16 while i <= limit:
17     if should_print[i]:
18         print(i)
19     i = i + 1

```

Hurra, det funkar! Vi har lyckats skriva ett kort program som skriver ut alla primtal mellan 1 och 25, och det är lätt att öka gränsen till 100. Programmet är dock inte så snabbt som det skulle kunna vara. Det gör en hel del beräkningar i onödan - det sätter `should_print` till `False` flera gånger för samma tal. Vi kan göra tre optimeringar.

För det första går den yttre loopen onödigt långt. Den går hela vägen upp till och med 25, men det behövs inte. Tänk dig att vi står på $i=7$ som ju är ett primtal. Då kommer vi att sätta `should_print=False` för $7*2 = 14$ och $7*3 = 21$. Men vi har ju redan gjort detta då vi stod på $i=2$ och $i=3$. Mer generellt så behöver den yttre loopen bara gå upp till och med 5, dvs kvadratroten ur 25. Vi använder funktionen `sqrt()` för att räkna ut kvadratroten.

För det andra. Först besöker vi alla multiplar av 2, dvs 4,6,8,10,12,14,16 osv. Och lite senare besöker vi alla multiplar av 4, dvs 8,12,16 osv, men det är ju onödigt, alla de är ju redan satta. När $i=4$ så borde vi inte köra den inre loopen alls, eftersom 4 inte är ett primtal. Mer generellt, när `should_print[i]` är `False` så borde vi inte köra den inre loopen alls.

För det tredje så börjar den inre loopen på ett onödigt lågt tal. När vi står på $i=3$ så behöver vi inte besöka $j=6$ för det har vi redan gjort, alltså kan den inre loopen börja på $j=9$. Och när vi står på $i=5$ så behöver vi inte besöka

$j==10$, $j==15$ eller $j==20$, för det har vi redan gjort, alltså kan den inre loopen då börja på $j==25$. Mer generellt kan den inre loopen börja på $j=i*i$.

Om vi implementerar alla dessa tre optimeringar så får vi den färdiga Sieve of Erastothenes:

```
1 limit = 25
2 should_print = repeat(True, limit+1)
3 i = 2
4 while i <= sqrt(limit): # optimering 1: sluta tidigare
5     if should_print[i]: # optimering 2
6         j = i * i; # optimering 3: börja loopa senare
7         while j <= limit:
8             should_print[j] = False
9             j = j + i
10            i = i + 1
11
12 # skriv ut
13 i = 1
14 while i <= limit:
15     if should_print[i]:
16         print(i)
17         i = i + 1
```


II

Övningar och facit

```
new
vel.append(newVel)
pos.append(newPos)
i += 1
plt.plot(pos)
plt.plot(vel)
# closed-form equivalent:
= 0.0257
d = 0.2255
= []
= 0
le i < 200:
    x.append(pow(math.e, (-r * i)))
```


8

ÖVNINGAR

8.1 Primtal, delbarhet och faktorisering

Följande övningar är lämpliga för Ma1c och är relaterade till följande centrala innehåll:

- **Taluppfattning, aritmetik och algebra:** [...] begreppen primtal och delbarhet. (Ma1c)

Övning (E), 8.1: Kontrollera faktorer (Ma1c)

Kim hävdar att om vi multiplicerar alla tal i listan [3, 5, 7, 17, 23] så blir produkten lika med 41055, och ber dig skriva ett program som kontrollräknar. Du föreslår att Kim helt enkelt skriver in $3*5*7*17*23$ som ett kommando i Python, men Kim säger att han kommer vilja använda programmet på väldigt långa listor som redan är färdigskrivna, så han orkar inte skriva in så många multiplikationstecken. Skriv ett program åt Kim så att han lätt kan kontrollräkna sånt själv i fortsättningen. Programmets första två rader ska vara:

```
1 expected_product = 41055 # här kan Kim ändra  
2 all_factors = array([3, 5, 7, 17, 23]) # här också
```

Programmet ska sedan använda de två variablerna, och skriva ut ”Rätt” eller ”Fel”.

Övning (E), 8.2: Delbart med 3? (Ma1c)

Skriv ett program som låter användaren mata in ett heltal. Programmet ska testa om talet är delbart med 3 eller inte. Skriv ut ”delbart” eller ”ej delbart” beroende på vad det är.

Tips: Bläddra tillbaka till exempel 1.11 och läs om funktionen `fmod`.

Övning (M), 8.3: Primtal eller ej (Ma1c)

Skriv ett program som låter användaren mata in ett heltal. Kolla om talet är ett primtal. Skriv ut ”primtal” eller ”ej primtal” beroende på vad det är.

Tips: Utgå från förra övningen och bygg vidare på den.

Övning (M), 8.4: Faktorisera ett heltal (Ma1c)

Skriv ett program som låter användaren mata in ett heltal. Skriv ut alla talets primtalsfaktorer.

Tips: Utgå från förra övningen och bygg vidare på den.

Övning (M), 8.5: Testa att programmet stämmer (Ma1c)

Gör om ovanstående faktoriserings-program så att det lägger alla faktorerna i en lista. Använd sedan programmet från Övning 8.1 för att testa att ditt faktoriserings-program verkligen räknade rätt. Om programmen inte håller med varandra så är det fel i antingen det ena eller det andra programmet. Hitta i så fall felet och fixa det.

8.2 Sannolikhet och statistik

Följande övningar är lämpliga för Ma1c och Ma2c och är relaterade till följande centrala innehåll:

- **Sannolikhet och statistik:** [...]etoder för beräkning av sannolikheter vid slumpförsök i flera steg med exempel från spel[...]. (Ma1c)
- **Sannolikhet och statistik:** Statistiska metoder för rapportering av observationer och mätdata från undersökningar. (Ma2c)
- **Sannolikhet och statistik:** Metoder för beräkning av olika lägesmått och spridningsmått inklusive standardavvikelse, med digitala verktyg. (Ma2c)
- **Samband och förändring:** Konstruktion av grafer till funktioner [...] med digitala verktyg. (Ma2c)

Programmering kan vara lämpligt för att räkna på och förstå sannolikhet och statistik. Matteuppgifter om sannolikhet har ofta en exakt lösning som vi kan få fram via någon färdig formel. Din vanliga mattebok är garanterat fyllt av såna. Men vad gör du om du får en uppgift där du inte har någon färdig formel? Om du klurar tillräckligt länge så kanske du kan komma på en formel själv. Men det är ofta lättare att istället skriva ett program som gör en simulering, t.ex. kastar tärning jättemånga gånger, och sedan samlar ihop statistik om resultatet. Du får inte ett exakt resultat, men ett ungefärligt. Ju fler kast desto mer exakt.

Exempel: Om du kastar två tärningar (vanliga sexsidiga tärningar), vad är sannolikheten att summan blir 2? Att den blir 3? Och så vidare.

Vi skriver ett program som kastar en tärning 100 gånger, och ritar resultatet i ett histogram som vi lärde oss i delkapitel 4.2:

```
1 random_numbers = array([])
2 number_throws = 100 # den här kan man ändra
3 throw = 1
4 while throw <= number_throws:
5     dice1 = randint(1, 7)
6     random_numbers = append(random_numbers, dice1)
7     throw = throw + 1
8 hist(random_numbers, arange(min(random_numbers)-0.5,
9     max(random_numbers)+1.5))
9 show()
```

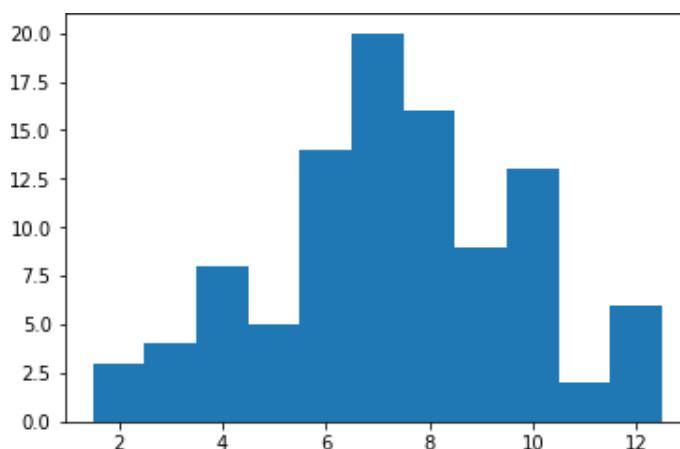
Nu utökar vi programmet så att det kastar två tärningar åt gången, och mäter summan av dem:

```

1 random_numbers = array([])
2 number_throws = 100 # den här kan man ändra
3 throw = 1
4 while throw <= number_throws:
5     dice1 = randint(1, 7)
6     dice2 = randint(1, 7)
7     sum = dice1 + dice2
8     random_numbers = append(random_numbers, sum)
9     throw = throw + 1
10 hist(random_numbers, arange(min(random_numbers)-0.5,
11     max(random_numbers)+1.5))
12 show()

```

Figur 8.1: Histogram över 100 summor



Övning (E), 8.6: Funktionen randi (Ma1c, Ma2c)

Eftersom programmet använder slumptalsfunktionen `randint` så kommer resultatet att bli olika varje gång du kör programmet. Prova det några gånger!

Övning (E), 8.7: När blir datorn långsam? (Ma1c, Ma2c)

Datorn är såpass snabb att den kan simulera hundra kast utan problem. Prova att öka antalet kast till 1000, 10000 och så vidare och kolla när det börjar gå långsamt på din dator!

Övning (E), 8.8: Hur förändras formen? (Ma1c, Ma2c)

Hur förändras formen på histogrammet när du ökar antalet kast, från 100 till 1000 och 10000? Beskriv med ord.

Övning (E), 8.9: Skillnad mellan två tärningar (Ma1c, Ma2c)

Gör om programmet så att det inte längre mäter summan av de två tärningarna, utan istället mäter skillnaden mellan dem. Här måste du tänka på att vi inte är intresserade av negativa skillnader. Alltså, om det första tärningskastet ger 2 och det andra tärningskastet ger 5 så vill vi ändå tänka att skillnaden är 3, inte -3 .

Tips: bläddra tillbaka till delkapitel 1.6.6 och läs om funktionen `abs`.

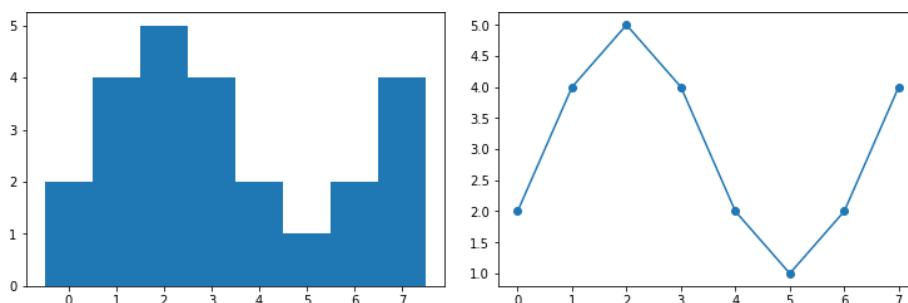
Övning (E), 8.10: Histogramanalys (Ma1c, Ma2c)

Titta i det nya histogrammet. Vad är den vanligaste skillnaden mellan två tärningar? Alltså, vad blev det oftast?

Övning (M), 8.11: Beräkna histogram själv (Ma1c, Ma2c)

Tänk om funktionen `hist` inte fanns, men du ändå väldigt gärna ville rita nån sorts histogram. Och tänk om funktionen `plot` fortfarande fanns. Du skulle kunna skriva egen kod som utgår från `random_numbers` och gör en sammanfattande beräkning och till slut använder `plot` för att rita sammanfatningen. Hur ser den koden ut? Observera att `plot(random_numbers)` inte är rätt svar. Din graf ska visa samma information som `hist`, förutom att den rent visuellt använder linjer istället för staplar.

Figur 8.2: Plot som motsvarar histogram



Tips: bläddra tillbaka till delkapitel 2.5.1 och läs om funktionen `int`.

Övning (E), 8.12: Chans att slå Yatzy på ett slag (Ma1c)

Alex spelar spelet Yatzy. I det spelet får hon kasta fem tärningar. Beroende på vad hon slår så blir det olika poäng. Det som ger mest poäng är att alla fem tärningarna får samma tal - detta kallas också Yatzy, och antalet ögon på tärningarna spelar alltså ingen roll. Fem 1:or är lika mycket Yatzy som fem 6:or. Gör ett program som beräknar sannolikheten att slå Yatzy på ett slag.

Tips: Om du är osäker på var du ska börja, gå tillbaka till Övning 6.4, och hitta på ett sätt att testa om alla tärningarna visar samma tal eller inte. Kör sedan tusentals spelomgångar och håll räkningen på hur många av dem som blir Yatzy.

Övning (M), 8.13: Spara tärningar i Yatzy (Ma1c, Ma2c)

Alex fortsätter spela Yatzy (se föregående övning). Om hon inte är nöjd med sitt första försök, så får hon välja att låta valfritt antal tärningar ligga kvar på bordet, och plocka upp resten och slå om dem. Den lättaste strategin för att slå Yatzy är att spara de tärningar som har samma tal, och slå om resten.

Skriv ett program som analyserar en lista med fem tärningar, och skriver ut vilket värde som är smartast att spara nästa slag. Om det finns flera på delad förstaplats så spelar det ingen roll vilken av dem som programmet skriver ut. Testkör programmet med dessa listor:

- [1, 4, 5, 4, 3]. Det är smartast att spara 4:orna.
- [1, 4, 5, 4, 1]. Spara 4:orna eller 1:orna, vilket som.
- [1, 4, 4, 4, 1]. Spara 4:orna.
- [1, 4, 5, 6, 3]. Alla olika, spelar ingen roll vad som sparas.

Övning (S), 8.14: Chans att slå Yatzy (Ma1c, Ma2c)

Enligt de riktiga reglerna för Yatzy (jämför föregående övning) så får man *två* gånger välja ett valfritt antal tärningar och slå om dem. Alex vill veta vad sannolikheten för att slå Yatzy är. Hjälp henne att skriva ett program som ger henne svar på frågan genom att spela väldigt många (säg 10000, 100000 eller ännu fler) spel.

Exempel:

- Första kastet: Tärningarna visar 4 5 1 4 2
- Alex sparar de två 4:orna och kastar om de tre övriga.
- Andra kastet: De nykastade tärningarna visar 1 1 1. Totalt visar tärningarna alltså 4 1 1 4 1.
- Nu har ju Alex fler 1:or än 4:or, så hon sparar de tre 1:orna och kastar om resten.
- Tredje kastet: De nykastade tärningarna visar 5 3. Totalt visar tärningarna alltså 5 1 1 3 1.
- Alex har nu gjort tre kast och får inte göra fler. Alex fick tyvärr inte Yatzy.

Tips:

- *Börja med att skriva koden för att göra ett enda kast med fem tärningar (se Övning 8.12).*
- *Skriv sedan kod som räknar ut vilka tärningar som är bäst att spara (se Övning 8.13).*
- *Ändra sedan koden så att den upprepar ovanstående tre gånger. Detta är en spelomgång. Men kasta inte om alla tärningar, utan bara de som inte ska sparas.*
- *Kör tusentals spelomgångar och hållräkningen på hur många av dem som blir Yatzy.*

Övning (E), 8.15: Standardavvikelse (Ma2c)

Utgå från följande lista: [4, 9, 10, 7.5, 8, 9, 3, 9, 4, -2]

Skriv ett program som räknar ut listans medelvärde och standardavvikelse. Vi utgår från att du redan känner till vad det innebär. Använd iteration.

Övning (S), 8.16: Typvärde (Ma2c)

Utgå från följande lista: [4, 9, 10, 7.5, 8, 9, 3, 9, 4, -2]

Skriv ett program som räknar ut listans typvärde. Vi utgår från att du redan känner till vad det innebär. Använd iteration.

Tips: Om du tycker att övningen är svår, börja med att lösa Övning 8.13.

Övning (M), 8.17: Inbyggda funktioner för lägesmått och spridningsmått (Ma2c)

Python innehåller förstås färdiga funktioner för att räkna ut medelvärdet, typvärdet och standardavvikelse. Gör en ny version av ovanstående övningar. Sök information om de inbyggda funktionerna `mean`, `mode` och `std`, och använd dem istället för att göra beräkningen själv. För `mode` kan du behöva importera ett extra s.k bibliotek via `from scipy.stats import mode`.

8.3 Numerisk lösning av linjära ekvationer

Följande övningar är lämpliga för Ma1c och är relaterade till följande centrala innehåll:

- **Taluppfattning, aritmetik och algebra:** Algebraiska och grafiska metoder för att lösa linjära ekvationer och olikheter samt potensekvationer[...] (Ma1c)

Övningarna lämpar sig även för Ma2c och Ma3c, då de är förkunskapskrav för att kunna lösa andragradsekvationer och för att kunna approximera talet e

Grafer är bra för att utforska ekvationer. Om din vanliga mattebok t.ex. nämner ekvationen $3x - 7 = 5$ och ber dig att hitta vad x är, så ska du förstås kunna göra beräkningen för hand med penna och papper, men att även se ekvationen grafiskt kan vara ett komplement som ökar din förståelse, och ett sätt att dubbelkolla att du räknat ut rätt svar.

Vi kan betrakta de två sidorna i ekvationen som varsin linje. Vi kan rita båda linjerna tillsammans i samma graf, så här:

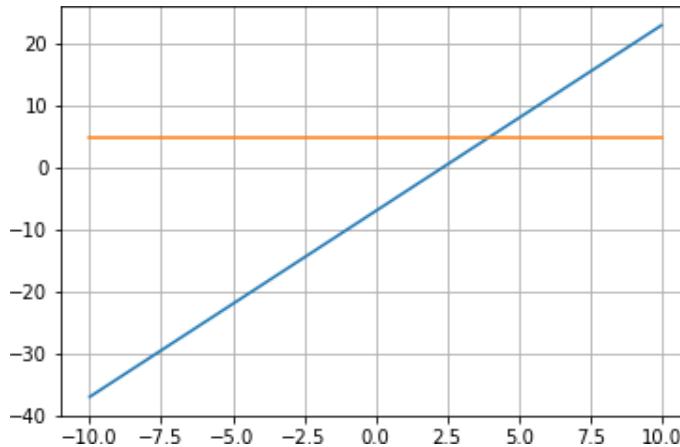
```

1 xmin = -10
2 xmax = 10
3 x = arange(xmin, xmax+1)
4 left_side = 3 * x - 7
5 plot(x, left_side) # det till vänster om likamedtecknet
6 plot([xmin,xmax], [5,5]) # det till höger
7 grid(True)
8 show()

```

Notera att vi använder `plot()` på två olika sätt, som vi lärde oss i delkapitel 4.1.3.

(Se graf på nästa sida...)

Figur 8.3: Ekvationen visualiseras som två linjer

Vad innebär nu egentligen ekvationen $3x - 7 = 5$? Den innebär att det finns något x som gör att vänstersidan och högersidan om likamedtecknet får samma värde. Grafiskt innebär det att de två linjerna någonstans korsar varandra. Titta på bilden! Det x -värde där linjerna korsar varandra är rätt svar på övningen. Alla de andra x -värdena i grafen gör att de två linjerna hamnar på olika höjd, så de är inte lösningar på övningen. Om du kommer fram till ett x när du räknar ut ekvationen på papper, och ett annat x när du tittar i grafen, så vet du att du har gjort fel någonstans.

När du tittar på bilden så använder du ögonen för att hitta det x -värde där linjerna möts. Men vi skulle kunna programmera datorn för att göra ungefär samma sak, utan ögon. Datorn är så snabb att den kan helt enkelt testa jättemånga x -värden för att hitta det värde där ekvationen stämmer. Detta kallas *numerisk lösning*.

Övning (E), 8.18: Enkel numerisk lösning (Ma1c)

Skriv ett program som går igenom alla heltalet mellan -10 och 10 , och testar om ekvationen stämmer. Med andra ord, börja med att låta x vara lika med -10 , gör beräkningen $3x - 7$, och om resultatet blir lika med 5 så skriv ut värdet på x , annars öka x ett steg till -9 och upprepa. Programmet ska skriva ut svaret 4 .

Den enkla metoden för numerisk lösning har dock några svagheter:

Övning (E), 8.19: Enkel numerisk lösning, svaghet 1 (Ma1c)

Prova att ändra ekvationen till $3x - 7 = 29$. Ditt program kommer inte att hitta lösningen. Varför? Och vad är det lättaste sättet att fixa programmet så att det hittar lösningen?

Övning (E), 8.20: Enkel numerisk lösning, svaghet 2 (Ma1c)

Prova att ändra ekvationen till $3x - 7 = 4$. Ditt program kommer inte att hitta lösningen. Varför?

Övning (E), 8.21: Lös linjär ekvation med `fsolve` (Ma1c)

Det är förstås möjligt att skriva ett bättre program som inte har svagheterna i de två ovanstående övningarna. Men det finns, föga förvånande, redan en färdig, inbyggd funktion som inte har dessa två svagheter. Först måste vi göra om ekvationen för hand så att den har en nolla på högersidan, alltså $3x - 7 - 4 = 0$. Sen använder vi funktionen `fsolve` såhär:

```
1 from scipy.optimize import fsolve
2 print(fsolve(lambda x: 3*x-7-4, 0))
```

Vi behöver `lambda x:` innan vi skriver själva ekvationen för att Python ska fatta att `x` är en okänd variabel. Efter ekvationen kommer det andra argumentet till `fsolve`, där vi skickar in en `0:a`. Datorn utgår då från värdet $x = 0$ som en första gissning när den börjar lösa ekvationen. Datorn kan misslyckas med att hitta en lösning om den första gissningen inte är tillräckligt nära rätt svar.

Övning: Använd `fsolve` för att lösa ekvationen: $4x + 15 = -9$. Vad är x ?

Använd sedan `fsolve` på några andra ekvationer från din vanliga mattebok.

Övning (S), 8.22: Klurig numerisk lösning med intervallhalvering (Ma1c)

Skriv ett eget program som kan lösa ekvationen $3x - 7 = 4$ och valfri annan ekvation av samma sort, och som inte har ovannämnda svagheter nummer två. (Använd inte den inbyggda `fsolve`.)

Det räcker inte att ta kortare steg, för hur korta steg ska vi ta för att vara säkra på att inte hoppa över svaret? Om vi tar steg med längden 0.1 så kommer vi ju att hoppa från $x=3,6$ till $x=3,7$ och därmed hoppa över svaret.

Tricket är att, istället för att bara leta efter *ett* x där vänsterledet blir exakt lika med 4, så kan vi utgå ifrån *två olika* x som ligger på varsin sida om rätt svar, och sedan ta oss närmare och närmare svaret med hjälp av samma metod som den optimala strategin för ”gissa talet”-spelet vi jobbade med i Övning 7.1.

Den här uppgiften är avsiktligt lite klurig. Börja med att fundera på problemet geometriskt med penna och papper, och se om du kan se samband mellan denna övning och hur du bäst löser gissa talet-spelet.

8.4 Andragradsekvationer

Följande övningar är lämpliga för Ma2c och Ma3c och är relaterade till följande centrala innehåll:

- **Taluppfattning, aritmetik och algebra:** Algebraiska och grafiska metoder för att lösa exponential-, andragrads- och rottekvationer [...] (Ma2c)
- **Samband och förändring:** Konstruktion av grafer till funktioner [...] med digital verktyg. (Ma2c)
- **Samband och förändring:** Egenskaper hos andragradsfunktioner. (Ma2c)
- **Samband och förändring:** Algebraiska och grafiska metoder för lösning av extremvärdesproblem (Ma3c)

Övning (E), 8.23: Lös andragradsekvation med `roots` (Ma2c)

Kan Python hjälpa oss att lösa andragradsekvationer? Funktionen `fsolve`, som vi lärde oss i Övning 8.21, passar inte så bra här, för `fsolve` hittar bara en lösning, men andragradsekvationer kan ha två lösningar. Lyckligtvis finns det en annan funktion som heter `roots`. Den passar bättre för andragradsekvationer och, mer generellt, för den typ av ekvationer som kallas *polynomekvationer*. Följande är exempel på polynomekvationer:

- $x = 5$
- $3x + 7 = 0$
- $x^2 - 2x - 12 = 0$
- $4x^2 + 9 = 0$
- $2x^3 + 6x^2 + 13x - 8 = 0$

Följande är *inte* polynomekvationer och går alltså inte att lösa med `root`:

- $7/x = 0$
- $3^x = 0$

- $\sin(x) = 0$

För att lösa en polynomekvation ser vi först till att den har rätt form, annars slår vi ihop och flyttar över termer tills den har rätt form. Sen tar vi bort alla x och x^2 , och gör en lista av det som blir kvar på vänstersidan. Generellt så blir ekvationen $ax^2 + bx + c = 0$ listan $[a, b, c]$. T.ex ekvationen $3x^2 = 4$ gör vi först om till $3x^2 + 0x - 4 = 0$ och sedan till listan $[3, 0, -4]$. Slutligen ger vi denna lista som ett argument till `roots` och skriver ut resultatet:

```
1 print(roots([3, 0, -4]))
```

Övning: lös så många som möjligt av dessa ekvationer med hjälp av `roots`:

- $6x^2 - 13x + 5 = 0$
- $4x^2 - 2x - x/x = 5$
- $2/x - 18 = 0$
- $2x^2/2 + 2x^2 - x - x = 0$

Övning (S), 8.24: Klurig numerisk lösning, del 2 (Ma2c)

Ekvationen i Övning 8.22 är nog ärligt talat lättare att lösa för hand än genom att skriva ett klurigt datorprogram. Men när du klarat den övningen, testa att använda samma program för att lösa den mycket mer komplexa ekvationen $x^6 - \sin(x) - 3^x + 7 = 0$.

Vilket värde har x ? Starta sökningen i intervallet $-50 \leq x \leq 50$. Om du har lyckats göra programmet tillräckligt generellt så ska det fungera!

Hade du kunnat lösa ut x för hand i denna ekvation?

Övning (S), 8.25: Klurig numerisk lösning, del 3 (Ma2c)

Testa att använda samma program för att lösa ekvationen $x^2 + 6x + 9 = 0$. Fungerar det? Om inte, varför inte?

Övning (M), 8.26: Andragradsfunktionens minsta värde, grafiskt (Ma2c, Ma3c)

Diogenes har fått i uppdrag att tillverka en stor plåtburk / liten tunna som ska rymma 6.2832 dm^3 vätska. Tunnan ska ha formen av en cylinder. Diogenes tänker göra tunnan av en rektangular plåtbit som rullas ihop till en cylinder, plus två cirkulära plåtbitar som täpper till ändarna på cylindern. Beroende på hur långsmal Diogenes gör tunnan så kommer det gå åt olika mycket plåt. Uppdragsgivaren bad Diogenes att använda så lite plåt som möjligt till tunnan.

Fråga: Vilken radie och höjd ska Diogenes välja på sin tunna?

Eftersom vi vet att volymen ska vara 6.2832 dm^3 så kan radien och höjden inte variera oberoende av varann. Om Diogenes väljer en radie i dm , så ges höjden h automatiskt av:

$$h = 6.2832 / (\pi * \text{radie}^2)$$

Ekvationen för hur mycket plåt som behövs är i dm^2 :

$$\text{area} = 2 * \pi * \text{radie} * (\text{radie} + h)$$

Problemet går att lösa analytiskt, men här ska vi lösa det genom att titta i en graf. Börja med att skapa en lista med ett antal tänkbara radier, till exempel från 0 till 10. Använd sedan ovanstående ekvationer för att beräkna en lista med höjder, och en lista med areor. Använd slutligen funktionen `plot()` för att grafiskt se vilken radie som ger den minsta arean. Du kan behöva ändra på din lista med radie-förslag flera gånger för att ”zooma in” på rätt del av grafen.

8.5 Potensfunktioner och exponentialfunktioner

Följande övningar är lämpliga för Ma1c och Ma2c och är relaterade till följande centrala innehåll:

- **Samband och förändring:** Begreppen förändringsfaktor och index. Metoder för beräkning av räntor och amorteringar för olika typer av lån [...] (Ma1c)
- **Samband och förändring:** egenskaper hos [...] exponentialfunktioner. (Ma1c)
- **Samband och förändring:** Konstruktion av grafer till funktioner [...] med digital verktyg. (Ma2c)
- **Taluppfattning, aritmetik och algebra:** Algebraiska och grafiska metoder för att lösa [...] exponentialekvationer (Ma2c)
- **Sannolikhet och statistik:** Metoder för beräkning av olika lägesmått och spridningsmått inklusive standardavvikelse [...]. (Ma2c)

Övning (E), 8.27: Exponentialfunktion (Ma1c, Ma2c)

En nystartad sommarfestival släpper 1 000 biljetter som blir slutsålda direkt. Arrangörerna vill att festivalen ska växa i en lagom takt, så de bestämmer sig för att släppa 10% fler biljetter varje år. Hur många biljetter släpps till den 21:a festivalen, när det alltså gått 20 år sedan starten? Använd iteration! Som bonus kan du även plotta en graf över antalet biljetter per år.

Övning (E), 8.28: Numerisk lösning av exponentialekvation (Ma1c, Ma2c)

Utgå från föregående övning. Vilken festival i ordningen blir den första att släppa 5 000 eller fler biljetter? Programmet ska skriva ut svaret. Använd iteration! Tänk på att festival nummer 2 inträffar 1 år efter starten, festival nummer 3 inträffar 2 år efter starten, och så vidare.

Övning (M), 8.29: Oregelbunden exponentialfunktion (Ma1c, Ma2c)

Det finns olika sätt att lösa ovanstående två matteproblem. Ett sätt är med algebraisk/analytisk matematik, genom att lösa ut relevanta variabler ur ekvationen $y = C * a^x$. Det numeriska sättet är att upprepa en uträkning i en loop, där varje varv i loopen t.ex. representerar att det gått ett år.

Det finns för- och nackdelar med de olika sätten. I de ovanstående övningarna vore nog faktiskt det analytiska sättet mycket enklare än det numeriska. Men ju mer ”oregelbundet” ett matematiskt problem är, desto svårare brukar det vara att lösa analytiskt, och desto lämpligare blir då det numeriska. T.ex. om ökningstakten i en exponentialfunktion inte är samma procent varje år, utan varierar.

En annan årlig festival bestämmer sig för att busa med matematiker och ändrar biljettantalet enligt följande sicksack-mönster:

<i>Festival nummer</i>	2	3	4	5	6	7	8	9
<i>Ökning sen förra året</i>	4%	2%	5%	3%	6%	4%	7%	5%

osv...

Övning: Den första festivalen har 100 biljetter. Hur många biljetter har den 26:e festivalen, när det alltså gått 25 år sedan start?

Hade du kunnat lösa denna övning analytiskt? Hade du orkat göra det numeriskt utan programmering?

Övning (S), 8.30: Oregelbunden exponentialfunktion, del 2 + slumpförsök och statistik (Ma1c, Ma2c)

Det är förstås inte så vanligt i verkligheten att ökningstakten i en exponentialfunktion går upp och ner i ett exakt sicksack-mönster. Det är betydligt vanligare att vi inte vet i förväg exakt hur stor ökningstakten kommer bli, men att vi vill göra någon sorts kvalificerad gissning ändå.

Till exempel: En population på 20 kaniner har utplanterats i ett område där det inte finns några rovdjur, men en del andra faror. Vi utgår från att antalet kaniner ökar med mellan 10% och 28% per månad.

Fråga: Ungefär hur många kaniner finns det efter 24 månader?

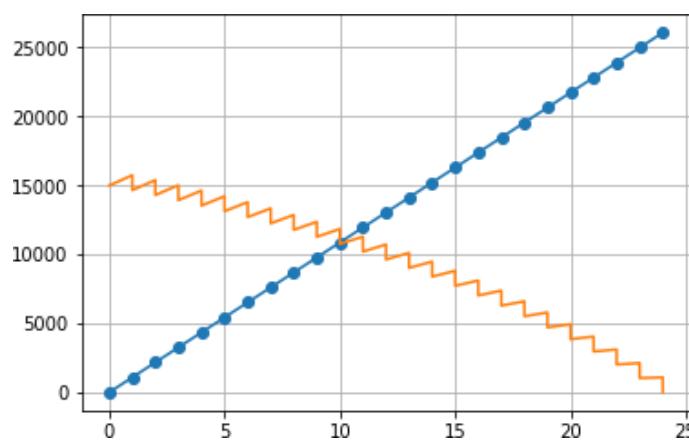
Tips: Vi kan välja ett slumpmässigt tal mellan 10% och 28% och låtsas att det talet är ökningstakten just den månaden. Vi kan välja ett nytt slumptal nästa månad, och så vidare, upp till 24 månader. Detta ger oss en gissning, ett tänkbart framtidsscenario för hur många kaniner som skulle kunna finnas om 24 månader. Skriv ett program som skapar en sådan gissning!

Utöka sedan programmet så att det gör om hela beräkningen 100 gånger. Du får då 100 olika gissningar för hur stor kaninpopulationen kommer vara efter 24 månader. Beräkna medelvärdet av alla dessa gissningar. Som bonus kan du även välja att plotta alla 100 scenarion i en och samma graf.

Denna metod brukar kallas en Monte Carlo-simulering, efter det berömda casinot i Monaco.

Övning (S), 8.31: Avbetalning av annuitetslån (Ma1c, Ma2c)

Gizem lånar 15 000 kr. Det är ett annuitetslån på 24 månader, vilket innebär att hon betalar tillbaka samma summa varje månad: 1087.07 kr. Men den delen av lånet som hon ännu inte betalat tillbaka, växer med månadsräntan 5%. Följande graf illustrerar lånet.

Figur 8.4

Den stigande blå linjen visar hur mycket pengar Gizem har betalat totalt efter varje månad. Den fallande orange trapp-likt linjen visar hur den återstående skulden förändras. Notera att den går både uppåt och nedåt - upp pga räntan, ner pga återbetalningen. Skriv koden för att skapa samma graf själv!

Tips: Rrita linjerna med metoden som vi lärde oss i delkapitel 4.1.3. Den orange linjen behöver två värden för varje månad, inte bara ett värde.

8.6 Derivata

Följande övningar är lämpliga för Ma3c och är relaterade till följande centrale innehåll:

- **Samband och förändring:** Orientering när det gäller kontinuerlig och diskret funktion samt begreppet gränsvärde. (Ma3c)
- **Samband och förändring:** Begreppen [...] ändringskvot och derivata för en funktion. (Ma3c)
- **Samband och förändring:** Algebraiska och grafiska metoder för bestämning av derivatans värde för en funktion. (Ma3c)
- **Samband och förändring:** Introduktion av talet e och dess egenskaper. (Ma3c)

Övning (E), 8.32: Försämringshastighet enligt tabell (Ma3c)

Koldioxidhalten i jordens atmosfär ökar varje år. Följande tabell innehåller årliga medelvärden av koldioxidhalten i PPM (parts per million), uppmätta på berget Mauna Loa på Hawaii:

```

1 x = array([2007, 2008, 2009, 2010, 2011, 2012, 2013,
2   2014, 2015, 2016, 2017])
3 y = array([383.79, 385.60, 387.43, 389.90, 391.65,
4   393.85, 396.52, 398.65, 400.83, 404.21, 406.53])
# källa:
# https://www.esrl.noaa.gov/gmd/ccgg/trends/data.html

```

Fråga: Hur snabbt ökade koldioxidhalten år 2016? Svarer ska ha enheten PPM/år.

Skriv ett program som approximerar derivatan i punkten $x = 2016$ med hjälp av ”central ändringskvot” - mät med andra ord den genomsnittliga ökningen från 2015 till 2017.

Övning (E), 8.33: Försämringshastighet enligt tabell, del 2 (Ma3c)

Utgå från föregående övning. Förbättra programmet så att det kan approximera derivatan för valfritt år. När programmet körs får användaren mata in det årtal de är intresserade av (alltså med funktionen `input`) mellan 2008 och 2016.

Tips: Kom ihåg argwhere som vi gick igenom i delkapitel 3.6.

Övning (E), 8.34: Ändringskvot för funktion (Ma3c)

Utgå från föregående övning, men byt från en tabell till funktionen $f(x) = (x^2 + 1)/x$

Till skillnad från ovanstående tabell, där vi jobbade med hela årtal, så är det nu inte självklart hur långt framåt och bakåt vi ska gå för att välja ”grannar” till x . För denna övning, välj avståndet $h = 0.4$ till grannarna! Använd programmet för att approximera derivatan för $x = 1$, så grannarna blir alltså 0.6 och 1.4.

Övning (M), 8.35: Testa olika värden för h (Ma3c)

Bygg vidare på föregående program och låt datorn testa en massa olika värden för h . Börja med $h = 0.4$, beräkna ändringskvoten, halvera h , beräkna ändringskvoten igen, och så vidare. Upprepa 8 gånger. Plotta en graf med ändringskvoten på y-axeln, och antalet halveringar på x-axeln. Baserat på grafen, vad tror du att den exakta derivatan är?

Övning (E), 8.36: Testa den inbyggda gradient (Ma3c)

Det finns (som vanligt) inbyggda funktioner för att mäta ändringskvoten. Bland annat en som heter `gradient` som i princip ersätter koden vi själva skrev i Övning 8.34. Vi måste fortfarande välja ett h själva, och svaret är fortfarande bara ungefärligt. Så här använder vi `gradient`:

```

1 xp = 1 # i vilken punkt vill vi veta lutningen
2 h = 0.001
3 x = array([xp - h, xp, xp + h])
4 y = (x**2 + 1) / x # vår funktion
5 estimate = gradient(y, h)[1]
6 print(estimate)

```

För att testa en annan punkt eller funktion behöver du bara ändra på rad 1 och 4.

Använd nu `gradient` för att beräkna $f'(1)$ för funktionen $f(x) = 3\sqrt{x} + 2/x^2$

Övning (S), 8.37: Approximera talet e (Ma3c)

Finns det något tal e som uppfyller att derivatan av e^x (med avseende på x) är lika med e^x för alla x ? Ja, det finns det. Skriv ett program som hittar (ett närmevärde till) det talet genom att prova sig fram.

Du behöver ett sätt för datorn att prova sig fram och närlägga sig svaret, så gör först Övning 7.2, och sedan Övning 8.22.

Sen behöver du ett sätt att mäta ändringskvoten i en viss punkt, så utgå från koden du skrev i Övning 8.34. Kombinera allt du lärt dig i de övningarna, så kan du nog lösa även denna!

Observera att du inte behöver testa för alla x , utan bara *något* x , och det spelar ingen roll vilket x du väljer, för resultatet blir ändå detsamma. Använd förslagsvis $x = 1$, så blir koden lite enklare - du kan då söka efter ett e där ändringskvoten för e^x kring punkten $x = 1$ är lika med $e^1 = e$. Och vilket intervall ska du söka i? Tips: e är större än 0, så du behöver inte söka bland negativa tal.

9

FACIT OCH LÖSNINGSFÖRSLAG

Här följer facilit till denna bok. I många av övningarna i denna bok ska du skriva kod. Den kod som presenteras här ska ses som lösningsförslag, snarare än ett absolut facilit. Med programmering går det ju att lösa ett problem på många olika sätt.

Facit till kapitel 1 - Datorn som miniräknare

Övningarna i kapitlet behöver inget facilit.

Facit till kapitel 2 - Variabler

Övning 2.1 - Höger eller vänster?

Efter att ha kört de tre kodraderna så är `a=2` och `b=2`. Alltså, när det står två variabler på varsin sida om tilldelningsoperatorn så ändras den på vänstra sidan.

Övning 2.2 - Kopplas variabler ihop för all framtid?

På rad 2 så får `b` det dåvarande värdet av `a`, alltså blir `b=1`. Efter att ha kört de tre kodraderna så är `b=1` fortfarande. Det är *inte* så att `b` ”kopplas ihop” med `a` för all framtid. När vi på rad 3 ändrar värdet på `a` så ändras alltså *inte* värdet på `b`.

Övning 2.3 - Summan och medelvärdet av tre tal

```
1 a = 23
2 b = 45
3 c = 67
4 print(a + b + c)
```

```
5 | print((a + b + c) / 3)
```

Övning 2.4 - Decimaltal till heltal

```
1 | a = 11.534
2 | print(round(a))
```

Facit till kapitel 3 - Listor

Övning 3.1 - Olika sätt att skapa samma lista

```
1 | # omständigt sätt:
2 | a = array([])
3 | a = append(a, 20)
4 | a = append(a, a[-1] + 10)
5 | a = append(a, a[-1] + 10)
6 | a = append(a, a[-1] + 10)
7 | a = append(a, a[-1] + 10)
8 | a = append(a, a[-1] + 10)
9 | a = append(a, a[-1] + 10)
10 | a = append(a, a[-1] + 10)
11 |
12 | # kortare sätt:
13 | a = array([20, 30, 40, 50, 60, 70, 80, 90])
14 |
15 | # ännu bättre sätt:
16 | a = arange(20, 91, 10)
```

Övning 3.2 - print med lista

Vi får följande utskrift:

```
1 | 22
2 | 3
```

Facit till kapitel 4 - Grafer och diagram

Övning 4.1 - Återskapa kod efter bild

```
1 | plot(array([-3, 3]), array([4, 5]), "-o")
2 | plot(arange(-3, 4), array([6, 7, 5, 6, 5, 4, 3]), "-o")
3 | grid(True)
4 | show()
```

Övning 4.2 - Plotta given funktion

```
1 | x = arange(-20, 20.01, 0.1)
2 | y = sin(x) / x
```

```

3 | plot(x, y)
4 | grid(True)
5 | show()

```

Notera att om vi bara använder `x = arange(-20, 21)` så blir grafen fult kantig.
Därför tar vi kortare steg med steglängden 0.1.

Facit till kapitel 5 - Selektion (med if)

Övning 5.1 - input

Jag testar att skriva in mitt namn som variabeln `b`:

```

1 Ange variabeln a: 12
2 Ange variabeln b: Emil
3 Traceback (most recent call last):
4 .....
5   File "C:/Users/emil/.spyder-py3/temp.py", line 2, in
6     <module>
7       b = float(input("Ange variabeln b: "))
8 ValueError: could not convert string to float: "Emil"

```

Python säger att på rad 2 försökte vi omvandla `Emil` till ett tal med hjälp av funktionen `float`, men att det naturligtvis inte gick. (Om vi inte hade använt funktionen `float` så hade vi istället fått ett felmeddelande på rad 3 där Python skulle försökt addera `12+Emil`.)

Övning 5.2 - Kontrollera vädret (fortsättning)

```

1 svaret = input("Är det fint väder? ")
2 if svaret == "j":
3     print("Vi går på picknick!")
4 if svaret == "n":
5     print("Vi stannar inne och läser en bok")

```

Övning 5.3 - Var är det kallast?

```

1 ostersund_temp = input("Ange temperaturen i Östersund: ")
2 goteborg_temp = input("Ange temperaturen i Göteborg: ")
3 if ostersund_temp < goteborg_temp:
4     print("Det är kallast i Östersund")
5 if goteborg_temp < ostersund_temp:
6     print("Det är kallast i Göteborg")
7 if goteborg_temp == ostersund_temp:
8     print("Det är lika kallt")

```

Övning 5.4 - Felaktig if-sats

Koden har två fel. Det första felet är att vi har råkat använda tilldelningsoperatorn = istället för jämförelseoperatorn ==. Det andra felet är att vi glömde skriva ett kolon i slutet av if-raden. Vi ville förstås egentligen skriva så här:

```

1 x = 9
2 if x == 10:
3     print("den är 10!")

```

Facit till kapitel 6 - Iteration (med while)

Övning 6.1 - Tal mellan 1 och 20

```

1 tal = 1
2 while tal <= 20:
3     print(tal)
4     tal = tal + 1

```

Övning 6.2 - Tal mellan 1 och 100

```

1 tal = int(input("Ange tal: "))
2 while tal <= 100:
3     print(tal)
4     tal = tal + 1

```

Övning 6.3 - Singla slant

```

1 antal_singlingar = int(input("Ange hur många gånger du
2     vill singla slant: "))
3 i = 1
4 while i <= antal_singlingar:
5     if randint(2) == 1:
6         print("Krona")
7     else:
8         print("Klave")
i = i + 1

```

Övning 6.4 - Yatzy

```

1 i = 1
2 while i <= 5:
3     print(randint(1, 7)) # du kommer väl ihåg varför det
4     ska stå 7, inte 6?
i = i + 1

```

Övning 6.5 - Väderstationen

```

1 num_temps = int(input("Ange antal mätningar: "))
2 # skapa tom lista som kommer fyllas på med temperaturer:
3 all_temps = array([])
4 i = 1 # räknare för antal iterationer
5 total = 0 # används för att räkna ut medelvärdet
6 while i <= num_temps:
7     temperature = float(input("Ange temperaturmätning: "))
8     # lägg till mätningen i slutet av listan:
9     all_temps = append(all_temps, temperature)
10    total = total + temperature
11    i = i + 1
12 print(all_temps)
13 # räkna ut medelvärdet:
14 average = total / num_temps
15 print(average)

```

Övning 6.6 - Multiplikationstabellen

```

1 i = 1 # det ena som ska gångras med...
2 while i <= 10:
3     j = 1 # ... det andra
4     while j <= 10:
5         print(i * j)
6         j = j + 1
7     i = i + 1
8     print("---") # separator

```

Facit till kapitel 7 - Problemlösning

Övning 7.1 - Gissa talet

```

1 answer = randint(1, 101) # slumpa ett tal mellan 1 och 100
2
3 nr_guesses = 1
4 guess = int(input("Gissa ett tal mellan 1 och 100: "))
5 while guess != answer:
6     if guess < answer:
7         guess = int(input("Fel! Mitt tal är högre. Gissa
8 igen: "))
9     if guess > answer:
10        guess = int(input("Fel! Mitt tal är lägre. Gissa
11 igen: "))
11    nr_guesses = nr_guesses + 1
11 print("Rätt! Du behövde", nr_guesses, "gissningar.")

```

Övning 7.2 - Datorn gissar talet

```

1 user_input = 0 # ska hantera r, l eller h
2 min = 1
3 max = 100
4 nr_guesses=0
5
6 while user_input != "r":
7     # vi avrundar nedåt för att bara jobba med heltal:
8     guess = floor((max+min)/2)
9     print("Jag gissar på:", guess)
10    user_input = input("Är det [r]ätt? Eller är ditt tal
11    [h]ögre eller [l]ägre? ")
11    if user_input == "h":
12        min = guess
13    if user_input == "l":
14        max = guess
15    nr_guesses = nr_guesses + 1
16 print("Hurra! Jag klarade det på", nr_guesses, "försök.")

```

Facit till kapitel 8 - Övningar

Övning 8.1 - Kontrollera faktorer (Ma1c)

```

1 expected_product = 41055 # här kan Kim ändra
2 all_factors = array([3, 5, 7, 17, 23]) # här också
3 product = 1
4 i = 0
5 while i < all_factors.size:
6     product = product * all_factors[i]
7     i = i + 1
8 if product == expected_product:
9     print("Rätt")
10 else:
11     print("Fel")

```

Övning 8.2 - Delbart med 3? (Ma1c)

```

1 the_number = int(input("Skriv ett heltalet: "))
2 # blir det ingen rest om vi dividerar talet med 3?
3 if fmod(the_number, 3) == 0:
4     print("delbart")
5 else:
6     print("ej delbart")

```

Övning 8.3 - Primtal eller ej (Ma1c)

```

1 the_number = int(input("Skriv ett heltalet: "))
2 is_prime = 1
3 factor = 2
4 while factor <= sqrt(the_number):
5     # blir det ingen rest om vi dividerar?
6     # dvs, är den jämt delbar?
7     # dvs, är "factor" en faktor i "the_number"?
8     if fmod(the_number, factor) == 0:
9         is_prime = 0
10    factor += 1
11 if is_prime:
12     print("primtal")
13 else:
14     print("ej primtal")

```

Övning 8.4 - Faktorisera ett heltal (Ma1c)

```

1 remaining = int(input("Skriv ett heltal: "))
2 factor = 2
3 while remaining > 1:
4     # blir det ingen rest om vi dividerar?
5     # dvs, är den jämt delbar?
6     # dvs, är "factor" en faktor i "remaining"?
7     if fmod(remaining, factor) == 0:
8         # vi har hittat en faktor, skriv ut den
9         print(factor)
10        # efter denna faktor, vad blir kvar av talet?
11        remaining = remaining / factor
12    else:
13        factor = factor + 1

```

Övning 8.5 - Testa att programmet stämmer (Ma1c)

```

1 the_number = int(input("Skriv ett heltal: "))
2 # faktorisera:
3 remaining = the_number
4 factor = 2
5 all_factors = array([])
6 while remaining > 1:
7     if fmod(remaining, factor) == 0:
8         print(factor)
9         all_factors = append(all_factors, factor)
10        remaining = remaining / factor
11    else:
12        factor = factor + 1
13 # kontrollräkna:
14 product = 1
15 i = 0
16 while i < all_factors.size:
17     product = product * all_factors[i]
18     i = i + 1
19 if product == the_number:
20     print("Rätt")
21 else:
22     print("Fel")

```

Övning 8.6 - Funktionen randi (Ma1c, Ma2c)

Övningen behöver inget facit.

Övning 8.7 - När blir datorn långsam? (Ma1c, Ma2c)

Övningen behöver inget facit.

Övning 8.8 - Hur förändras formen? (Ma1c, Ma2c)

När vi bara gör 100 kast så blir histogrammet ofta ganska ojämnt och ”taggigt”. Men ju fler kast, desto mer antar histogrammet formen av en likbent triangel.

Övning 8.9 - Skillnad mellan två tärningar (Ma1c, Ma2c)

```

1 random_numbers = array([])
2 number_throws = 100 # den här kan man ändra
3 throw = 1
4 while throw <= number_throws:
5     dice1 = randint(1, 7)
6     dice2 = randint(1, 7)
7     difference = abs(dice1 - dice2)
8     random_numbers = append(random_numbers, difference)
9     throw = throw + 1
10    hist(random_numbers, arange(random_numbers.min()-0.5,
11                                 random_numbers.max()+1.5))
11    show()

```

Övning 8.10 - Histogramanalys (Ma1c, Ma2c)

Den vanligaste skillnaden mellan två tärningar är 1.

Övning 8.11 - Beräkna histogram själv (Ma1c, Ma2c)

```

1 # ... detta är fortsättning på koden i övningen ovan
2 # med andra ord funkar denna kod inte för sig själv
3 low = random_numbers.min()
4 high = random_numbers.max()
5 num_points_in_histogram = 1 + high - low
6 summary = repeat(0, num_points_in_histogram)
7 i = 0
8 while i < random_numbers.size:
9     one_number = random_numbers[i]
10    summary_index = int(one_number - low)
11    summary[summary_index] = summary[summary_index] + 1
12    i = i + 1
13 xaxis = arange(low, high+1)
14 plot(xaxis, summary, "-o")
15 show()

```

Övning 8.12 - Chans att slå Yatzy på ett slag (Ma1c)

Rätt svar är ca 0,077% men det krävs väldigt många spelomgångar för att få ett bra närmevärde.

```

1 total_nr_yatzy = 0 # hur många gånger vi fått yatzy
2 nr_games = 100000 # hur många spel vi vill köra
3 game = 0
4 while game < nr_games:
5
6     # slå en tärning:
7     first = randint(1, 7)
8     # slå fyra tärningar till, se om alla blir samma:
9     all_are_same = 1
10    die = 2 # för vi har redan slagit tärning 1
11    while die <=5:
12        if randint(1, 7) != first:
13            all_are_same = 0
14        die = die + 1 # gå vidare till nästa tärning
15
16    if all_are_same == 1: # blev det yatzy?
17        total_nr_yatzy = total_nr_yatzy + 1
18    game = game + 1 # för att gå vidare till nästa spel
19
20 # skriv ut vårt närmevärde till sannolikheten för yatzy,
21 # i procent:
22 print(100 * total_nr_yatzy / nr_games)

```

Övning 8.13 - Spara tärningar i Yatzy (Ma1c, Ma2c)

```

1 dices = [1, 4, 5, 4, 3] # lista med våra fem tärningar
2
3 save_eyes = 0 # vilket tärningsvärdet vi slog flest av
4 nr_save = 0 # antal tärningar med det värdet
5
6 # iterera sex gånger, en för varje tänkbart tärningsvärdet
7 eyes = 1
8 while eyes <= 6:
9
10    # räkna hur många tärningar som visar just detta värdet
11    counter = 0
12    die = 0
13    while die < 5:
14        if dices[die] == eyes:
15            counter = counter + 1
16        die = die + 1
17
18    # ska vi byta vilket tärningsvärdet vi ska spara?
19    if nr_save < counter:
20        nr_save = counter
21        save_eyes = eyes
22
23    eyes = eyes + 1
24 print("Bäst att spara alla tärningar med antal ögon:")
25 print(save_eyes)

```

Alternativ lösning:

```

1 dices = [1, 4, 5, 4, 3] # lista med våra fem tärningar
2 dice_histogram = repeat(0, 6)
3 die = 0
4 while die < 5:
5     eyes = dices[die]
6     dice_histogram[eyes-1] = dice_histogram[eyes-1] + 1
7     die = die + 1
8 # dice_histogram(1) är hur många 1:or vi slog
9 # dice_histogram(2) är hur många 2:or vi slog, osv
10 nr_save = 0
11 eyes = 1
12 while eyes <= 6:
13     if nr_save < dice_histogram[eyes-1]:
14         nr_save = dice_histogram[eyes-1]
15         save_eyes = eyes
16     eyes = eyes + 1
17 print(save_eyes)

```

Övning 8.14 - Chans att slå Yatzy (Ma1c, Ma2c)

Ungefär 4.6% chans.

```

1 dices = [0, 0, 0, 0, 0] # lista med våra fem tärningar
2 total_nr_yatzy = 0 # hur många gånger vi fått yatzy
3
4 nr_games = 10000 # hur många spel vi vill köra
5 game = 0
6 while game < nr_games:
7     # save_eyes håller reda på vilket tärningsvärdet vi har
8     # flest av. Här sätter vi den till -1 för vi ska behöva
9     # slå om alla tärningar första gången i ett nytt spel:
10    save_eyes = -1
11
12    # roll är en räknare som går från 1 till 3.
13    # vi itererar alltså tre gånger, en gång för varje slag:
14    roll = 1
15    while roll <= 3:
16
17        # iterera fem tärningar, slå om några eller alla:
18        die = 0
19        while die < 5:
20            # om vi inte vill spara tärningen, slå om den:
21            if dices[die] != save_eyes:
22                dices[die] = randint(1, 7)
23                die = die + 1 # gå vidare till nästa tärning
24
25        #räkna ut vilket tärningsvärdet som vi slog flest av.
26        # det är de tärningarna som vi vill spara nästa slag.
27

```

```
28 save_eyes = 0 # vilket tärningsvärdet vi slog flest av
29 nr_save = 0 # antal tärningar med det värdet
30
31 # iterera 6 gånger, ett varv för varje
32 # tänkbart tärningsvärdet
33 eyes = 1
34 while eyes <= 6:
35
36     # räkna hur många tärningar som visar just detta
37     # värde
38     counter = 0
39     die = 0
40     while die < 5:
41         if dices[die] == eyes:
42             counter = counter + 1
43             die = die + 1
44
45     # ska vi byta vilket tärningsvärdet vi ska spara?
46     if counter > nr_save:
47         nr_save = counter
48         save_eyes = eyes
49
50     eyes = eyes + 1
51
52     roll = roll + 1 # för att gå vidare till nästa slag
53
54     if nr_save == 5: # blev det yatzy?
55         total_nr_yatzy = total_nr_yatzy + 1
56     game = game + 1 # för att gå vidare till nästa spel
57
58 # skriv ut vårt närmevärde till sannolikheten för yatzy,
59 # i procent
60 print(100 * total_nr_yatzy / nr_games)
```

Övning 8.15 - Standardavvikelse (Ma2c)

Medelvärde: 6.15, standardavvikelse: 3.801

```

1 my_list = array([4, 9, 10, 7.5, 8, 9, 3, 9, 4, -2])
2 nr_items = my_list.size
3
4 # räkna ut summan för alla tal i listan:
5 total = 0
6 i = 0
7 while i < nr_items:
8     total = total + my_list[i]
9     i = i + 1
10
11 # räkna ut medelvärdet:
12 mid = total / nr_items
13
14 # räkna ut standardavvikelse:
15 total = 0 # återställ total till 0
16 i = 0
17 while i < nr_items:
18     total = total + (my_list[i] - mid)**2
19     i = i + 1
20
21 std_dev = sqrt(total/ (nr_items-1))
22 # nr_items-1 pga Bessels korrektion
23
24 print(mid)
25 print(std_dev)

```

Övning 8.16 - Typvärde (Ma2c)

Typvärde: 9

```

1 my_list = array([4, 9, 10, 7.5, 8, 9, 3, 9, 4, -2])
2 nr_items = my_list.size
3
4 nr_most_common = 0 # antal element med typvärdet
5
6 i = 0
7 while i < nr_items:
8
9     # räkna antal element med "det här" värdet:
10    counter = 0
11    j = 0
12    while j < nr_items:
13        if my_list[j] == my_list[i]:
14            counter = counter + 1
15        j = j + 1
16
17    # ska vi ersätta det förra typvärdet?

```

```

18     if nr_most_common < counter:
19         nr_most_common = counter
20         most_common = my_list[i]
21
22     i = i + 1
23
24 print("Typvärde: ")
25 print(most_common)

```

Övning 8.17 - Inbyggda funktioner för lägesmått och spridningsmått (Ma2c)

```

1 my_list = array([4, 9, 10, 7.5, 8, 9, 3, 9, 4, -2])
2 print(mean(my_list)) # medelvärde
3 print(std(my_list)) # standardavvikelse
4 from scipy.stats import mode
5 print(mode(my_list).mode[0]) # typvärde

```

Övning 8.18 - Enkel numerisk lösning (Ma1c)

```

1 x = -10
2 while x <= 10:
3     if 3 * x - 7 == 5:
4         print(x)
5     x = x + 1

```

Övning 8.19 - Enkel numerisk lösning, svaghet 1 (Ma1c)

Vi ändrar på rad 3 i föregående kodstycke till `if 3 * x - 7 == 29:`. Programmet misslyckas för att lösningen då är 12, vilket ligger utanför intervallet -10 till 10 som ju är de värden vi testar. Den lättaste fixen är att testa ett större intervall, till exempel -1000 till 1000.

Övning 8.20 - Enkel numerisk lösning, svaghet 2 (Ma1c)

Vi ändrar på rad 3 i föregående kodstycke till `if 3 * x - 7 == 4:`. Lösningen på $3x - 7 = 4$ är $x=3,666\dots$ men programmet kan bara hitta heltalslösningar. När datorn testar $x=3$ så blir vänsterledet lika med 2, och när datorn i nästa varv testar $x=4$ så blir vänsterledet lika med 5. Vänsterledet "hoppar" alltså direkt från 2 till 5, och hoppar över rätt svar. Det finns inget jättelätt sätt att fixa programmet - vi behöver byta till en mer avancerad metod.

Övning 8.21 - Lös linjär ekvation med `fsolve` (Ma1c)

$$x = -6$$

Först skriver vi om ekvationen från $4x + 15 = -9$ till $4x + 15 + 9 = 0$ så att högerledet är en nolla, sen ger vi vänsterledet till `fsolve`:

```

1 from scipy.optimize import fsolve
2 print(fsolve(lambda x: 4*x+15+9, 0))

```

Övning 8.22 - Klurig numerisk lösning med intervallhalvering (Ma1c)

Precis som i ”gissa talet”-spelet så är det smart att gissa mitt emellan två tal. Sedan testar vi vår gissning x_{med} genom att mata in den i ekvationen, och kolla om vi hamnade *under* eller *över* 4. Den grundläggande strategin brukar kallas *binär sökning* eller *intervallhalvering*, och används som lösning på en mängd olika problem inom programmering.

```

1 xmin = -10 # dessa måste sättas manuellt,
2 xmax = 10 # så att de "omfamnar" svaret
3 i = 0
4 while i < 20: # ju fler iterationer desto mer exakt svar
5     xmed = (xmin + xmax) / 2 # gissa mitt emellan
6     # testa ekvationens (snarare olikhetens) värde
7     # i tre punkter:
8     ymin = 3 * xmin - 7 < 4
9     ymax = 3 * xmax - 7 < 4
10    ymed = 3 * xmed - 7 < 4
11    # kontrollera att vi är på vardera sidan om rätt svar:
12    if ymin == ymax:
13        print("fel! välj bättre xmin och xmax. Starta om")
14        i = 10000 # avbryt loopen i förtid
15    # välj vilken halva vi ska söka vidare i:
16    if ymin != ymed:
17        xmax = xmed
18    if ymax != ymed:
19        xmin = xmed
20    i = i + 1
21 print(xmin)
22 print(xmax)

```

Det här programmet är för övrigt ett bra exempel på när det skulle löna sig att skapa våra egna funktioner i Python. Med en egen funktion skulle vi kunna slippa upprepa nästan samma ekations-kod tre gånger. Om du har lust och tid över, sök information om det på nätet, t.ex. på <https://www.learnpython.org/en/Functions>

Övning 8.23 - Lös andragradsekvation med roots (Ma2c)

- `roots([6, -13, 5])` ger $x_1 = 1.66667$ och $x_2 = 0.5$
- `roots([4, -2, -6])` ger $x_1 = 1.5$ och $x_2 = -1$
- $2/x - 18 = 0$ är inte en polynomekvation, går inte att lösa med `roots`

- `roots([3, -2, 0])` ger $x_1 = 0.66667$ och $x_2 = 0$

Övning 8.24 - Klurig numerisk lösning, del 2 (Ma2c)

Samma program som i Övning 8.22, förutom att vi ändrar startvärderna `xmin` och `xmax`, samt ändrar ekvationen (olikheten). Programmet hittar lösningen $x = 14.667$

```

1 xmin = -50
2 xmax = 50
3 i = 1
4 while i <= 20:
5     xmed = (xmin + xmax) / 2
6     ymin = xmin**6 - sin(xmin) - 3**xmin + 7 < 0
7     ymax = xmax**6 - sin(xmax) - 3**xmax + 7 < 0
8     ymed = xmed**6 - sin(xmed) - 3**xmed + 7 < 0
9     if ymin == ymax:
10        print("fel! välj bättre xmin och xmax. Starta om")
11        i = 10000 # avbryt loopen i förtid
12     if ymin != ymed:
13        xmax = xmed
14     if ymax != ymed:
15        xmin = xmed
16     i = i + 1
17 print(xmin)
18 print(xmax)

```

Övning 8.25 - Klurig numerisk lösning, del 3 (Ma2c)

Nej, lösningsförslaget i den här boken klarar inte att lösa den ekvationen. Intervallhalverings-metoden bygger ju på att `ymin` och `ymax` ska vara på varsin sida rätt svar, så metoden kräver alltså att kurvan korsar linjen $y = 0$. Men polynomfunktionen $y = x^2 + 5x - 6$ bara nuddar vid linjen utan att korsa den! Det finns dock andra numeriska metoder som klarar att lösa sådana ekvationer.

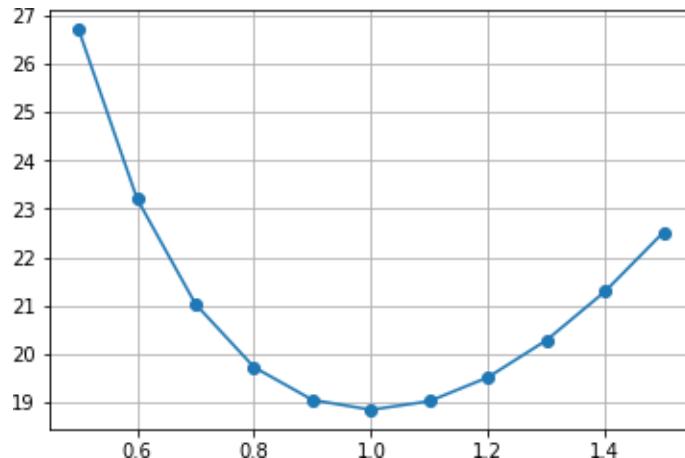
Övning 8.26 - Andragradsfunktionens minsta värde, grafiskt (Ma2c, Ma3c)

Radie=1 dm, höjd=2 dm

```

1 radius = arange(0.5, 1.51, 0.1)
2 volume = 6.2832
3 height = volume / (pi * (radius**2))
4 area = 2 * pi * radius * (radius + height)
5 plot(radius, area, "-o")
6 grid(True)
7 show()

```

Figur 9.1: Grafen för ovanstående kod**Övning 8.27 - Exponentialfunktion (Ma1c, Ma2c)**

6727.5 biljetter (troligen avrundat uppåt eller nedåt).

```

1 tickets = array([1000]) # ett element per år
2 yr = 1
3 while yr <= 20:
4     tickets = append(tickets, tickets[-1] * (1 + 10/100))
5     yr = yr + 1
6 print(tickets[-1])
7 plot(tickets, "-o")
8 grid(True)
9 show()

```

Övning 8.28 - Numerisk lösning av exponentialekvation (Ma1c, Ma2c)

Den 18:e festivalen.

```

1 tickets = array([1000]) # ett element per år
2 yr = 1
3 while tickets[-1] < 5000:
4     tickets = append(tickets, tickets[-1] * (1 + 10/100))
5     yr = yr + 1
6 print(yr)

```

Övning 8.29 - Oregelbunden exponentialfunktion (Ma1c, Ma2c)

Den 26:e festivalen har 8110.5 biljetter (troligen avrundat uppåt eller nedåt).

```

1 nr_years = 25
2 changes = array([])
3 # changes innehåller ökningen i procent till nästa
   festival
4 # dvs, changes[0] är ökningen från tickets[0] till
   tickets[1]
5 # sätt de två första ökningarna:
6 changes = append(changes, 4)
7 changes = append(changes, 2)
8 # räkna ut alla andra ökningar enligt mönstret:
9 yr = 2
10 while yr < nr_years:
11     changes = append(changes, changes[yr-2] + 1)
12     yr = yr + 1
13
14 tickets = array([1000]) # ett element per år
15 yr = 1
16 while yr <= nr_years:
17     change = (1 + changes[yr-1] / 100)
18     tickets = append(tickets, tickets[yr-1] * change)
19     yr = yr + 1
20 print(tickets[-1])
21 plot(tickets, "-o")
22 grid(True)
23 show()

```

Övning 8.30 - Oregelbunden exponentialfunktion, del 2 + slumpförsök och statistik (Ma1c, Ma2c)

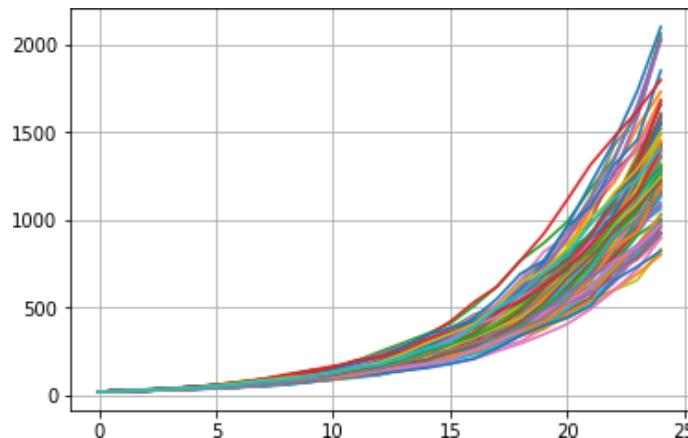
ungefärligen 1300 kaniner.

```

1 estimates = array([])
2 total = 0
3 i = 1
4 while i <= 100:
5     rabbits = array([20]) # ett element per månad
6     mon = 0
7     while mon < 24:
8         change = 1 + randint(10, 29) / 100
9         rabbits = append(rabbits, rabbits[mon] * change)
10        mon = mon + 1
11    plot(rabbits)
12    estimates = append(estimates, rabbits[-1])
13    total = total + rabbits[-1]
14    i = i + 1
15
16 show()
17 # skriv ut medelvärde
18 print("Ca antal kaniner: ")
19 print(mean(estimates))

```

Figur 9.2



Övning 8.31 - Avbetalning av annuitetslån (Ma1c, Ma2c)

```

1 # konstanter:
2 interest = 5 # månadsränta i procent
3 amort_per_month = 1087.07
4
5 # listor:
6 months = array([0])
7 sum_payed = array([0])
8 months_twice = array([0])
9 debt = array([15000])

```

```

10
11 while debt[-1] > 0:
12     month = months[-1] + 1
13
14     months = append(months, month)
15     new_sum_payed = sum_payed[-1] + amort_per_month
16     sum_payed = append(sum_payed, new_sum_payed)
17
18     months_twice = append(months_twice, month)
19     debt = append(debt, debt[-1] * (1 + interest / 100))
20     months_twice = append(months_twice, month)
21     debt = append(debt, debt[-1] - amort_per_month)
22
23 plot(months, sum_payed, "-o")
24 plot(months_twice, debt)
25 grid(True)
26 show()

```

Övning 8.32 - Försämplingshastighet enligt tabell (Ma3c)

Med hjälp av följande kod:

```

1 x = array([2007, 2008, 2009, 2010, 2011, 2012, 2013,
2   2014, 2015, 2016, 2017])
3 y = array([383.79, 385.60, 387.43, 389.90, 391.65,
4   393.85, 396.52, 398.65, 400.83, 404.21, 406.53])
5
6 index = 9 # det 10:e mätvärdet i listan
7 result = (y[index+1] - y[index-1]) / 2
8 print(result)

```

Får vi resultatet 2.85 ppm/år.

Övning 8.33 - Försämplingshastighet enligt tabell, del 2 (Ma3c)

```

1 x = array([2007, 2008, 2009, 2010, 2011, 2012, 2013,
2   2014, 2015, 2016, 2017])
3 y = array([383.79, 385.60, 387.43, 389.90, 391.65,
4   393.85, 396.52, 398.65, 400.83, 404.21, 406.53])
5
6 year = int(input("ange år mellan 2008 och 2016: "))
7
8 # hitta index för årtal det användaren vill se:
9 index = argwhere(x == year)[0][0]
10 # räkna ut den centrala differenskvoten kring det året
11 result = (y[index+1] - y[index-1]) / 2
12 print(result)

```

Övning 8.34 - Ändringskvot för funktion (Ma3c)

Ett närmevärde till derivatan är: -0.19048

```

1 x = float(input("ange x-värde att beräkna derivatan för:
2   "))
3 h = 0.4
4 x_left = x - h
5 x_right = x + h
6 y_left = (x_left**2 + 1) / x_left
7 y_right = (x_right**2 + 1) / x_right
8 estimate = (y_right - y_left) / (2 * h)
9 print("ett närmevärde till derivatan är: ")
10 print(estimate)

```

Övning 8.35 - Testa olika värden för h (Ma3c)

Följande kod:

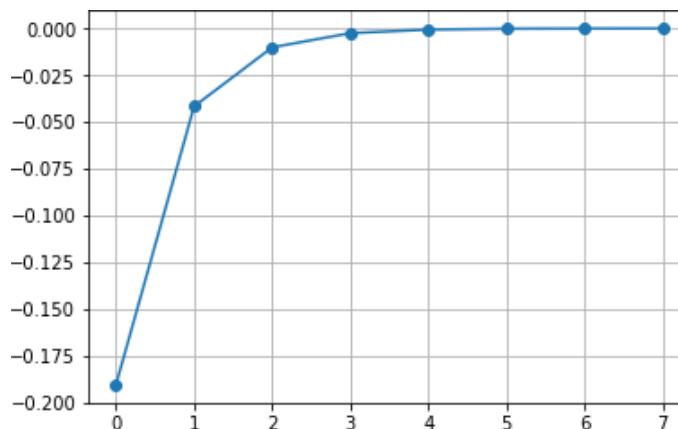
```

1 estimates = array([])
2 x = float(input("ange x-värde att beräkna derivatan för:
3   "))
4 h = 0.4
5 num_halvings = 1
6 while num_halvings <= 8:
7     x_left = x - h
8     x_right = x + h
9     y_left = (x_left**2 + 1) / x_left
10    y_right = (x_right**2 + 1) / x_right
11    estimate = (y_right - y_left) / (2 * h)
12    print(estimate)
13    estimates = append(estimates, estimate)
14    h = h / 2
15    num_halvings = num_halvings + 1
16 plot(estimates, "-o")
17 grid(True)
18 show()

```

Ger grafen (se nästa sida...):

Figur 9.3



I grafen ser det ut som att den exakta derivatan är: 0

Övning 8.36 - Testa den inbyggda gradient (Ma3c)

Följande kod ger approximationen $f'(1) = -2.5$

```

1 xp = 1 # i vilken punkt vill vi veta lutningen
2 h = 0.001
3 x = array([xp - h, xp, xp + h])
4 y = 3 * sqrt(x) + 2 / (x**2) # vår funktion
5 estimate = gradient(y, h)[1]
6 print(estimate)

```

Övning 8.37 - Approximera talet e (Ma3c)

Följande kod ger approximationen $e = 2.7183$

```

1 emin = 0 # dessa måste sättas manuellt,
2 emax = 10 # så att de "omfamar" svaret.
3
4 # för ändringskvot. mindre h ger exaktare svar:
5 h = 0.00001
6 i = 0
7 while i <= 40: # ju fler iterationer desto exaktare svar
8     emed = (emin + emax) / 2 # gissa mitt emellan
9     # beräkna ändringskvoten kring emed^1:
10    slope = (emed**(1+h) - emed**(1-h)) / (2 * h)
11    # vi önskar att "slope" ska bli exakt lika med emed,
12    # så välj vilken halva vi ska söka vidare i:
13    if slope > emed:
14        # vår gissning var för hög, sök i nedre halvan
15        emax = emed
16    else:
17        # vår gissning var för låg, sök i övre halvan
18        emin = emed
19    i = i + 1

```

```
20 | if abs(slope - emed) < 0.001:  
21 |     print("Talet e är ungefär = ")  
22 |     print(emed)  
23 | else:  
24 |     print("Hittade inte något tal e mellan emin och emax.")  
25 |     print("Ändra startvärdet och försök igen.")
```


10

S A K R E G I S T E R

abs, **9**
addition, **2**
arccos, **7**
arcsin, **7**
arctan, **7**
arcus cosinus, **7**
arcus sinus, **7**
arcus tangens, **7**
argument, **5**
aritmetik, **2**
array, **19**
avrundning, **8**
binär sökning, **91**
ceil, **8**
cos, **7**
cosinus, **7**
decimaltal, **2**
diagram, **27**
division, **2**

e, **7**, **76**
element, **19**
else, **38**
end, **23**

floor, **8**
fmod, **5**
funktioner, **5**
grafer, **27**
histogram, **32**
if, **36**
index, **19**
indexering (av listor), **21**
input, **35**
intervallhalvering, **49**, **66**, **91**
iteration, **41**

jämförelseoperatorer, **3**, **4**, **39**
kommentarer, **1**
kvadratrot, **5**
listor, **19**
log, **7**
logaritmer, **7**

max, **5**
mean, **64**
medelvärde, **64**
min, **5**
mode, **64**

- multiplikation, **2**
- operatorer, **3**
- pi, **7**
- plot, **27**
- potenser, **6**
- print, **14**
- problemlösning, **47**
- randint, **8**
- round, **8**
- selektion, **35**
- sin, **7**
- sinus, **7**
- size, **22**
- slumptal, **8**
- sqrt, **5**
- standardavvikelse, **64**
- std, **64**
- subtraktion, **2**
- tan, **7**
- tangens, **7**
- tilldelningsoperatorn **=**, **12**
- trigonometri, **7**
- typvärde, **64**
- variabler, **11**
- vektor, **19**
- while, **41**