

Introduktion till matematisk modellering och databehandling i Python

Per Jönsson och Stefan Gustafsson

Institutionen för materialvetenskap och tillämpad matematik
Malmö universitet



Juni 2024

Omslagsbild: Flock med jagande grotteljona målade i Chauvetgrottan (Ardèche, Frankrike). Mållingen kan vara så gammal som 38 000 år och visar människans skapande kraft.

Innehåll

1 Matematisk modellering och databehandling – överblick	7
1.1 Modellering och simulering	7
1.2 Olika typer av modeller och simuleringar	8
1.3 Databehandling – medelvärde och histogram	8
1.4 Databehandling – minstakvadratanpassningar	9
1.5 Formulera modeller	9
1.6 Vad behöver man kunna och i vilken ordning?	12
I Python	13
2 Introduktion till Python	15
2.1 Installera Python	15
2.2 Spyder	15
2.3 Moduler	16
2.4 Python som miniräknare	17
2.5 Script och py-filer	17
2.6 Automatisk komplettering	19
2.7 Olika typer av tal	20
2.8 Räkning med tal	20
2.9 Funktioner	22
2.10 Lambda-funktioner	23
2.11 Organisera ditt arbete	24
3 Variabler, datatyper och tilldelningssatser	25
3.1 Datatyper	25
3.2 Tilldelningssatser	25
3.3 Tilldelningssatser i tekniska termer	26
3.4 Listor, tupler och strängar – indexering	28
3.5 Delområden av listor, tupler och strängar	30
3.6 Klasser – instans, attribut, metod	31
3.7 Metoder för listor	32
3.8 Metoder för strängar	34
4 Variabelnamn, inläsning och utskrift	37
4.1 Variabelnamn, variabelutforskaren	37
4.2 Inläsning och utskrift till skärmen	38
4.3 Läsa och skriva variabler till fil	39
4.4 Tillämpning: fönsterbyte	39

5 Vektorer och matriser	43
5.1 NumPy	43
5.2 Fält	43
5.3 Fält – attribut	44
5.4 Vektorer	44
5.5 Arange och linspace	46
5.6 Delvektorer	47
5.7 Matriser	48
5.8 Noll- och ettmatrimer	49
5.9 Slumpmatriser	50
5.10 Delmatriser	51
5.11 Läsa och skriva variabler till fil	52
5.12 Aritmetiska operationer på fält	55
5.13 Elementvisa funktioner	57
5.14 Aggregerings- och lokaliseringsfunktioner	58
5.15 Tillämpning: temperaturdata	60
6 Plotting och grafik	63
6.1 Matplotlib – pyplot	63
6.2 Exempelgalleriet	64
6.3 Skapa och spara en figur	65
6.4 Grundläggande plottmetoder	66
6.5 Axlar och skalning	69
6.6 Text och teckenförklaring	72
6.7 Pilar och förklarande text	74
6.8 Histogram och stolpdiagram	75
6.9 Bilder	77
6.10 Matriser och bilder	79
6.11 Plotta som i matteboken	81
7 Programmering	83
7.1 Logiska uttryck	83
7.2 If-satser	84
7.3 While-loopar	87
7.4 For-loopar	92
7.5 Nästlade loopar	94
7.6 Tillämpning: signalbehandling	96
7.7 Tillämpning: bildbehandling	97
7.8 Tillämpning: bildbehandling – ändra färger	99
7.9 Tillämpning: bildbehandling – avskogning av Amazonas	101
8 Programstruktur	105
8.1 Program, funktioner och moduler	105
8.2 Funktioner	105
8.3 Anrop av funktioner	107
8.4 Funktioner – odefinierade lokala variabler	110
8.5 Funktionsnamn som invariabler	110
II Databehandling och modellering	113
9 Databehandling	115
9.1 Summasymbolen	115
9.2 Medelvärde, frekvenstabell och histogram	117

9.3 Standardavvikelse	122
9.4 Median	124
9.5 Slumpmässiga försök – normalfördelning	124
9.6 Normalfördelade slumptal	126
9.7 Tillämpning: känslighetsanalys	127
9.8 Tillämpning: histogram från frekvenstabeller	129
10 Funktionsklasser: polynom och rationella funktioner	133
10.1 Inledande exempel	133
10.2 Linjära funktioner	134
10.3 Den räta linjens ekvation	136
10.4 Proportionalitet	137
10.5 Polynom och polynomfunktioner	138
10.6 Polynomfunktioner av andra graden	139
10.7 Polynomfunktioner av tredje graden	141
10.8 Rationella funktioner	142
11 Funktionsklasser: potens- och exponentialfunktioner	145
11.1 Potenser	145
11.2 Potenser med heltalsexponent	145
11.3 Potenser med rationella exponenter	146
11.4 Potenser med reella exponenter	148
11.5 Beräkning av potenser på dator	148
11.6 Potensfunktioner	148
11.7 Exponentialfunktioner	151
11.8 Funktionen $y = e^x$	153
11.9 Normalfördelning	154
11.10 Tillämpning: befolkningstillväxt	155
11.11 Tillämpning: radioaktivt nedfall	156
12 Funktionsklasser: periodiska funktioner	159
12.1 Periodiska funktioner	159
12.2 Cosinus- och sinusfunktioner	160
12.3 Allmänna cosinus- och sinusfunktioner	161
12.4 Addition av cosinus- och sinusfunktioner	162
12.5 Tillämpning: medeltemperatur	163
13 Ekvationer	165
13.1 Bestämning av x som hör till givet funktionsvärde.	165
13.2 Inbyggda funktioner för ekvationslösning	166
13.3 Egen funktion för ekvationslösning	167
13.4 Tillämpning: jordens befolkning exponentiell modell	169
14 Minstakvadratanpassningar	171
14.1 Minstakvadratanpassningar	171
14.2 Polynom som modelfunktioner	172
14.3 Allmänna modelfunktioner	174
14.4 Tillämpning: jordens befolkning logistisk modell	177
14.5 Tillämpning: CO ₂ i atmosfären	179

15 Förändring	183
15.1 Tangenten till en kurva	183
15.2 Derivator	185
15.3 Viktiga deriveringsregler	186
15.4 Om derivatans definition	186
15.5 Om derivatans beteckning	187
15.6 Gridd och finita differensapproximationer	188
15.7 Tillämpning: befolkningstillväxt	190
16 Dynamiska modeller	193
16.1 Differentialekvationer	193
16.2 Differentialekvationer – stegmetoder	194
16.3 Lösning av differentialekvationer i Python	195
16.4 Anpassning av modellparametrar	197
16.5 Modeller för populationsutveckling	199
16.5.1 Obegränsad tillväxt	199
16.5.2 Täthetsberoende konkurrens	200
16.5.3 Täthetsberoende konkurrens – bestämning av modellparametrar	202
16.6 System av differentialekvationer	204
16.7 Lösning av system av differentialekvationer i Python	206
16.8 Tillämpning: smittspridning	209
16.8.1 SIR-modellen	209
16.8.2 SIR-modellen med vaccination	211
16.8.3 Flockimmunitet	212
III Modellering i perspektiv	215
17 Modellering för att rädda och bevara planeten	217
17.1 Copernicus och Sentinel	217
17.2 Klimatmodeller	218
17.3 Copernicus interaktiva klimatatlas	219
17.4 Modellering av den gröna ekonomin	219
17.5 Engagera dig	219

Kapitel 1

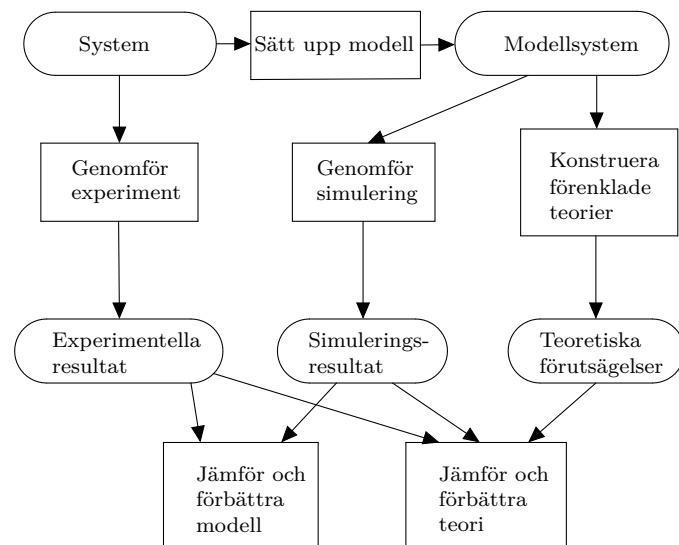
Matematisk modellering och databehandling – överblick

Det här kompendiet utgör kurslitteratur till kursen 'Introduktion till matematisk modellering och databehandling i Python'. Det känns riktigt att börja med en översikt om modellering och simulering innan vi går in på detaljer i de kommande kapitlen.

1.1 Modellering och simulering

En datorsimulering är matematisk modellering med hjälp av en dator, med målet att förutspå och förstå uppförandet av ett system. Tillförlitligheten hos en matematisk modell bestäms genom att jämföra simuleringens resultaten med observationer. Datorsimuleringar är ett viktigt verktyg för att studera naturliga system inom fysik, astronomi, klimatologi, kemi och biologi, men också mänskliga system inom ekonomi, medicin och miljövetenskap.

Man skiljer på modell och simulering. En datormodell består av matematiska uttryck och ekvationer som beskriver systemet vi vill studera. En simulering innebär att man kör ett datorprogram, som hanterar uttrycken och ekvationerna, med olika indata. Sambandet mellan modell, simulering och det verkliga systemet illustreras i figur 1.1.



Figur 1.1: *Samband mellan modell, simulering och verkligt system.*

1.2 Olika typer av modeller och simuleringar

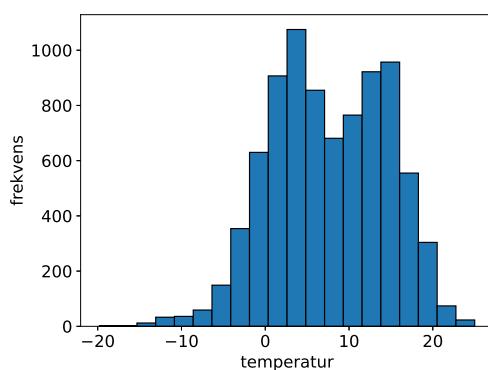
Variationen hos de system man vill studera är stor, och det finns flera olika typer av modeller och simuleringar. Vid stokastisk modellering beaktar man osäkerheter i indata som beror på slumpmässiga variationer. Snarare än att använda bestämda (deterministiska) värden på ingående variabler, inkluderar stokastisk modellering slumpmässiga variationer i beräkningarna. Genom upprepade simuleringar får man till slut ett stabilt interval, inom vilket man kan förvänta sig att resultatet ligger. Stokastiska modeller är mycket vanliga, och används till exempel inom ekologin för att studera hur sårbar en djurpopulation är för variationer i födotillgång och häckningsresultat. Stokastiska modeller kan användas inom hydrologen för att studera hur grundvattentillgången varierar givet slumpmässiga nederbördsvariationer.

Utöver stokastiska modeller har vi deterministiska modeller, för vilka slumpen inte beaktas. För de senare kan vi skilja mellan statiska och dynamiska modeller. I statiska modeller beräknar vi ett jämviktsläge för systemet. Genom upprepade simuleringar kan vi få en uppfattning om hur detta jämviktsläge beror på olika modellparametrar.

I dynamiska modeller är vi intresserade av hur ett system utvecklas över tid. Dynamiska modeller är oftast baserade på ekvationer, vilka kopplar ihop förändring av systemvariabler med variabelvärdena själva. Det finns hur många exempel på dynamiska modeller som helst. Inom byggnadsläran vill vi kanske ta reda på hur fort en byggnad kyls av och hur denna process påverkas av yttertemperaturen och hur byggnaden är isolerad. Inom epidemiologi vill man till exempel ta reda på hur en sjukdom överförs till olika grupper av människor och hur spridningsförlopp påverkas av vaccinering.

1.3 Databehandling – medelvärde och histogram

Grunden för all modellering är att vi har förståelse för data som modellen avser att beskriva. Ofta beskrivs data genom att man anger medelvärdet, men också spridningen kring medelvärdet, den så kallade standardavvikelsen. Mer information om data kan man få genom ett histogram. Om vi t.ex. ska göra en modell för hur temperaturen kommer att öka i en region beroende på klimatförändringar kan vi börja med att ta fram medeltemperatur över året. Vi kan också presentera temperaturdata över året i form av ett histogram, se figur 1.2. Genom att jämförande nuvarande temperaturdata med data tillbaka i tiden kan vi ta fram en trend som kan tala om hur temperaturerna kan se ut framåt i tiden. Vi kan också få en uppfattning om hur antalet dagar med mycket kalla respektive mycket varma temperaturer kommer att ändras. Medelvärdet, standardavvikelsen och histogram hanteras med inbyggda rutiner i Python, se kapitel 5 och 6.



Figur 1.2: Histogram över den dagliga temperaturen på en ort i Sverige under 23 år.

1.4 Databehandling – minstakvadratanpassningar

I vissa fall vi kan formulera en modell i form av en funktion som beskriver eller förklarar data. Modellen, eller modelfunktionen, beror ofta på parametrar, vilka kan bestämmas med hjälp av minstakvadratanpassningar (regression). Den anpassade modelfunktionen användas sedan för att göra förutsägningar, extrahera information och så vidare. I tabell 1.1 har vi data från ett experiment där ingenjörsstudenter vid Malmö universitet släppte en bordtennisboll från olika höjder och mätte motsvarande studshöjd i cm.

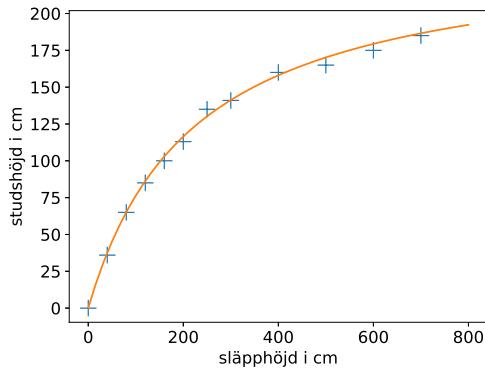
Tabell 1.1: Studshöjd i cm som funktion av släpphöjd i cm.

släpp	0	40	80	120	160	200	250	300	400	500	600	700
studs	0	36	65	85	100	113	135	141	160	165	175	185

Under antagandet att studshöjden är proportionell mot släpphöjden för låga släpphöjder, men måste plana ut mot ett konstantt värde, tog vi en modelfunktion av formen

$$f(x) = \frac{a_0 x}{1 + a_1 x}.$$

Parametrarna a_0 och a_1 bestämdes sedan med hjälp av minstakvadratanpassning till data. Detta görs enklast med hjälp av inbyggda rutiner i Python, och vi återkommer till detta i kapitel 14. Data och den anpassade funktionen visas i figur 1.3. Från den anpassade funktionen kan vi skatta den maximala studshöjden till 240 cm.

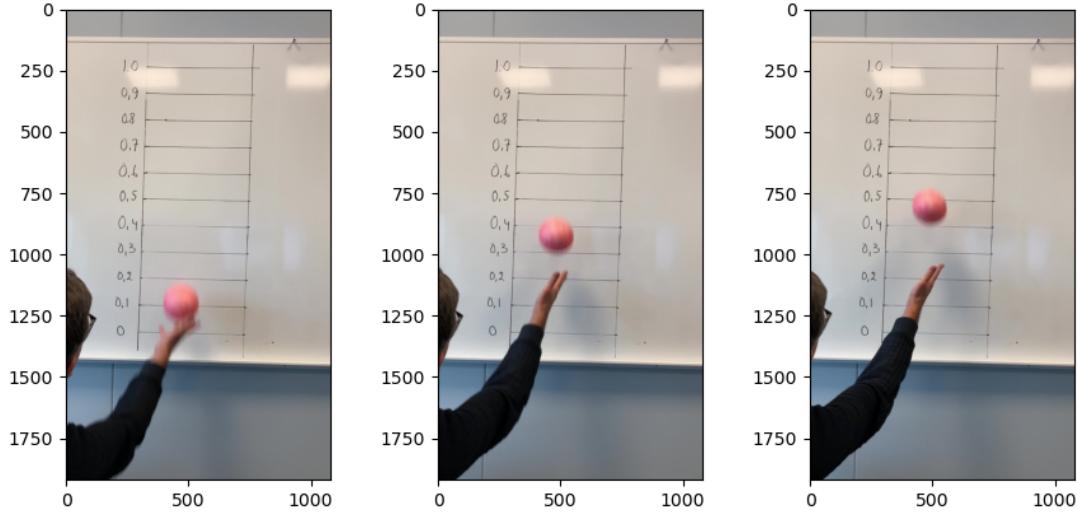


Figur 1.3: Minstakvadratanpassning av en modelfunktion till experimentella data. Den anpassade funktionen kan användas för att besvara frågor i relation till det fenomen eller den process vi studerar.

1.5 Formulera modeller

Inom många fält är funktionsformen, vilken beskriver en process eller ett skeende, känd från någon bakomliggande teori. Vår uppgift kan då bli att bekräfta att funktionsformen genom att göra minstakvadratanpassningar till data och bestämma eventuella parametrar. I dynamiska modeller är funktionen som beskriver skeendet en funktion av tiden t .

Exempel 1.1. Vi ställer oss vid en whiteboardtavla, där vi har ritat ett koordinatsystem. Vi kastar en boll uppåt samtidigt som vi filmar med mobiltelefonen. Från filmen drar vi sedan ut en sekvens av bilder som visar bollens position vid olika tider, se figur 1.4.



Figur 1.4: Vi kastar en boll och filmar och drar ut en sekvens av bilder. Genom att mäta bollens position på de olika bilderna får vi data som anger bollens höjd som funktion av tiden.

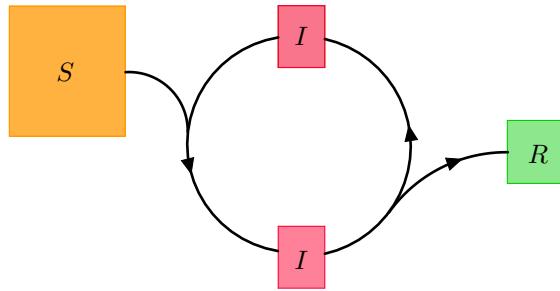
Med användning av Newtons lagar får vi att bollens position som funktion av tiden t ges av funktionen

$$y = y_0 + v_0 t - \frac{g}{2} t^2,$$

där y_0 och v_0 är bollens position och hastighet vid tiden $t = 0$. Konstanten g är tyngdaccelerationen $g = 9.81 \text{ m/s}^2$. Vi gör en minstakvadratanpassning av funktionen till de uppmätta positionerna, vilket bestämmer y_0 och v_0 . Vi kan sedan beräkna bollens position för en godtycklig tid. \square

I många fall känner vi inte formen på den eller de funktioner som beskriver ett skeende eller process för ett system, men kan göra rimliga antagande om hur förändringen per tidsenhet beror av funktionerna. Vi kan då sätta upp så kallade differentialekvationer, vilka, givet kännedom om systemet vid tiden $t = 0$, låter oss bestämma egenskaper hos systemet vid senare tider t . Det är ju fantastiskt, vi kan se in i framtiden!

Exempel 1.2. För att modellera spridning av covid kan man använda sig av SIR-modellen, läs vidare på https://en.wikipedia.org/wiki/Compartmental_models_in_epidemiology. Här delar man in befolkningen i tre grupper: de som är mottagliga (Susceptible) för sjukdomen, de som är infekterade (Infected) och kan infektera andra och de som har återhämtat sig (Recovered) och som är immuna mot sjukdomen. Vi kan tänka oss att befolkningen består av N personer och att vid tiden $t = 0$, när vi börjar studera smittspridningen, har vi ett stort antal mottagliga (Susceptible), S , ett litet antal infekterade (Infected), I och inga återhämtade (Recovered), R . Smittspridningen går till som i figur 1.5.

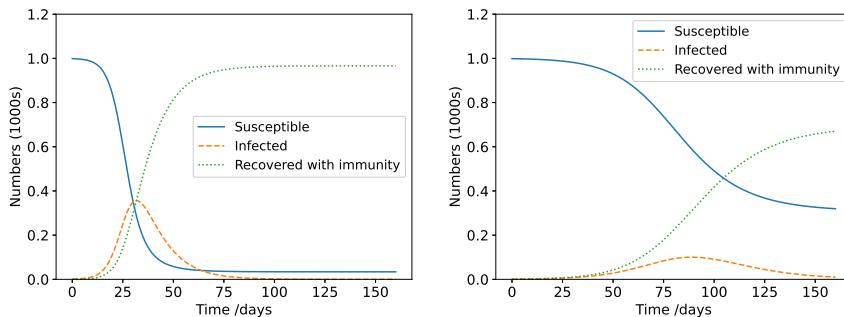


Figur 1.5: *Smittspridning enligt SIR-modellen*. Genom möte mellan mottagliga (Susceptible), S , och infekterade (Infected), I , minskar antalet mottagliga samtidigt som antalet infekterade ökar. Antalet infekterade minskar genom att folk återhämtar sig (Recover), varvid antalet återhämtade R hela tiden ökar.

När infekterade möter mottagliga sprider sig sjukdomen och antalet infekterade ökar samtidigt som gruppen av mottagliga minskar. Det är rimligt att anta att minskning per tidsenhet av antalet mottagliga är proportionell mot SI/N (antal möten mottagliga och infekterade relativt det totala antalet personer). Antalet infekterade måste då öka i motsvarande grad, dvs. ökningen per tidsenhet av antalet infekterade är proportionell mot SI/N . Antalet infekterade minskar också eftersom en del av dem återhämtar sig. Det är rimligt att anta att det leder till en minskning per tidsenhet av antalet infekterade som är proportionell mot antalet infekterade (ju fler infekterade där är desto fler återhämtar sig per tidsenhet). Att antalet infekterade minskar måste innebära att antalet återhämtade ökar med lika många per tidsenhet. Formulerat i matematiska termer leder detta till ett system av differentialekvationer

$$\begin{cases} \frac{dS}{dt} = -\beta \frac{SI}{N} \\ \frac{dI}{dt} = \beta \frac{SI}{N} - \gamma I \\ \frac{dR}{dt} = \gamma I \end{cases}$$

där $\frac{dS}{dt}$, $\frac{dI}{dt}$ och $\frac{dR}{dt}$ betecknar förändringen av antalet mottagliga, antalet infekterade och antalet återhämtade per tidsenhet. β är en konstant som bestämmer hur smittsam sjukdomen är medan γ är ett mått på hur fort infekterade återhämtar sig och blir immuna. Vi kan modellera smittspridningen genom att lösa systemet av ekvationer i Python under olika förutsättningar, se kapitel 16. Modellen, körd för tusen personer $N = 1000$ med $\beta = 0.35$ och $\gamma = 1/10$ (genomsnittlig återhämtningstid 10 dagar), ger plotten till vänster i figur 1.6. Vi ser att i stort sett hela populationen blir infekterad och får sjukdomen. Om vi till exempel genom social distansering får ner kontakthastigheten blir förloppet ett helt annat. Modellen, körd med $\beta = 0.17$ och $\gamma = 1/10$, ger plotten till höger i figur 1.6. Nu är förloppet lugnare och bara 10 % av populationen är infekterade samtidigt, vilket skulle innebära en mindre belastning för vården. Vi kan bygga ut modellen och köra simuleringar med olika grad av vaccinationstäckning. \square



Figur 1.6: Antal mottagliga, infekterade och återhämtade som funktion av tiden. Till vänster har vi kurvorna för $\beta = 0.35$ och till höger för $\beta = 0.17$. I båda fallen är $\gamma = 1/10$.

1.6 Vad behöver man kunna och i vilken ordning?

För att behandla data och modellera behöver vi kunna beräkna medelvärden och standardavvikelser. Vi behöver också veta hur vi kan få en överblick över data med hjälp av histogram eller stolpdiagram. Vidare behöver vi känna till olika funktionsklasser (typer av funktioner) och vilka egenskaper dessa funktionsklasser har. Vi behöver veta vad som menas med en funktions förändringshastighet (derivata) och hur detta begrepp används inom modellering för att sätta upp ekvationer som beskriver hur en process eller ett system förändras över tid.

Databehandling, modellering och simulering sker idag på dator med användande av olika programspråk. Det mest använda programspråket är Python och vi behöver kunna hantera olika datatyper och göra operationer på dessa med hjälp av inbyggda kommandon. Vi behöver kunna plotta och visualisera både data och anpassade funktioner. Vidare behöver vi behärska grunderna i programmering för att skriva program som kan användas för mer omfattande databehandling och simulering. Programmering och databehandling är viktiga kunskaper, både i arbetslivet och vid fortsatta studier.

Del I

Python

Kapitel 2

Introduktion till Python

Python är ett programspråk som används för vetenskapliga beräkningar, modellering och simulerings, maskininlärning, datautvinning och datavisualisering. Det är för närvarande världens mest använda programspråk.

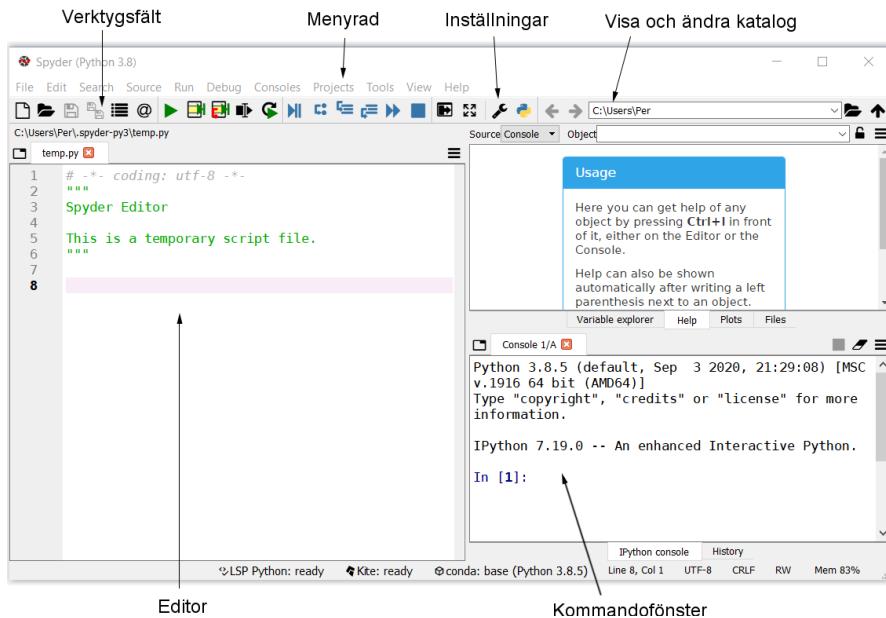
2.1 Installera Python

Vi rekommenderar att Python installeras via Anaconda Individual Edition, vilken finns för Windows, Mac och Linux (se Canvas för installationsinstruktioner). Då vi installerar Anaconda får vi bland annat Python tillsammans med utvecklingsmiljön Spyder och anteckningsboken Jupyter.

2.2 Spyder

Spyder är en integrerad utvecklingsmiljö (eng. Integrated Development Environment, IDE) för Python. Miljön innehåller ett kommandofönster (eng. IPython console), där man kan arbeta interaktivt, en editor, där man skriver sina program (script), och en variabelutforskare, där man kan se vilka variabler som är definierade och deras värden. Spyder har även verktyg för att hantera och spara plotter och felsöka program. Spyder visas i figur 2.1.

Då man startar Spyder får man ljus text mot en svart bakgrund. Om man inte tycker detta är snyggt, och vill byta till en vit bakgrund med svart text, så gör man det genom att i verktygsfältet klicka på *Inställningar* och välja *Appearance*. Under *Syntax highlighting* väljer man *Spyder* i stället för *Spyder Dark*. Det är ofta bekvämt att interaktivt kunna manipulera plotter och grafik. För att möjliggöra detta går vi till *Inställningar* och väljer *IPython console*. Under *Graphics* väljer vi *Backend: Automatic*. I en video som finns på Canvas visar vi hur man ändrar inställningar i Spyder.



Figur 2.1: *Spyder utvecklingsmiljö för Python*. Fönstret nere till höger är kommandofönstret (eng. *IPython console*), där man skriver in sina kommandon. Fönstret till vänster är editorn, där man skriver sina script.

2.3 Modularer

Vid starten av Python finns bara en begränsad mängd kommandon att tillgå. Beroende på uppgiften vi vill lösa, behöver vi importera kommandon och funktioner från olika paket eller moduler. Detta kan göras på flera olika sätt. Det finns till exempel inga matematiska funktioner då vi startar Python. För att importera kvadratrotfunktionen `sqrt` från modulen (paketet) `numpy` skriver vi

```
from numpy import sqrt
```

Vi kan nu använda kvadratroten som vanligt. För att beräkna kvadratroten ur fyra skriver vi

```
sqrt(4)
```

Det är ofta opraktiskt att importera funktioner en och en. För att importera alla funktioner från `numpy` ger vi kommandot

```
import numpy
```

Vid anropet av funktionerna måste vi nu ha med modulnamnet som ett förled och anropet blir

```
numpy.sqrt(4)
```

Vi kan också importera modulen och ge den ett förkortat namn. Det har blivit standard att `numpy` importeras som

```
import numpy as np
```

Anropet av funktionen blir nu

```
np.sqrt(4)
```

De viktigaste modulerna (paketen) är `numpy`, som innehåller de matematiska funktionerna somt kommandon för att arbeta med samlingar av tal (eng. arrayer), `scipy`, som används för att modellera och simulera, och `matplotlib`, som innehåller funktioner för plottnings och grafik. Vi återkommer till dessa modular längre fram.

2.4 Python som miniräknare

Vi kan använda Python som en miniräknare. Kommandon skrivs i kommandofönstret och utförs när returtangenten trycks ned. Vi börjar med några enkla exempel.

Exempel 2.1. För att beräkna $16 \cdot 27$ skriver vi

```
16*27
```

och trycker retur. Python skriver ut svaret

```
432
```

□

Exempel 2.2. Vi ska beräkna arean av en cirkel med radien $r = 3$. Arean ges av $A = \pi r^2$. Värdet för π finns i modulen `numpy`, som vi måste importera. Radien upphöjt till två, r^2 , skrivs som `r**2` och kommandona blir

```
import numpy as np
r = 3
np.pi*r**2           # pi skrivs som np.pi
```

och Python svarar

```
28.274333882308138
```

□

Exempel 2.3. Vi har talen 1, 2, 4, 2, 7, 8, 2, 13 och ska beräkna medelvärdet. Funktionen som beräknar medelvärde heter `mean` och finns i modulen `numpy`, som vi måste importera. Vi ger kommandona

```
import numpy as np
x = [1,2,4,2,7,8,2,13]
np.mean(x)           # beräkna medelvärdet av elementen i x
```

Python svarar

```
4.875
```

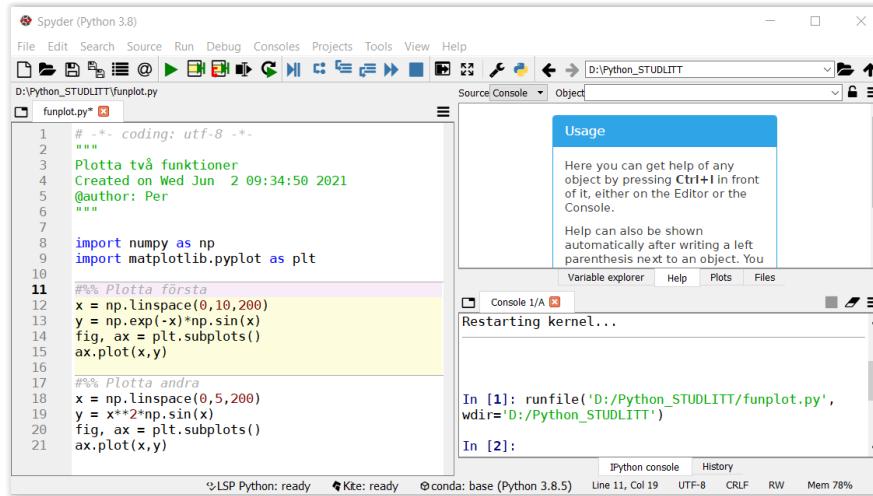
□

2.5 Script och py-filer

En serie kommandon samlade i en fil utgör ett program eller ett script. Kommandona i filen delas ofta in i mindre enheter, vilka kallas celler. En cell inleds med `#%%`. I Python har filer som innehåller kommandon alltid suffixet (extensionen) `.py` och kallas ibland py-filer.

Script och kommandofiler skapas i Spyders editor. Då man skrivit in sina kommandon i editorn, klickar man på ikonen för att *spara*, varvid Spyder ber en att ge namnet på filen och katalogen där den ska sparas. För att köra eller exekvera programmet, vilket innebär att de inskrivna kommandona utförs i ordning uppifrån och ner (sekventiellt), klickar vi på ikonen med den gröna triangeln

i Spyders verktygsfält. I stället för att exekvera hela programmet kan vi exekvera enskilda celler. Gå in i editorn och ställ dig i den aktuella cellen. Klicka sedan på ikonen till *höger om den gröna triangeln* så körs (exekveras) kommandona i cellen. Program kan också köras från kommandofönstret genom att skriva `runfile` följt av programnamn, inklusive sökväg, inom parentes. Editorn med inskrivna kommandon i två celler visas i figur 2.2.



Figur 2.2: Fönstret till vänster är editorn, där man skriver sina program. Första raden kommer upp automatiskt och anger att Unicode Transformation Format 8 bits (`utf-8`) används för att avkoda symboler. Text inom de trippla citationstecknen är kommentarer. Det är lämplig att här skriva vad programmet gör, när det skapades och vem som skrivit det. Programmet, vilket har givits namnet `funplot.py`, körs genom att klicka på den gröna triangeln i verktygsfältet. Programmet kan också köras genom att i kommandofönstret ge kommandot `runfile` följt av programnamnet inom parentes.

Namn på py-filer får ej börja med siffror. Namn får inte heller innehålla punkter annat än för att markera extension.

För att öppna en redan existerande py-fil klickar man på ikonen för att *öppna* i verktygsfältet. Man kan också gå in via *File* i menyraden. Då man gjort ändringar i en fil måste dessa sparas genom att klicka på ikonen för att *spara*.

Vi tar två exempel för att visa på användningen av py-filer. För att inte skriva så mycket visar vi inte de första raderna med Unicode Transformation Format eller kommentarerna inom citationsstecknen. Text efter `#` är kommentarer och ignoreras av Python. Kommentarer är viktiga för att andra ska kunna förstå hur programmet fungerar.

Exempel 2.4. Den kinetiska energin W för en partikel med massan m och farten v ges av $W = \frac{1}{2}mv^2$. Om vi känner energin W och massan m kan farten beräknas genom

$$v = \sqrt{\frac{2W}{m}}$$

Vi har en partikel med kinetisk energi $W = 50$ och massa $m = 3$. Farten beräknas med följande py-fil

```
# fart.py
import numpy as np      # vi måste importera numpy för att använda roten
W = 50
```

```
m = 3
v = np.sqrt(2*W/m)
print('Farten är ',v) # skriv ut resultatet
```

Py-filen sparas genom att klicka på ikonen för att *spara* och döps till **fart.py**. Filen (programmet) körs genom att klicka på den gröna triangeln och Python skriver ut

Farten är 5.773502691896258

□

Exempel 2.5. Vi har en summa $s = 1 + 2 + 3 + \dots + 100$. Summan beräknas med följande py-fil

```
# summa.py
s = 0                      # sätt summa till noll
for i in range(1,101): # loopa från 1 till 100, måste skriva 101 inte 100
    s = s + i              # addera term till summan

print('Summan är ',s) # skriv ut resultatet
```

Py-filen sparas genom att klicka på ikonen för att *spara* och döps till **summa.py**. Filen (programmet) körs genom att klicka på den gröna triangeln och Python skriver ut

Summan är 5050

Vi återkommer till for-loopar i kapitel 7, där vi också förklarar varför vi skrev 101 inte 100. □

2.6 Automatisk komplettering

Kommandona som skrivs i kommandofönstret lagras i en buffert. Genom att använda piltangenterna ↑ och ↓ kan man bläddra bland de inskrivna kommandona.

Spyder använder sig av automatisk komplettering (eng. TAB-completion), vilket innebär att du bara behöver skriva in början av ett kommando och sedan trycka på TAB-tangenten för att få förslag på hur kommandot kan kompletteras. Detta är mycket användbart! Automatisk komplettering fungerar även på egeninförda variabler, vilket gör att vi inte behöver vara rädda för långa variabelnamn eller för att skriva fel.

Exempel 2.6. Vi avser att skriva kommandot `import numpy`. Då vi börjar skriva

```
im <tryck TAB>
```

och trycker på TAB-tangenten, kompletteras kommandot automatiskt till

```
import
```

Då vi fortsätter kommandot och adderar `num`

```
import num <tryck TAB>
```

och trycker på TAB-tangenten, kommer 5 möjliga kompletteringar fram och vi väljer den önskade kompletteringen `import numpy`. □

Exempel 2.7. Vi har importerat modulen `numpy` genom

```
import numpy as np
```

För att se vilka funktioner som är tillgängliga i modulen skriver vi

```
np. <tryck TAB>      # notera punkten efter np
```

och trycker på TAB-tangenten. Vi får upp en lista över alla funktioner som är tillgängliga i modulen, och vi kan välja den vi önskar. Mycket användbart. □

2.7 Olika typer av tal

I Python arbetar man med heltal (eng. integer) och flyttal (eng. float). Flyttal ges i dubbel precision, vilket innebär att man har 16 värdesiffror. Tal som skrivs utan decimalpunkt blir automatiskt heltal

```
4          # heltal
```

Tal som skrivs med decimalpunkt blir automatiskt flyttal

```
4.0        # flyttal
1.23       # flyttal
```

Observera att man för flyttal måste använda decimalpunkt i stället för decimalkomma.

Stora och små tal skrivs i exponentform. Talet 7.51×10^{-6} skrivs som

```
7.51e-6    # flyttal i exponentform
```

medan talet 8.32×10^{11} skrivs som

```
8.32e11    # flyttal i exponentform
```

För att omvandla från flyttal till heltal använder man kommandona `round` och `int`. För att omvandla från heltal till flyttal använder man `float`.

2.8 Räkning med tal

Python har sju aritmetiska operatorer. Dessa är i prioritetsordning

<code>**</code>	potens (upphöjt till)	prioritet 1
<code>*</code>	multiplikation	prioritet 2
<code>/</code>	division	prioritet 2
<code>//</code>	heltalsdivision	prioritet 2
<code>%</code>	heltalsrest	prioritet 2
<code>+</code>	addition	prioritet 3
<code>-</code>	subtraktion	prioritet 3

Då två operatorer har samma prioritetsordning utförs beräkningarna från vänster till höger.

Exempel 2.8.

(a) För att addera heltalen 12 och 13 skriver vi

```
12 + 13    # heltal + heltal = heltal
```

Resultatet blir ett heltal som skrivs ut på skärmen

25

(b) Vid addition av ett flyttal och ett heltal

```
12.0 + 13  # flyttal + heltal = flyttal
```

blir resultatet alltid ett flyttal

25.0

(c) För att beräkna $2 \cdot 3^2$ skriver vi

```
2*3**2    # först upphöjt till, prio 1, sedan multiplikation, prio 2
```

Eftersom upphöjt till har högst prioritet börjar Python med att beräkna $3^{**}2$. Sedan följer multiplikation med 2 och Python svarar med heltalet

18

(d) Division och multiplikation har samma prioritet. Då vi skriver

```
1/2*3    # operationer från vänster till höger
```

börjar Python från vänster och beräknar $1/2$. Sedan följer multiplikation med 3, vilket ger ett flyttal

1.5

För att undvika missförstånd bör man sätta ut parenteser och i stället skriva talet som $(1/2) \cdot 3$.

(e) För att omvandla flyttalet 4.6789 till ett heltal skriver vi

```
int(4.6789)    # avhuggning
```

Python hugger av (tar bort decimaler och behåller heltalsdelen) och ger

4

(f) För att avrunda flyttalet 4.6789 till ett tal med två decimaler skriver vi

```
round(4.6789,2)  # avrundning till två decimaler
```

och får

4.68

(g) Om vi använder kommandot `round` utan att ange antalet decimaler, får vi avrundning till närmaste heltal. Kommandot

```
round(4.6789)    # avrundning till närmaste heltal
```

ger

5

Notera skillnaden mellan `int`, som hugger av, och `round`, som avrundar till närmaste heltalet. \square

Division av två heltal ger normalt ett flyttal, till exempel $7/2 = 3.5$. I många situationer har man emellertid användning av heltalsdivision

$$\frac{7}{2} = \overbrace{3}^{\text{kvot}} + \overbrace{\frac{1}{2}}^{\text{rest}}.$$

Om resten är noll går divisionen jämnt ut, och nämnaren är en faktor i täljaren. I Python finns kommandona `//` och `%` för att bestämma kvot och rest vid heltalsdivision.

Exempel 2.9.

(a) För att beräkna kvoten vid heltalsdivisionen $11/4 = 2 + 3/4$ skriver vi

```
11//4      # kvot vid heltalsdivision
```

och får heltalet

2

(b) Resten vid heltalsdivisionen beräknas genom

```
11%4      # rest vid heltalsdivision
```

och ger heltalet

3

\square

2.9 Funktioner

Python har ett antal matematiska funktioner som, förutom tal, även accepterar samlingar av tal (vektorer) som argument. Funktionerna finns i modulen `numpy`. Precis som tidigare måste modulen importeras. Detta görs genom kommandot

```
import numpy as np
```

Funktionerna blir nu åtkomliga genom

```
np.funktionsnamn
```

En del av funktionerna i modulen `numpy` listas nedan. Om du inte känner till exponentialfunktionen e^x eller logaritmfunktionen $\ln x$ bli inte orolig. Vi återkommer till dessa i kapitel 11. I exemplen som följer förutsätter vi att `numpy` har importerats.

<code>npfabs(x)</code>	absolutbeloppet $ x $.
<code>np.sqrt(x)</code>	kvadratroten \sqrt{x} .
<code>np.exp(x)</code>	exponentialfunktionen e^x .
<code>np.log(x)</code>	naturliga logaritmen $\ln x$.
<code>np.trunc(x)</code>	x tar bort decimaler, behåller heltalsdelen.
<code>np.round(x,n)</code>	avrundar till angivet antal decimaler.
<code>np.floor(x)</code>	avrundar nedåt (floor är golv).
<code>np.ceil(x)</code>	avrundar uppåt (ceil är tak).
<code>np.pi</code>	$\pi \approx 3.141592653589793$.

Vi tar några exempel för att visa på användningen av funktionerna.

Exempel 2.10. För att beräkna $\sqrt{10}$, dvs. kvadratroten ur 10, ger vi kommandona

```
np.sqrt(10)
```

Python svarar

```
3.1622776601683795
```

Vi kan också lagra 10 i en variabel x och sedan anropa funktionen. Kommandona blir

```
x = 10
np.sqrt(x)
```

och utskriften blir densamma. \square

Exempel 2.11. Följande kommando beräknar 4π och avrundar resultatet till 3 decimaler

```
np.round(4*np.pi,3)      # round(4*np.pi,3) går också bra
```

Python svarar

```
12.566
```

Exempel 2.12. Absolutbeloppet $|x|$ är definierat som avståndet från x till origo. Talet 3 har avståndet 3 till origo, alltså är $|3| = 3$. Talet -3 har avståndet 3 till origo, alltså är $|-3| = 3$. Det följer att avståndet mellan två tal x_1 och x_2 ges av $|x_1 - x_2|$. För att beräkna avståndet mellan π och det Babyloniska närmvärdet $25/8$ (utgrävningar i Susa har visat att Babylonerna kände till detta närmvärdet för π redan 1800 f.Kr.) skriver vi

```
np.fabs(25/8 - np.pi)      # abs(25/8 - np.pi) går också bra
```

Python svarar

```
0.016592653589793116
```

2.10 Lambda-funktioner

Man kan definiera egna så kallade lambda-funktioner (eller anonyma funktioner) som fungerar på samma sätt som Pythons inbyggda funktioner. Lambda-funktioner fås genom kommandot

```
funktionsnamn = lambda variabel: funktionsuttryck
```

Det går även att definiera funktioner som beror av flera variabler. Funktionerna anropas genom att skriva funktionsnamnet följt av argumenten inom parentes. Nedan ger vi exempel på användningen av lambda-funktioner.

Exempel 2.13.

(a) Funktionen $f(x) = \sqrt{x} - 1$ definieras genom att skriva

```
f = lambda x: np.sqrt(x) - 1
```

För att beräkna funktionsvärdet för $x = 4$ matar vi in

```
f(4)
```

Svaret blir

```
1.0
```

(b) Funktionen $g(x) = kx^2$ införs genom

```
g = lambda x, k: k*x**2
```

För att till exempel få $(-1) \cdot 3^2$ skriver vi

```
g(3,-1) # obs ordningen, först x sedan k
```

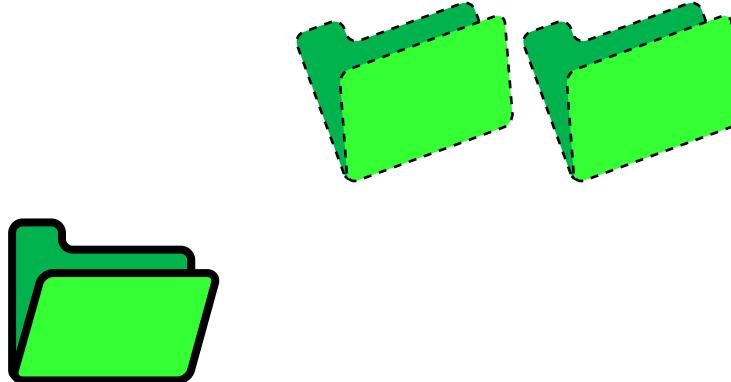
Python returnerar

-9

□

2.11 Organisera ditt arbete

Då du arbetar med Python inom ramen för en kurs är det en mycket god ide att skapa en huvudkatalog med kursens namn. Sedan skapar du underkataloger för olika kapitel eller inlämningsuppgifter. Ge py-filer du skapar i editorn för att köra exempel, testa olika programmeringselement eller lösa inlämningsuppgifter väl valda namn och spara dem i lämpliga underkataloger, se figur 2.3.



Figur 2.3: Det är en god ide att skapa en katalog med kursens namn. Ge sedan py-filer väl valda namn och lagra dem i lämpliga underkataloger.

Kapitel 3

Variabler, datatyper och tilldelningssatser

Variabler är storheter man själv inför och som man kan tilldela värden. Man kan när som helst ta reda på variabelns aktuella värde och utnyttja det i beräkningar. Varje variabel ges ett namn, som man använder då man refererar till variabeln. Variabler är av bestämda datatyper beroende på vilka värden som tilldelas.

3.1 Datatyper

De grundläggande datatyperna för tal är heltal och flyttal. Heltal skrivs utan decimaler medan flyttal skrivs med decimaler. Så är till exempel 4 ett tal av typen heltal medan 4.0 är ett tal av typen flyttal. Matematiskt är båda talen lika, men de lagras och behandlas olika i Python. Förutom datatyperna för tal finns teckensträngar, listor och tupler. En teckensträng är ett antal tecken lagrade efter varandra. Teckensträngar används för utskrift till skärmen, men också för att styra vissa kommandon. Teckensträngar skrivs omslutna av apostrofer. Exempel på teckensträngar är

```
'resultatet är',      'Sara Andersson'.
```

En lista är ett antal objekt givna inom hakparenteser och separerade med komma. Objekten i listan kan vara tal, teckensträngar, andra listor etc. En tupel är detsamma som en lista, men nu används vanliga parenteser i stället för hakparenteser som avgränsare. Det finns skillnader mellan listor och tupler med avseende på hur inmatade objekt kan ändras.

3.2 Tilldelningssatser

Tilldelning av tal

En variabel tilldelas ett talvärde genom att skriva variabelnamnet till vänster och talet till höger

```
variabelnamn = tal
```

Beroende på talet kommer variabeln att vara av typen heltal eller flyttal.

Tilldelning av strängar

En variabel tilldelas värdet av en teckensträng genom att skriva variabelnamnet till vänster och tecknen i teckensträngen omslutna av apostrofer till höger

```
variabelnamn = 'teckensträng'
```

Tilldelning av listor

En variabel tilldelas värdet av en lista med n objekt genom att skriva variabelnamnet till vänster och objekten inom hakparentes separerade med komma

```
variabelnamn = [obj1, obj2, obj3, ..., objn]
```

Tilldelning av tupler

En variabel tilldelas värdet av en tupel med n objekt genom att skriva variabelnamnet till vänster och objekten inom vanlig parentes separerade med komma

```
variabelnamn = (obj1, obj2, obj3, ..., objn)
```

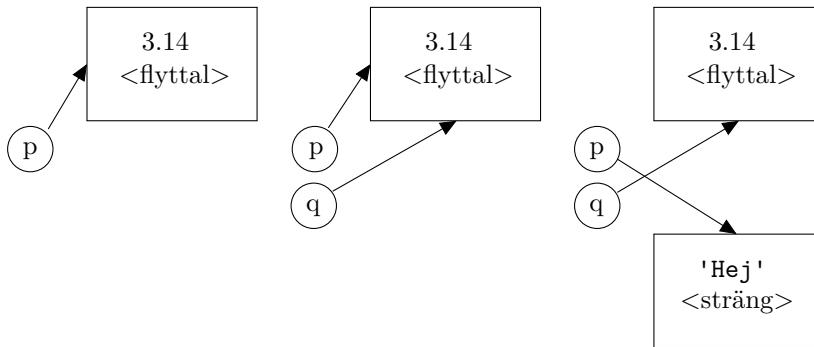
3.3 Tilldelningssatser i tekniska termer

Då en variabel tilldelas ett värde skapas, i tekniska termer, ett objekt i datorns minne, vilket innehåller talet, strängen, listan eller tupeln tillsammans med information om typen samt eventuell annan information. Variabeln blir sedan en referens till det skapade objektet. Då en variabel tilldelas värdet av en tidigare införd variabel skapas inget nytt objekt i datorns minne, utan den nya variabeln refererar till, eller pekar på, objektet som hör till den gamla variabeln.

Exempel 3.1. Vi har följande tilldelningar

```
p = 3.14
q = p
p = 'Hej'
```

Vid första tilldelningen skapas ett objekt av typen flyttal som innehåller värdet 3.14. Variabeln p refererar till detta objekt. Vid andra tilldelningen skapas inget nytt minnesobjekt, i stället refererar q till samma objekt som p . Vid tredje tilldelningen skapas ett nytt objekt av typen sträng som innehåller värdet 'Hej'. Variabeln p refererar till detta objekt medan q fortsätter peka på det ursprungliga minnesobjektet, som innehåller talet 3.14. Tilldelningarna visas i figur 3.1. \square



Figur 3.1: Då en variabel tilldelas värdet av en tidigare variabel skapas inget nytt objekt i minnet, utan den nya variabeln refererar till det gamla objektet.

Exempel 3.2.

(a) Vi har följande tilldelningar

```
m = 6
v = 3.0
W = 0.5*m*v**2
```

Vid den första tilldelningen skapas ett objekt av typen heltal som innehåller värdet 6. Variabeln m refererar till detta objekt. Vid den andra tilldelningen skapas ett objekt av typen flyttal som innehåller värdet 3.0. Variabeln v refererar till detta objekt. Vid den tredje tilldelningen läser Python in aktuella värden på variablerna m och v från datorns minne och beräknar värdet av högerledet, vilket lagras i ett nytt objekt i datorns minne tillsammans med den aktuella typen. Variabeln W refererar till det beräknade objekten.

(b) För att ta reda på värdet av W skriver vi

```
print(W)
```

och Python returnerar

27.0

(c) För att ta reda på datatypen av variabeln W ger vi kommandot

```
type(W)
```

Python svarar

float

□

Exempel 3.3. Vi har $x = 3$ och ska beräkna uttrycket

$$y = \frac{3x + 1}{\sqrt{x^2 - 1} - 1}.$$

Det är så lätt att göra parentesfel. Bäst att mata in täljaren för sig och nämnaren för sig

```
x = 3
t = 3*x + 1           # täljaren för sig och
n = np.sqrt(x**2 - 1) - 1   # nämnaren för sig
y = t/n                # sätt ihop
```

Vid den första tilldelningen skapas ett objekt av typen heltal som innehåller värdet 3. Variabeln x refererar till detta objekt. Vid den andra och tredje tilldelningen läser Python in aktuellt värden på variabeln x från datorns minne och beräknar värdena av högerledet, vilka lagras i t och n . Vid den fjärde tilldelningen läser Python in aktuella värden på variablerna t och n från datorns minne och beräknar värdet av högerledet, vilket lagras i y .

□

Exempel 3.4. Vi inför en lista med fyra tal

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

Då vi skriver

```
a?
```

returnerar Python följande information.

```
Type:      list
String form: [1, 3, 2, -4]
Length:    4
Docstring:
Built-in mutable sequence.
```

Vi ser att variabeln är av typen lista. Vi får värdena på elementen samt längden (antal element) i listan. \square

3.4 Listor, tupler och strängar – indexering

Element i en lista, tupel eller sträng med namnet `u` nås via ett index. Python indexerar från 0 och det första elementet blir `u[0]`, det andra elementet `u[1]` osv. För att få fram det totala antalet element i en lista, tupel eller sträng använder man den inbyggda funktionen `len`. Index för sista elementet blir då `len(u)-1` och vi har uppräkningen

```
u[0], u[1], u[2], ..., u[len(u)-2], u[len(u)-1]
```

Indexering från noll innebär att vi måste vara mycket noga med hur vi uttrycker oss. Första elementet är samma som elementet med index 0, andra elementet är samma som elementet med index 1 osv. Nast sista elementet är elementet med index `len(u)-2`, vilket Python förkortar till elementet med index -2. Sista elementet är elementet med index `len(u)-1`, vilket Python förkortar till elementet med index -1. Negativa index är mycket användbara, se till att du behärskar dem.

Exempel 3.5. Definiera en lista genom

```
u = [-1,3,4,2,-12,8]      # lista med 6 element
```

(a) Antal element i listan fås genom

```
len(u)
```

och vi får svaret

```
6
```

(b) För att skriva ut första, fjärde, sista och nästa sista elementet skriver vi

```
print(u[0])      # första elementet, index 0
print(u[3])      # fjärde elementet, index 3
print(u[-1])     # sista elementet, index -1
print(u[-2])     # näst sista elementet, index -2
```

och Python svarar

```
-1
2
8
-12
```

(c) För att ändra värdet för andra elementet, index 1, till värdet 52 ger vi kommandot

```
u[1] = 52          # ändra andra elementet, index 1
print(u)
```

och får

`[-1, 52, 4, 2, -12, 8]`

□

Exempel 3.6. Definiera en tupel genom

```
x = (1,2,3)      # tupel med tre element
```

(a) Antal element i tupeln fås genom

```
len(x)
```

och vi får svaret

3

(b) För en tupel eller sträng går det inte att ändra värdena för enskilda element, och om vi ger kommandot

```
x[1] = 52
```

får vi felmeddelandet

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

En av skillnaderna mellan listor och tupler är just denna: det går att ändra enskilda element för de förra men inte för de senare. □

Exempel 3.7. Listor är mycket generella, och en lista kan innehålla både tal och teckensträngar, men även andra listor eller tupler, vilka i sin tur är indexerade. Betrakta

```
name    = ['Gudrun', 'Jönsson']
age     = 24
gender = 'female'
person = [name, age, gender]
```

(a) För att få information om innehållet i variabeln `person` skriver vi

```
person?
```

och Python svarar

```
Type:           list
String form:  [['Gudrun', 'Jönsson'], 24, 'female']
Length:        3
Docstring:
Built-in mutable sequence.
```

(b) Vi får fram första elementet genom

```
person[0]
```

Det första elementet är en lista med strängar och Python returnerar

```
['Gudrun', 'Jönsson']
```

För att få det första elementet i denna lista skriver vi

```
person[0][0]
```

och får

```
'Gudrun'
```

Fundera på vilket index den tredje bokstaven i 'Gudrun' har.

□

3.5 Delområden av listor, tupler och strängar

I praktiska problem, som lösas med dator, behöver man ofta arbeta med och referera till delområden (eng. slices) av listor, tupler eller strängar. Antag att vi har dagliga temperaturvärden under en period av ett år lagrade i en lista u med 365 element. För att till exempel beräkna medeltemperaturen under januari måste vi på ett enkelt sätt komma åt delområdet motsvarande denna period, dvs. elementen med index mellan 0 och 30. Det finns flera användbara kommandon för detta.

$u[i]$	ger elementet med index i .
$u[i:j]$	ger delområdet bestående av elementen med index från i till $j - 1$ i steg om 1.
$u[i:]$	ger delområdet bestående av elementen med index från i till $\text{len}(u) - 1$ (sista elementet) i steg om 1.
$u[:j]$	ger delområdet bestående av elementen med index från 0 (första elementet) till $j - 1$ i steg om 1.

Notera mycket noga att sista elementet i delområdet $u[i:j]$ har index $j - 1$ och inte j , som man kanske skulle kunna tro. En fördel med detta är att delområdets längd, antal element, nu enkelt fås som den inskrivna högergränsen j minus den inskrivna vänstergränsen i , dvs. $j - i$.

Om ett delområde finns på högersidan i en tilldelningssats, tilldelas den nya variabeln värdet av delområdet.

Exempel 3.8. Vi har en lista

```
u = [-3, -2, 0, 1, 4, 7]
```

(a) Genom kommandot

```
v = u[2:5] # delområdet har 5 - 2 = 3 element
print(v)
```

tilldelas v värdet av delområdet bestående av element med index 2 till och med index 4 varvid Python svarar

```
[0, 1, 4]
```

(b) Genom kommandot

```
v = u[1:] # samma som u[1:len(u)]
print(v)
```

tilldelas v värdet av delområdet bestående av element med index 1 till och med index $\text{len}(u) - 1$ (sista elementet). Python svarar

```
[-2, 0, 1, 4, 7]
```

(c) Genom kommandot

```
v = u[0:5:2]
print(v)
```

tilldelas `v` värdet av delområdet bestående av element med index 0 till och med index 4 i steg om 2, dvs. index 0, 2, 4 och vi får

`[-3, 0, 4]`

□

Exempel 3.9. Delområden fungerar också för strängar. Tilldela `s` värdet av en sträng

```
s = 'Hejsan hoppsan'
```

Genom kommandot

```
t = s[7:]
print(t)
```

tilldelas `t` värdet av den sista delen av teckensträngen och vi får

`hoppsan`

□

Pythons specifikation av delområden känns lite knepig: att högergränsen skall anges som ett mer än det man vill ha. Det är dock bara att acceptera.

Tänk så här så kommer du alltid rätt.

Jag vet vänstergränsen, dvs. index för första elementet i delområdet, och jag vet också hur många element i delområdet jag vill ha, då blir högergränsen

```
högergränsen = vänstergränsen + antal element
```

Om du strikt håller dig till detta kommer du alltid rätt.

3.6 Klasser – instans, attribut, metod

Datatyperna tal, teckensträngar, listor och tupler är implementerade som klasser. En klass är en beskrivning av objekt med samma struktur och egenskaper. Ett variabel som tillhör en klass kallas en instans av klassen. Objekt som beskriver instansens egenskaper kallas attribut (synonymer: utmärkande egenskap, igenkänningstecken, kännetecken, särmärke) till instansen. Attribut till en instans fås genom den så kallade punktnotationen

```
instans.attribut      # punktnotation för attribut
```

En funktion som utför operationer på en instans kallas en metod till instansen. En metod till en instans anropas genom

```
instans.metod(argument)  # punktnotation för metod
```

där `argument` är eventuella ytterligare data som metoden behöver. Om metoden utför operationer på instansen utan ytterligare argument blir anropet

```
instans.metod()          # parentesen måste med, även om den är tom
```

Låt variabeln `a` vara en instans till en klass. Vi kan använda TAB-tangenten för att ta reda på vilka metoder som är knutna till instansen

```
a. <tryck TAB> # glöm ej punkten
```

I listan med metoder som kommer upp kan vi välja den metod vi är intresserade av. För att få tillgång till dokumentationen till I listan med metoder som kommer upp kan vi välja den metod vi är intresserade av. För att få tillgång till dokumentationen tillhörande den valda metoden ger vi kommandot

```
a.metod?
```

Dokumentationen talar ofta om huruvida metoden verkar på instansen på plats (eng. in-place) eller om metoden returnerar ett värde, vilket kan lagras i en ny variabel. I sektionerna nedan kommer vi att diskutera de viktigaste metoderna för listor och strängar och ge exempel på hur metoderna används.

3.7 Metoder för listor

En instans *a* av klassen listor har följande metoder.

<i>a.append(x)</i>	addrar <i>x</i> i slutet av listan.
<i>a.copy()</i>	returnerar en kopia av listan.
<i>a.extend(x)</i>	bygger på listan med elementen i <i>x</i> .
<i>a.insert(i,x)</i>	skjuter in <i>x</i> i positionen <i>i</i> .
<i>a.remove(x)</i>	tar bort första elementet i listan vars värde är <i>x</i> .
<i>a.count(x)</i>	antal gånger <i>x</i> förekommer i listan.
<i>a.sort()</i>	sorterar elementen i listan på plats.
<i>a.reverse()</i>	vänder på ordningen av elementen i listan på plats.
<i>a.pop()</i>	tar bort sista elementet i listan.

Utöver metoderna ovan finns även användbara funktioner som opererar på listor och som man anropar på vanligt sätt. Funktionen (kommandot) *dèl* tar bort element.

Exempel 3.10. Vi definierar en lista

```
u = [1,2,5,'hej',9]      # u instans till klassen listor
```

(a) För att addera 42 i slutet av listan skriver vi

```
u.append(42)            # append, metod som verkar på instansen u
print(u)
```

och får

```
[1,2,5,'hej',9,42]
```

(b) För att skjuta in strängen 'Nisse' på position 1

```
u.insert(1,'Nisse')     # insert, metod som verkar på instansen u
print(u)
```

Listan ser nu ut som

```
[1,'Nisse',2,5,'hej',9,42]
```

(c) För att ta bort första förekomsten av 2 från listan ger vi kommandot

```
u.remove(2)          # remove, metod som verkar på instansen u
print(u)
```

och listan blir

```
[1, 'Nisse', 5, 'hej', 9, 42]
```

(d) För att ta bort elementet med index 0 skriver vi

```
del u[0]          # del ingen metod, vanligt kommando
print(u)
```

Listan är nu

```
['Nisse', 5, 'hej', 9, 42]
```

(e) För att ta bort elementen med index 2 till och med sista elementet skriver vi

```
del u[2:]
print(u)
```

vilket ger

```
['Nisse', 5]
```

□

Exempel 3.11. Vi startar med en lista

```
x = [1, 2, 5, -4, 2, -8]    # x instans till klassen lista
```

(a) Följande kommandon sorterar listan på plats

```
x.sort()          # sort, metod som verkar på instansen x,
print(x)          # x sorteras på plats
```

och vi får

```
[-8, -4, 1, 2, 2, 5]
```

Att listan sorteras på plats innebär att det *inte* går att göra en tilldelning

```
y = x.sort()      # fungerar inte, x sorteras på plats
```

(b) För att kasta om ordningen på elementen i listan, även detta på plats, ger vi kommandot

```
x.reverse()          # reverse, metod som verkar på instansen x
print(x)              # vänder på ordningen av x på plats
```

Utskriften blir

```
[5, 2, 2, 1, -4, -8]
```

□

3.8 Metoder för strängar

Där finns 45 metoder för strängar, och vi redovisar här bara de mest användbara. Låt *s* vara en instans av klassen strängar. Vi har då följande metoder.

<code>s.split()</code>	delar upp en sträng i bitar, vilka lagras i en lista.
<code>s.join(list)</code>	sätter ihop strängar i en lista till en större sträng. <i>str</i> bestämmer hur strängarna separeras.
<code>s.find(delstr)</code>	letar efter en delsträng i en sträng.
<code>s.replace(old,new)</code>	ersätter en gammal sträng med en ny sträng.
<code>s.strip()</code>	tar bort inledande och avslutande blanktecken.
<code>s.lstrip()</code>	tar bort inledande blanktecken.
<code>s.rstrip()</code>	tar bort avslutande blanktecken.

Exempel 3.12.

- (a) Vi har en sträng *s*, där delsträngarna är separerade av blanktecken

```
s = 'Massan av partikeln' # s instans till klassen strängar
```

Kommandona

```
ord = s.split()      # split, metod som verkar på instansen s
print(ord)
```

ger en uppdelning av delsträngar som lagras i en lista

```
['Massan', 'av', 'partikeln']
```

- (b) Vi har en sträng *s*, där delsträngarna är separerade av komma

```
s = 'massa,hastighet'
```

Kommandona

```
ord = s.split(',')      # split, metod som verkar på instansen s
print(ord)
```

ger uppdelningen

```
['massa', 'hastighet']
```

□

Exempel 3.13.

- (a) Vi har två listor

```
l1 = ['Lösningen','till','ekvationen']
l2 = ['massa','impuls','rörelsemängd']
```

Följande kommandon sätter ihop strängarna i lista 1 med blanktecken och strängarna i lista 2 med kommatecken och ett blanktecken

```
s1 = ' '.join(l1) # sätt ihop med ett blanktecken
s2 = ', '.join(l2) # sätt ihop med komma och ett blanktecken
```

De resulterande strängarna blir

```
'Lösningen till ekvationen'
'massa, impuls, rörelsemängd'
```

(b) Vi har en variabel m , vilken tilldelas flyttalet 5.8

```
m = 5.8
```

Följande kommandon omvandlar flyttalet som är lagrat i m till en sträng, vilken läggs i en lista tillsammans med andra strängar

```
l3 = ['massan är', str(m), 'kilogram']
```

Strängarna sätts ihop med ett blanktecken genom

```
s3 = ' '.join(l3) # sätt ihop med ett blanktecken
```

och resultatet blir

```
'massan är 5.8 kilogram'
```

Ovanstående metod att foga samman tal och strängar till en ny sträng är mycket användbar och vi kommer att använda den längre fram. \square

Exempel 3.14. Vi har en sträng

```
s = ' farten är ' # s instans till klassen strängar
```

(a) För att ta bort inledande och avslutande blanktecken skriver vi

```
s1 = s.strip() # strip metod som verkar på instansen s
print(s1)
```

och får

```
'farten är'
```

(b) För att rensa en sträng från inledande blanktecken ger vi kommandot

```
s2 = s.lstrip()
print(s2)
```

Python returnerar

```
'farten är '
```

(c) För att ersätta 'farten' med 'hastigheten' skriver vi

```
s3 = s2.replace('farten', 'hastigheten')
```

och får

```
'hastigheten är '
```

\square

Kapitel 4

Variabelnamn, inläsning och utskrift

Detta är ett kort kapitel som diskuterar reglerna för variabelnamn och understryker vikten av att välja dessa omsorgsfullt. Vidare diskuterar vi metoder att både skriva ut variabler till skärmen, men också att läsa in variabler från fil.

4.1 Variabelnamn, variabelutforskaren

Variabelnamn får innehålla bokstäver från det engelska alfabetet (inga å, ä, ö) samt understrykning. Variabelnamn får innehålla siffror, men inte börja med en siffra. I Python skiljer man på stora och små bokstäver och `x` och `X` är olika variabler. Man får inte heller använda nyckelord som `import`, `if`, `and`, `true` etc. som variabelnamn.

Då vi börjar skriva program ökas läsbarheten av väl valda variabelnamn. Om du skriver ett program för att beräkna ett värde av ett matematiskt uttryck, använd samma namn som i uttrycket. Om variablerna kallas x och β i uttrycket, kalla dem `x` och `beta` i programmet. Använd korta variabelnamn som `t` för tid och `E` eller `W` för energi. Ibland kan det dock vara motiverat att skriva ut vad en variabel representerar, t.ex. `massa` eller `fart`. Det är inget problem att använda längre variabelnamn, då vi kan använda oss av automatisk komplettering via TAB-tangenten.

Det är en god vana att använda i, j, k, l, m, n för heltal och index och övriga bokstäver för flyttal. Fastän det inte är förbjudet, bör man inte använda samma variabelnamn för olika ändamål i ett program.

Exempel 4.1. Följande är exempel på tillåtna variabelnamn

```
massa, v, t_start, t_slut, x0, x1
```

Exempel på otillåtna variabelnamn

```
1ex, år, false, if
```

För att hålla reda på vilka variabler som är definierade samt deras typ värde använder man oftast Spyders variabelutforskare (eng. variable explorer). Variabelutforskaren aktiveras genom att klicka på knappen `Variable explorer` under fönstret uppe till höger, se figur 2.1. Ett exempel på hur informationen i variabelutforskaren ser ges i figur 4.1. Genom att använda ikonen med sotpungan (radergummi i tidigare versioner av Spyder) kan vi ta bort alla definierade variabler.

Name	Type	Size	Value
a	list	4	[1, 3, 2, -4]
m	int	1	6
v	float	1	3.0
w	float	1	27.0

Help Variable explorer Plots Files Profiler

Figur 4.1: I variabelutforskaren kan man se vilka variabler som är definierade, vilken typ de har och vilka värden de antar. Genom att använda ikonen med soptunnan (radergummit) kan vi ta bort alla definierade variabler. Ikonen längst till vänster används för att läsa data från fil. Ikonen till vänster om radergummit används för att skriva variabler till fil.

4.2 Inläsning och utskrift till skärmen

Python har olika in- och utmatningsmöjligheter för variabler. Inläsning från tangentbordet kan göras med kommandot `input`. Utskrift till skärmen sker med hjälp av kommandot `print`.

<pre>x = input(teckensträng)</pre> <pre>print(sträng och tal)</pre>	<p>skriver ut teckensträngen på skärmen och väntar på att användaren ska mata in en variabel i form av en teckensträng och rycka på retur. Om numeriska värden önskas måste strängen omvandlas till heltal eller flyttal med <code>int</code> eller <code>float</code>. Talet lagras i variabeln <code>x</code>.</p> <p>skriver en kombination av strängar och tal.</p>
--	---

Exempel 4.2.

(a) Följande kommandon, samlade i ett program med namnet `inut1.py`, läser in massan och farten hos en partikel och räknar ut den kinetiska energin. Energin skrivs ut på skärmen.

```
# inut1.py
m = float(input('Ge m ')) # omvandla input till flyttal
v = float(input('Ge v ')) # omvandla input till flyttal
W = 0.5*m*v**2
print('Kinetisk energi',W)
```

När vi kör programmet och matar in 2.0 för massan och 5.0 för farten ger Python följande utskrift

Kinetisk energi 25.0

□

Exempel 4.3.

Arean av en trianglar ges fås som basen gånger höjden genom två. Då vi skriver

```
b = 3
h = 5
A = b*h/2
print('Om basen är',b, 'och höjden är',h,'blir arean',A)
```

kommer teckensträngarna och variablene b , h och A att skrivas ut och vi får

Om basen är 3 och höjden är 5 blir arean 7.5 □

4.3 Läsa och skriva variabler till fil

I många situationer behöver vi läsa och skriva variabler till fil. Det kan göras på många olika sätt och i olika format och vi återkommer till detta i kapitel 5.11. Det enklaste sättet är att använda verktygen i variabelutforskaren. Då vi klickar på ikonen för *Save data as ...* (tredje ikonen från vänster i figur 4.1) sparar alla definierade variabler i en fil i Spyders eget format 'Spyder data files' med extensionen .spydata. Användaren bestämmer själv namnet på filen. Vi kan nu rensa alla variabler, igen genom att använda verktygen i variabelutforskaren. Då vi klickar på ikonen för *Import data* (första ikonen från vänster i figur 4.1) och anger namnet på filen där vi sparade variablene, läser Spyder in variablene igen så att vi kan fortsätta arbeta med dessa. Filer som innehåller variabler kan t.ex. skickas per epost och delas med andra eller göras tillgängliga via hemsidor eller andra plattformar. Under kursens gång kommer du att behöva ladda ner filer med variabler från Canvas.

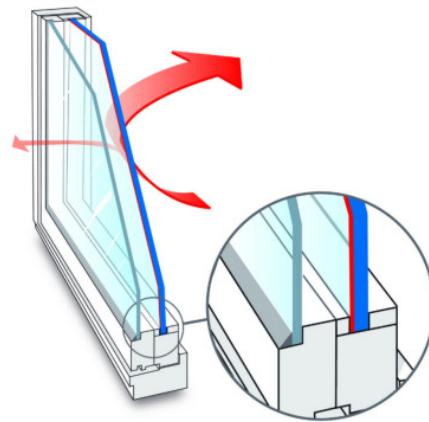
Exempel 4.4. Vi börjar med att rensa alla variabler genom att använda radergummit i variabelutforskaren. Vi har tagit upp ett antal samhörande värden på x och y från ett experiment. Vi lagrar värdena i två listor x och y

```
x = [1.1, 1.9, 3.0, 4.7]
y = [1.1, 4.1, 9.5, 21.3]
```

Vi vill dela värdena med en kompis och klickar på klickar på ikonen för *Save data as ...* (tredje ikonen från vänster i figur 4.1) och väljer filnamnet **experiment**. Vi får nu en fil med namnet **experiment.spdata** i den aktuella arbetskatalogen. Vi skickar ett mail till kompisens och inkluderar filen som en bilaga. Komisen öppnar mailet och laddar ner filen till en katalog på sin dator, klickar på ikonen för *Import data* (första ikonen från vänster i figur 4.1), bläddrar till rätt katalog och väljer filen, varvid variablene x och y läses in. □.

4.4 Tillämpning: fönsterbyte

Fönsters U-värde är ett mått på hur isolerande de är. Ju lägre U-värde desto bättre isoleringsförmåga. Moderna fönster har U-värden $1.0 - 1.3 \text{ W/m}^2\text{K}$. Gamla tvåglasfönster har U-värden på $2.8 - 3.0 \text{ W/m}^2\text{K}$, medan gamla treglasfönster har U-värden $1.8 - 2.0 \text{ W/m}^2\text{K}$, se figur 4.2.



Figur 4.2: Ett fönster med lågt U-värde släpper ut lite värme till den kalla uteluften.

Vid byte av gamla fönster mot nya bestäms energibesparingen av skillnaden i U-värde mellan gamla och nya fönster, den totala fönsterytan och en faktor som kallas gradtimmar. Gradtimmar, gett i enheten kKh, är den över året ackumulerade skillnaden mellan utetemperatur och normal innehålls temperatur och är olika för olika orter i Sverige. Gradtimmarna för några orter i Sverige ges nedan.

ort	gradtimmar	ort	gradtimmar
Malmö	79	Falun	113
Göteborg	80	Umeå	125
Stockholm	93	Östersund	127
Växjö	94	Luleå	139
Gävle	109	Kiruna	143

Som exempel på beräkning av energibesparing tittar vi på fönsterbyte i Falun. För en villa med fönsterytan 20 m^2 ger ett byte från gamla fönster med $U = 2.8 \text{ W/m}^2\text{K}$ till nya fönster med $U = 1.2 \text{ W/m}^2\text{K}$ en årlig energibesparing på

$$\underbrace{(2.8 - 1.2)}_{\text{skillnad } U} \cdot \underbrace{20}_{\text{area}} \cdot \underbrace{113}_{\text{gradtimmar}} \text{ kWh} = 3616 \text{ kWh.}$$

Givet uppvärmningssättet för huset, direktverkande el, fjärrvärme, värmepump etc. och aktuella energipriser, kan vi omvandla energibesparingen till en kostnadsbesparing.

Programmet **energispar.py** tillåter användaren att mata in gamla och nya U-värden, fönsterarea, gradtimmar, samt aktuellt elpris i kr/kWh och beräknar kostnadsbesparingen per år (förutsatt direktverkande el).

```
# energispar.py
Ugamla      = float(input('U-värde för gamla fönster '))
Unya        = float(input('U-värde för nya fönster '))
area        = float(input('Fönsterarea '))
gradtimmar = float(input('Gradtimmar för orten '))
elpris      = float(input('Elpris '))

energibesparing = (Ugamla - Unya)*area*gradtimmar
kostnadsbesparing = energibesparing*elpris
```

```
print('Energibesparing ', round(energibesparing), 'kWh per år')
print('Kostnadsbesparing', round(kostnadsbesparing), 'kr per år')
```

Då vi kör programmet och matar in värdena 2.9 och 1.2 för U-värdena, 20 för arean, 79 för gradtimmar (Malmö) och 1.5 för elpris får vi utskriften

```
Energibesparing 2686 kWh per år
Kostnadsbesparing 4029 kr per år
```

Beräkningar från energikonsulter ger vid handen att återbetalningstiden för ett fönsterbyte är omkring 15 till 18 år.

Kapitel 5

Vektorer och matriser

Ett fält är en samlings tal som kan ordnas med hjälp av ett eller flera index. Fält är den viktigaste datatypen vid beräkningar och simuleringar och kommer att användas i alla följande kapitel. Vi ska här se hur fält införs i paketet `numpy`. Vidare ska vi gå igenom hur man räknar med fält och se vilka funktioner och metoder som finns definierade.

5.1 NumPy

Det viktigaste paketet i Python är `numpy`. Grunden i `numpy` är fält (eng. arrays) av tal implementerade i klassen `ndarray`. Till klassen hör hundratals funktioner och metoder som opererar på instanser av klassen. Det har blivit standard att `numpy` importeras genom kommandot

```
import numpy as np
```

Funktioner från `numpy` blir nu tillgängliga via

```
np.funktionsnamn
```

5.2 Fält

Fält i klassen `ndarray` karakteriseras av antalet index eller dimensioner som används för att ordna elementen. I stället för dimensioner talar man om axlar, där axel 0 svarar mot första dimensionen och axel 1 mot andra dimensionen. Formen på ett fältet fås genom att ange antalet element för varje dimension (antal element längs varje axel) av fältet. Storleken på ett fält är det totala antalet element och fås genom att räkna ihop antalet element längs varje axel. Slutligen är fält av olika typ beroende på typen av de lagrade talen.

Fält av dimensionen ett – vektorer

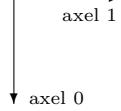
Ett fält x av dimensionen ett kallas en vektor och är en samlings heltal eller flyttal som kan ordnas med hjälp av ett index

$$x = (x_0, x_1, \dots, x_n).$$

Längden av vektorn är lika med antalet element.

Fält av dimensionen två – matriser

Ett fält a av dimensionen två kallas en matris och är ett rektangulärt schema av heltal eller flyttal

$$a = \begin{pmatrix} a_{00} & a_{01} & \dots & a_{0n} \\ a_{10} & a_{11} & \dots & a_{1n} \\ \dots & \dots & \dots & \dots \\ a_{m0} & a_{m1} & \dots & a_{mn} \end{pmatrix}$$


Matrisen ovan har $m + 1$ rader och $n + 1$ kolonner och sägs vara en $m + 1 \times n + 1$ -matris. Talen a_{jk} kallas matriselement. Första indexet j anger i vilken rad (position längs axel 0) och andra indexet k i vilken kolonn (position längs axel 1) talet a_{jk} står.

5.3 Fält – attribut

Låt a vara en instans av klassen `ndarray`. Vi har följande attribut relaterade till dimension, form, antal element och datatyp.

<code>a.ndim</code>	antal dimensioner (index).
<code>a.shape</code>	tupel som ger antalet element för varje dimension (antalet element längs axlarna).
<code>a.size</code>	totala antalet element.
<code>a.dtype</code>	datatypen: heltal, flyttal.

Vi kommer att exemplifiera hur attributen används i flera av de följande kapitlen.

5.4 Vektorer

I Python får vektorer genom att använda den inbyggda funktionen `array` och skriva in elementen inom hakparentes

```
np.array([x0,x1,...,xn])
```

där datatypen (eller bara typen) för vektorn beror av typen på de inmatade talen. Vi kan också explicit ange typen och skriva

```
np.array([x0,x1,...,xn],dtype='typ')
```

där `typ` är `int` för heltal och `float` för flyttal.

Exempel 5.1.

(a) Vi genererar vektorn $x = (5, -4, 6)$ och visar resultatet

```
x = np.array([5,-4,-6])      # x instans till klassen ndarray
print(x)
```

Python svarar

```
[ 5 -4 -6]
```

Eftersom alla elementen är heltal blir vektorn av typen heltal.

(b) Vi kan få typen som

```
x.dtype      # dtype, attribut till instansen x
```

och svaret blir

```
dtype('int32')
```

(c) Antalet element fås som

```
x.size      # size, attribut till instansen x
```

och Python returnerar

```
3
```

(d) Vi kan ändra elementet med index 1 till något annat värde, t.ex. 92, genom att skriva

```
x[1] = 92
print(x)
```

Python returnerar

```
[ 5 92 -6]
```

(e) Om vi försöker ändra ett elementet till ett flyttal, trunkeras (trunkera = hugga av) flyttalet till ett heltal innan tilldelningen. Till exempel ger

```
x[0] = 10.6      # varning! x är av typen heltal, talet 10.6
print(x)         # trunkeras, huggs av, innan tilldelningen
```

resultatet

```
[10 92 -6]
```

(f) Vi omvandlar vektorn x av typen heltal till typen float genom

```
x = x.astype(dtype='float')
print(x)
```

Exempel 5.2.

(a) Då vi ger kommandona

```
x = np.array([5.0,-4.0,-6.0])
print(x)
```

alternativt

```
x = np.array([5,-4,-6],dtype='float')
print(x)
```

svarar Python

```
[ 5. -4. -6.]
```

Eftersom ett av elementen i vektorn är ett flyttal blir vektorn av typen flyttal. Då vi frågar efter typen

```
x.dtype
```

svarar Python

```
dtype('float64')
```

(b) Tilldelningen

```
x[0] = 10.6          # x är av typen float, talet 10.6 behåller sitt
print(x)             # värde vid tilldelningen
```

ger

```
[10.6 -4. -6.]
```

□

5.5 Arange och linspace

Vektorer, där elementens värden ligger på konstanta avstånd, är mycket vanliga inom programmering. Python har flera funktioner för att generera sådana vektorer.

<code>np.arange(a,b)</code>	ger en vektor med värden från a till b i steg om ett. Om $a > b$ fås en tom vektor. Sista talet i vektorn är alltid mindre än b .
<code>np.arange(a,b,h)</code>	ger en vektor med värden från a till b i steg om h , där h kallas steglängden. Steglängden kan vara negativ.
<code>np.linspace(a,b)</code>	ger en vektor med 50 element jämnt fördelade mellan a och b .
<code>np.linspace(a,b,n)</code>	ger en vektor med n element jämnt fördelade mellan a och b .

Exempel 5.3.

(a) Då vi skriver

```
x = np.arange(20,24)      # resultatet blir heltal, sista elementet
print(x)                  # alltid mindre än högergränsen
```

blir resultatet av typen heltal. Python svarar med

```
[20 21 22 23]
```

Steglängden är ett och behöver inte sättas ut. Observera att det sist genererade elementet är mindre än högergränsen 24.

(b) Motsvarande tilldelning men med talen givna som flyttal

```
x = np.arange(20.0,24.0)  # resultatet blir flyttal, sista elementet
print(x)                  # alltid mindre än högergränsen
```

ger en vektor av typen flyttal. Python svarar

```
[20. 21. 22. 23.]
```

För att få flyttal kan vi också ange gränserna som heltal, men explicit ange typen flyttal på följande sätt

```
x = np.arange(20,24,dtype='float')
```

(c) Då vi skriver

```
x = np.arange(20.0,22.2,0.3)
print(x)
```

svarar Python med

```
[20. 20.3 20.6 20.9 21.2 21.5 21.8 22.1]
```

Steglängden är 0.3.

(d) Även negativa steglängder är tillåtna. Till exempel ger

```
x = np.arange(10.0,0.0,-2.0)
print(x)
```

resultatet

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

(d) Kommandot

```
x = np.linspace(0,10)      # för linspace är sista talet lika med
                           # högergränsen
print(x)
```

ger en vektor x med 50 element jämnt fördelade mellan 0 och 10

```
[ 0. 0.20408163 0.40816327 .... 9.79591837 10.]
```

Om vi vill ha vektor med 100 element jämnt fördelade mellan 0 och 100 skriver vi

```
x = np.linspace(0,10,100)
print(x)
```

5.6 Delvektorer

I praktiska problem som lösas med dator behöver man ofta arbeta med och referera till delvektorer. Detta görs på samma sätt som när vi refererar till delområden av listor, tupler eller strängar (jfr kapitel 4.2). Låt x vara en vektor.

$x[i]$	ger elementet med index i .
$x[i:j]$	ger delvektorn bestående av elementen med index från i till $j - 1$ i steg om 1.
$x[i:]$	ger delvektorn bestående av elementen med index från i till sista elementet i steg om 1.
$x[:j]$	ger delvektorn bestående av elementen med index från 0 (första elementet) till $j - 1$ i steg om 1.
$x[[i1,\dots,in]]$	ger delvektorn bestående av elementen med index i_1, i_2, \dots, i_n .

Exempel 5.4. Vektorn x är definierad som

```
x = np.array([-3,-2,0,1,4,7])
```

(a) Genom kommandot

```
y = x[2:5]          # delområdet har 5 - 2 = 3 element
print(y)
```

tilldelas y värdet av delvektorn av x bestående av elementen med index 2, 3 och 4 varvid Python svarar

[0 1 4]

(b) Genom kommandot

```
y = x[2:]          # från index 2 till sista
print(y)
```

tilldelas y värdet av delvektorn av x bestående av elementen med index från 2 till och med sista elementet varvid Python svarar

[0 1 4 7]

□

Vi kan använda notationen för delvektorer för att tilldela nya värden för elementen i en vektor.

Exempel 5.5. Vektorn v är definierad som

```
v = np.array([5,2,3,4,9,7])
```

(a) För att tilldela elementen med index 0, 2, 4 (tre stycken) de nya värdena 0, 6, 8 skriver vi

```
v[0:5:2] = [0,6,8]
print(v)
```

Python ger den nya vektorn

[0 2 6 4 8 7]

(b) För att tilldela elementen med index från 0 till och med 2 de nya värdena 7, 8, 9 ger vi kommandot

```
v[:3] = [7,8,9]
print(v)
```

Svarsutskriften blir

[7 8 9 4 8 7]

□

5.7 Matriser

I Python fås matriser genom att skriva elementen radvis inom hakparentes

```
A = np.array([[a00,...,a0n],[a10,...,a1n],...])
```

där typen på matrisen beror av typen på de inmatade talen. Vi kan också explicit ange typen och skriva

```
A = np.array([[a00,...,a0n],[a10,...,a1n],...],dtype='typ')
```

Exempel 5.6. Vi har en matris

$$A = \begin{pmatrix} 1 & -2 & 3 \\ 0 & 5 & 4 \end{pmatrix}.$$

(a) Vi matar in matrisen genom

```
A = np.array([[1,-2,3],      # vi kan mata in matrisen rad för rad
              [0,5,4]])    # vilket är lättare att läsa
print(A)                      # A instans till klassen ndarray
```

Utskriften blir

```
[[ 1 -2  3]
 [ 0  5  4]]
```

(b) Typen fås genom

```
A.dtype          # dtype, attribut till instansen A
```

Eftersom de inmatade element är av typen heltal blir matrisen av typen heltal

```
dtype('int32')
```

(c) Dimensionen (antal index) av A fås genom

```
A.ndim          # ndim, antal index, är attribut till instansen A
```

Python svarar

```
2
```

(d) För att få formen (antalet rader och kolonner) ger vi kommandot

```
A.shape         # shape, antal rader och kolonner, attribut till instansen A
```

Vi får svaret i form av en tupel

```
(2,3)
```

Matrisen har två rader och tre kolonner. För att lagra formen i två variabler, m och n , där m är antalet rader och n antalet kolonner ger vi i stället kommandot

```
(m,n) = A.shape    # m antalet rader, n antalet kolonner
```

5.8 Noll- och ettmatraser

Man behöver ibland generera vektorer och matriser med enbart nollor eller enbart ettor. Detta kan göras med kommandona `zeros` och `ones`. Precis som med andra kommandon som genererar vektorer och matriser kan man explicit ange typen.

<code>np.zeros((m,n))</code>	ger en $m \times n$ -matris med nollor, obs ge m och n som en tupel.
<code>np.ones((m,n))</code>	ger en $m \times n$ -matris med ettor, obs ge m och n som en tupel.
<code>np.zeros_like(B)</code>	ger en matris med ettor med samma dimension som matrisen B .
<code>np.ones_like(B)</code>	ger en matris med nollor med samma dimension som matrisen B .

Exempel 5.7.

- (a) En 2×3 -matris A med bara nollor skapas genom

```
A = np.zeros((2,3)) # obs, mata in som tupel
```

Om vi inte anger typen blir den automatiskt flyttal och Python svarar

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

- (b) För att få en 3×3 -matris med bara ettor av typen heltal skriver vi

```
A = np.ones((3,3)) # obs, mata in som tupel
```

Python returnerar

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

□

5.9 Slumptäta matriser

Slumptal är användbara i en mängd olika sammanhang då man vill simulera stokastiska (slumptäta) skeenden eller processer. I Python finns kommandon för att generera slumptal som följer en mängd olika fördelningar. Här ska vi titta på kommandon för att generera slumptal som är likformigt fördelade i intervallet $[0, 1]$ med täthetsfunktionen

$$f(x) = 1, \quad 0 < x < 1,$$

och slumptal som följer en normalfördelning (Gaussfördelning) med täthetsfunktionen

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}, \quad -\infty < x < \infty.$$

Täthetsfunktionen är symmetrisk kring medelvärdet noll och har standardavvikelsen $\sigma = 1$. Normalfördelning och motsvarande täthetsfunktion utforskas närmare i avsnitten 9.5 och 11.9.

<code>np.random.rand(m,n)</code>	ger en $m \times n$ -matris där elementen är slumptal som är likformigt fördelade mellan 0 och 1.
<code>np.random.randn(m,n)</code>	ger en $m \times n$ -matris där elementen är normalfördelade slumptal.
<code>np.random.randint(i,j, [m,n])</code>	ger en $m \times n$ -matris där elementen är heltal, slumpvis dragna från heltalen mellan i och $j - 1$.

Exempel 5.8.

- (a) En vektor v med 5 slumptal, likformigt fördelade mellan 0 och 1 fåras genom

```
v = np.random.rand(5)          # likformigt fördelade mellan 0 och 1
print(v)
```

Python ger då (olika element för varje gång vi kör kommandot)

```
[0.28553887 0.7995323 0.94427488 0.37395085 0.47575929]
```

- (b) En 4×3 -matris A med normalfördelade slumptal genereras genom

```
A = np.random.randn(4,3)          # normalfördelade, medelvärde 0 och
print(A)                         # standardavvikelse 1
```

vilket ger resultatet

```
[[ 1.85250857 -0.38234192  0.42401549]
 [-0.55625384  0.6777707   0.71666356]
 [ 1.71032702 -0.01210735  0.13541593]
 [ 0.4788194   1.26397417  0.54245342]]
```

(c) För att generera en vektor v med 15 heltal, slumpvis dragna mellan 1 och 6 skriver vi

```
v = np.random.randint(1,7,15)    # obs 1,7 och inte 1,6
print(v)
```

och får

```
[3 3 5 4 1 2 3 1 3 5 5 6 3 2 2]
```

□

5.10 Delmatriser

Precis som med vektorer kan man arbeta med och referera till delmatriser.

$A[j,k]$	ger elementet med index (j,k) i matrisen A .
$A[j1:j2,k1:k2]$	ger delmatrisen till A bestående av elementen i raderna med index $j1$ t.o.m. $j2 - 1$ och kolonnerna med index $k1$ t.o.m. $k2 - 1$.
$A[:,k1:k2]$	ger delmatrisen till A bestående av elementen i raderna med index j t.o.m. sista raden och kolonnerna med index $k1$ t.o.m. $k2 - 1$.
$A[:j,k1:k2]$	ger delmatrisen till A bestående av elementen i raderna med index från 0 till $j - 1$ och kolonnerna med index $k1$ t.o.m. $k2 - 1$.
$A[j1:j2,:]$	ger delmatrisen till A bestående av elementen i raderna med index $j1$ t.o.m. $j2 - 1$ och kolonnerna med index k t.o.m sista kolonnen.
$A[j1:j2,:k]$	ger delmatrisen till A bestående av elementen i raderna med index $j1$ t.o.m. och med $j2 - 1$ och kolonnerna med index från 0 t.o.m. $k - 1$
$A[j1:j2,k]$	ger delmatrisen till A bestående av elementen i raderna med index $j1$ t.o.m. $j2 - 1$ i kolonnen med index k .
$A[:,k]$	ger delmatrisen till A bestående av elementen i alla raderna i kolonnen med index k . Kommandot ger alltså kolonnen med index k .
$A[:,[k1,\dots,kp]]$	ger delmatrisen till A bestående av kolonnerna med index $k1,\dots,kp$.
$A[j,k1:k2]$	ger delmatrisen till A bestående av elementen i kolonnerna med index $k1$ t.o.m. $k2 - 1$ i raden med index j .
$A[j,:]$	ger delmatrisen till A bestående av elementen i alla kolonnerna i raden med index j . Kommandot ger alltså raden med index j .
$A[[j1,\dots,jp],:]$	ger delmatrisen till A bestående av raderna med index $j1,\dots,jp$.

Observera att delmatriser som motsvarar kolonnvektorer automatiskt omvandlas till radvektorer.

Exempel 5.9. Vi börjar med att definiera en 3×4 -matris A som

```
A = np.array([[1,2,3,4],      # rad med index 0
              [5,6,7,8],      # rad med index 1
              [9,10,11,12]]) # rad med index 2
```

(a) Kommandot

```
B = A[1:3,1:4]
```

ger

```
[[ 6  7  8]
 [10 11 12]]
```

vilket är delmatrisen av A bestående av raderna med index 1 till 2 och kolonnerna med index 1 till 3.

(b) Kommandot

```
B = A[-1,1:4]
```

ger

```
[10 11 12]
```

vilket är delmatrisen av A bestående av sista raden och kolonnerna med index 1 till 3.

(c) Då vi skriver

```
C = A[2,:]
```

får vi utskriften

```
[ 9 10 11 12]
```

vilket är radvektorn i A med index 2.

(d) Då vi skriver

```
D = A[:,1] # omvandlar automatiskt till radvektor
```

ger Python

```
[ 2  6 10]
```

vilket är kolonnvektorn i A med index 1 omvandlad till en radvektor. □

På samma sätt som för vektorer kan vi använda notationen för delmatriser för att tilldela nya värden för elementen i en matris.

5.11 Läsa och skriva variabler till fil

Det är viktigt att kunna läsa och skriva variabler till fil, jämför kapitel 4.3. Filen kan vara en vanlig textfil eller en binär fil i numpyformat. Det finns en mängd kommandon för att läsa och skriva. Här ger vi bara prov på några av de enklaste.

<code>np.savetxt</code>	sparar en variabel i en textfil.
<code>np.loadtxt</code>	läser en variabel från en textfil.
<code>np.save</code>	sparar en variabel i en binär fil i numpyformat.
<code>np.load</code>	läser en variabel från en binär fil i numpyformat.
<code>np.savez</code>	sparar flera variabler till ett binärt arkiv i numpyformat.
<code>np.load</code>	läser flera filer från ett binärt arkiv i numpyformat.

Exempel 5.10.

(a) Följande kommando definierar en 3×2 -matris a som sedan sparas i textfilen **data1.txt**.

```
a = np.array([[1,2],
              [3,4],
              [5,6]])
np.savetxt('data1.txt',a)
```

Filen **data1.txt** har, då vi öppnar den i en vanlig editor, följande utseende

```
1.000000000000000e+00 2.000000000000000e+00
3.000000000000000e+00 4.000000000000000e+00
5.000000000000000e+00 6.000000000000000e+00
```

Talen separeras med blanktecken och varje tal har skrivits ut i exponentform med 19 siffror.

(b) För att läsa talen från filen **data1.txt** och lagra i matrisen *b* ger vi kommandona

```
b = np.loadtxt('data1.txt')
print(b)
```

Python skriver ut

```
[[1. 2.]
 [3. 4.]
 [5. 6.]]
```

(c) Vi kan spara enligt givna format och med talen separerade med blanktecken, komma, semikolon etc. Följande kommando sparar *a* i textfilen **data2.txt**. Talen sparas kommasseparerade.

```
np.savetxt('data2.txt',a, delimiter=',')
```

Filen **data2.txt** har nu följande utseende

```
1.000000000000000e+00,2.000000000000000e+00
3.000000000000000e+00,4.000000000000000e+00
5.000000000000000e+00,6.000000000000000e+00
```

(d) För att läsa talen från filen **data2.txt** och lagra i matrisen *b* ger vi kommandona

```
b = np.loadtxt('data2.txt', delimiter=',')
```

(e) Det är möjlig att läsa specificerade kolonner. För att läsa kolonnen med index 1 och lagra i vektorn *v* ger vi kommandot

```
v = np.loadtxt('data2.txt', delimiter=',', usecols=(1))
print(v)
```

Vi får

```
[2. 4. 6.]
```

□

Exempel 5.11. Vi definierar en vektor

```
x = np.array([1,2,3])
```

(a) Då vi ger kommandot

```
np.save('data1', x)
```

sparas vektorn x i en binär fil i numpyformat som får namnet **data1.npy**.

(b) För att läsa talen från filen **data1.npy** och lagra i vektorn y ger vi kommandona

```
y = np.load('data1.npy')
print(y)
```

Python skriver ut

```
[1 2 3]
```

□

Exempel 5.12. Det går bra att skriva flera vektorer och matriser till samma fil. Vi definierar en vektor och en matris

```
x = np.array([1,2,3])
a = np.array([[0,4],
              [7,6]])
```

(a) Följande kommandon öppnar filen **data2.npy** för skrivning i binärt format. Därefter skrivs vektorn x och matrisen a till filen

```
with open('data2.npy','wb') as f: # wb = write binary
    np.save(f, x)                  # indentera (dra in) 4 positioner
    np.save(f, a)
```

(b) För att läsa talen från filen **data2.npy** och lagra i vektorn x och matrisen a ger vi kommandona

```
with open('data2.npy','rb') as f: # rb = read binary
    x = np.load(f)                # indentera (dra in) 4 positioner
    a = np.load(f)
```

(c) Det finns ett alternativt sätt att skriva flera variabler till fil. Kommandot

```
np.savez('data3',x=x, a=a)
```

skriver x och a till ett binärt arkiv **data3.npz**. Sista delen av kommandot, `x=x, a=a`, gör att de skriva variablerna identifieras med x och a .

(d) För att läsa variablerna i arkivet

```
npzfile = np.load('data3.npz')
```

Alla variabler finns nu i strukturen **npzfile**. För att få fram identifierarna skriver vi

```
npzfile.files
```

och Python svarar

```
['x', 'a']
```

För att spara innehållet i strukturen **npzfile** till variablerna x och a ger vi kommandona

```
x = npzfile['x']
a = npzfile['a']
```

Utskrift av variablerna ger det förväntade svaret. □

5.12 Aritmetiska operationer på fält

Vektorer och matriser kan multipliceras med tal (skalärer). Vektorer och matriser med samma dimension kan också adderas och subtraheras. Multiplikation med skalär samt addition och subtraktion sker elementvis

$$3 \cdot \begin{pmatrix} 1 & 2 \\ 3 & 0 \end{pmatrix} = \begin{pmatrix} 3 & 6 \\ 9 & 0 \end{pmatrix} \quad \text{och} \quad \begin{pmatrix} 1 & 2 \\ 3 & 0 \end{pmatrix} + \begin{pmatrix} -1 & 1 \\ 2 & 4 \end{pmatrix} = \begin{pmatrix} 0 & 3 \\ 5 & 4 \end{pmatrix}.$$

I Python kan man även addera ett tal med en vektor eller matris. Man tänker sig då att talet adderas till varje vektor och till varje matriselement. Det är praktiskt att utöka operationerna på vektorer och matriser till att omfatta även elementvis multiplikation, division, och upphöjt till.

- + addition, matriserna ska ha samma dimension.
- subtraktion, matriserna ska ha samma dimension.
- * elementvis multiplikation, matriserna ska ha samma dimension.
- / elementvis division, matriserna ska ha samma dimension.
- ** elementvis upphöjt till.

Python använder så kallad broadcasting, vilket gör att det i vissa fall går att t.ex. addera matriser med olika dimension. Även om detta kan vara effektivt, avråder vi från att använda broadcasting till annat än att addera ett tal till varje vektor och till varje elementen i en vektor eller matris.

Exempel 5.13. Vi definierar två vektorer och två matriser

```
x = np.array([1,2,6])
y = np.array([-1,1,2])

A = np.array([[1,1],
              [4,6]])
B = np.array([[4,1],
              [2,3]])
```

(a) Multiplikation med skalär (tal)

```
z = 3*x      # varje element i x multipliceras med 3
print(z)
```

Varje element i x multipliceras med 3

[3 6 18]

(b) Addition av en vektor och ett tal

```
u = x + 2    # varje element i x adderas med 2,
print(u)      # exempel på så kallad broadcasting
```

Varje element i x adderas med 2

[3 4 8]

(c) Elementvis multiplikation

```
v = x*y      # varje element i x multipliceras med motsvarande
print(v)     # element i y
```

Varje element i x multipliceras med motsvarande element i y

$[-1 \ 2 \ 12]$

(d) Elementvis upphöjt till

```
w = x**2      # varje element i x upphöjs till 2 (kvadreras)
print(w)
```

Varje element i x upphöjs till 2

$[1 \ 4 \ 36]$

(e) Addition av matriser

```
C = A + B
print(C)
```

Matriserna adderas elementvis

$[[5 \ 2]
[6 \ 9]]$

(f) Elementvis division av matriser

```
D = A/B
print(D)
```

Varje element i matrisen A divideras med motsvarande element i B

$[[0.25 \ 1.]
[2. \ 2.]]$

(g) Observera att elementvis multiplikation för matriser inte får blandas samman med matrismultiplikation definierat i linjär algebra.

Elementvis multiplikation

```
E = A*B      # elementvis multiplikation
print(E)
```

Varje element i A multipliceras med motsvarande element i B och vi får

$[[4 \ 1]
[8 \ 18]]$

□

5.13 Elementvisa funktioner

En elementvis funktion, även kallad en universell funktion, accepterar både vektorer och matriser som argument. Funktionsberäkningarna sägs vara vektoriserade och sker elementvis. Beroende på funktionen, kan typen ändras jämfört med typen för argumentet.

Nedan följer en förteckning av elementvisa matematisk funktioner. Funktionerna accepterar både vektorer och matriser som argument och returnerar vektorer och matriser av samma form (eng. shape) som argumentet. Typen kan ändras beroende på vilken funktion det är. Vi återkommer till alla dessa funktioner i senare kapitel. I tabellen nedan kan x vara ett tal, en vektor eller en matris.

<code>npfabs(x)</code>	absolutbeloppet $ x $.
<code>np.sqrt(x)</code>	kvadratroten \sqrt{x} .
<code>np.exp(x)</code>	exponentialfunktionen e^x .
<code>np.log(x)</code>	naturliga logaritmen $\ln x$.
<code>np.log10(x)</code>	10-logaritmen $\log_{10} x$.
<code>np.sin(x)</code>	$\sin x$ där x är radianer.
<code>np.cos(x)</code>	$\cos x$ där x är radianer.
<code>np.asin(x)</code>	$\arcsin x$, resultatet i radianer.
<code>np.acos(x)</code>	$\arccos x$, resultatet i radianer.
<code>np.trunc(x)</code>	x tar bort decimaler, behåller heltalsdelen.
<code>np.round(x,n)</code>	avrundar till angivet antal decimaler.
<code>np.floor(x)</code>	avrundar nedåt (floor är golv).
<code>np.ceil(x)</code>	avrundar uppåt (ceil är tak).
<code>np.pi</code>	$\pi \approx 3.141592653589793$.
<code>np.e</code>	$e \approx 2.718281828459045$.

Vi tar ett exempel för att visa på användningen av de vektoriserade funktionerna.

Exempel 5.14. Vi definierar en vektor

```
x = np.array([1, 4, 9, 16])
```

och en matris

```
A = np.array([[1.1, 1.9],  
             [2.5, 3.6]])
```

(a) Då vi skriver

```
y = np.sqrt(x)      # rotens ur av vart och ett av elementen i x  
print(y)
```

svarar Python

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

Elementen i vektorn y fås genom att dra rotens ur motsvarande element i vektorn x .

(b) Kommandona

```
B = np.round(A)      # avrundning av vart och ett av elementen i A  
print(B)
```

ger

```
[[1. 2.]  
 [2. 4.]]
```

Elementen i matrisen B fås genom att avrunda motsvarande element i matrisen A . □

5.14 Aggregerings- och lokaliseringarfunktioner

Bland de universella funktionerna finns även så kallade aggregeringsfunktioner som summerar, medelvärdesbildar etc. elementen i en vektor eller matris. Denna typ av operationer är mycket användbara och kan appliceras antingen på alla element eller på elementen längs en given dimension (axel).

Lokaliseringarfunktionerna bestämmer största eller minsta värde av element i en vektor eller matris samt dessa värdes positioner. På samma sätt som för aggregeringsfunktionerna kan vi bestämma största eller minsta värde av alla element eller av elementen längs en given axel. I kommandona nedan låter vi a vara en vektor eller matris.

<code>np.sort(a)</code>	sorterar elementen i a .
<code>np.mean(a)</code>	medelvärdet av elementen i a .
<code>np.median(a)</code>	medianen av elementen i a .
<code>np.std(a)</code>	standardavvikelsen av elementen i a .
<code>np.sum(a)</code>	summan av elementen i a .
<code>np.prod(a)</code>	produkten av elementen i a .
<code>np.min(a)</code>	minsta elementet i a .
<code>np.max(a)</code>	största elementet i a .
<code>np.argmin(a)</code>	index för minsta elementet.
<code>np.argmax(a)</code>	index för största elementet.
<code>unravel_index</code>	omvandlar från linjärt index till en tupel av index.

Exempel 5.15. Vi har en 3×2 -matris

$$A = \begin{pmatrix} 1 & 5 \\ 4 & 0 \\ 2 & 8 \end{pmatrix},$$

vilken skrivs som

```
A = np.array([[1,5],
             [4,0],
             [2,8]])
```

(a) Följande kommando sorterar elementen i A längs axel 1 (default)

```
B = np.sort(A,axis=1)      # axis=1, sortera elementen i varje rad
print(B)
```

Vi får

```
[[1 5]
 [0 4]
 [2 8]]
```

(b) För att sortera elementen i A längs axel 0 ger vi kommandot

```
B = np.sort(A,axis=0)      # axis=0, sortera elementen i varje kolonn
print(B)
```

Detta ger

```
[[1 0]
 [2 5]
 [4 8]]
```

(c) För att summa alla element i matrisen skriver vi

```
np.sum(A)
```

och får

20

(d) För att summa elementen i A längs axel 0 skriver vi

```
s0 = np.sum(A, axis=0)      # axis=0, beräkna summan av varje kolonn
print(s0)
```

Resultatet blir en vektor

[7 13]

där första elementet i vektorn är summan av elementen i kolonn 1 och andra elementet är summan av elementen i kolonn 2.

(e) För att summa elementen längs axel 1 skriver vi

```
s1 = np.sum(A, axis=1)      # axis=1, beräkna summan av varje rad
print(s1)
```

Resultatet blir en vektor

[6 4 10]

där första elementet är summan av elementen i rad 1, andra elementet är summan av elementen i rad 2 och tredje elementet är summan av elementen i rad 3.

(f) För det bestämma det största elementet i hela matrisen skriver vi

```
np.max(A)
```

och får

8

För att få ett linjärt index (index med start i 0 vid radvis uppräknande av elementen) för maximum ger vi kommandot

```
np.argmax(A)
```

Python svarar

5

Kommandot `unravel_index` tillsammans med formen $(3, 2)$ av matrisen A omvandlar linjärt index för största värde till rad- och kolonndindex (m, n)

```
(m, n) = np.unravel_index(np.argmax(A), A.shape)
```

Efter kommandot får m värdet 2 och n värdet 1. Största elementet står i tredje raden (radindex 2) och andra kolonnen (kolonndindex 1).

(g) För att bestämma de största värdena längs axel 0 skriver vi

```
max0 = np.max(A, axis=0)      # axis=0, bestäm maxvärdet i varje kolonn
print(max0)
```

Resultatet blir en vektor

[4 8]

där första elementet är det största värdet i kolonn 1 och andra elementet är det största värdet i kolonn 2.

(h) För att bestämma index längs axel 0 för de största värdena ger vi kommandot

```
argmax0 = np.argmax(A, axis=0)
print(argmax0)
```

Vi får

[1 2]

där första elementet är radindex av största värdet i kolonn 1 och andra elementet är radindex av det största värdet i kolonn 2. \square

Exempel 5.16. Vi har en 3×4 -matris

$$A = \begin{pmatrix} 1 & 2 & 3 & 0 \\ 8 & 6 & 4 & 5 \\ 9 & 8 & 3 & 1 \end{pmatrix}.$$

Matrisen skrivs som

```
A = np.array([[1,2,3,0],
             [8,6,4,5],
             [9,8,3,1]])
```

(a) Vi kan operera på delar av matrisen. Kommandot

```
np.mean(A[0:2,1:4])
```

ger medelvärdet av elementen med radindex från 0 till och med 1 och kolonndindex från 1 till och med 3. Vi får

3.3333333333333335

(b) För att beräkna medianen av elementen i sista kolonnen ger vi kommandot

```
np.median(A[:,3]) # även np.median(A[:, -1])
```

Python returnerar

1.0

5.15 Tillämpning: temperaturdata

I filen **T10365.txt**, som laddas ner från Canvas, finns dagliga temperaturvärden från en mätstation i Skåne. Mätserien omfattar 10 år från 1 januari 1981 till och med 31 december 1990 och ligger lagrad i en 10×365 -matris (vi bortser från skottår). Vi ska skriva ett program, **temperatur.py**, som läser in temperaturvärdena och bestämmer följande kvantiteter:

1. maximal och minimal temperatur under de 10 åren
2. medeltemperatur under de 10 åren
3. medeltemperatur för januari och juni under de 10 åren
4. medeltemperatur för 1981 och 1990.

Vi börjar med att rita temperaturmatrisen T , se figur 5.1, för att få en uppfattning om vilka delmatriser som behöver användas för att beräkna de olika kvantiteterna.

	Jan	Feb	Mar	Apr	Maj	Jun	Jul	Aug	Sep	Okt	Nov	Dec
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
Dag 1 – 365												

Figur 5.1: Matris innehållande temperaturvärden. Notera att i Python motsvaras år 1 av index 0, år 2 av index 1 etc. På samma sätt motsvaras dag 1 av index 0, dag 2 av index 1.

Efter att temperaturmatrisen har lästs in med `loadtxt` kan maximal och minimal temperatur samt medeltemperatur under de 10 åren beräknas genom upprepad användning av kommandona `np.max`, `np.min` och `np.mean` på hela matrisen. Medeltemperaturen för januari under de 10 åren fås genom användning av kommandot `np.mean` på delmatrisen bestående av kolonnerna med index 0 till 30. Medeltemperaturen för juni under de 10 åren fås genom användning av `np.mean` på hela delmatrisen bestående av kolonnerna med index 151 till 180. Medeltemperaturen för 1981 och 1990 åren fås genom att använda kommandot `np.mean` på raderna med index 0 respektive 9. Tips: för att reda ut dagnummer och index för olika månader kan vi ta hjälp av en dagnummerkalender för ett år som inte är skottår <https://www.kalender-365.se/dagnummer/2023.html>. För utskrift använder vi kommandot `print`.

Vi har nu all information vi behöver för programmet.

```
# temperatur.py
# beräknar olika temperaturmedelvärden
import numpy as np

# Läs in temperaturvärden från filen T10365.txt.
# Lagra i 10 x 365 matrisen T
T = np.loadtxt('T10365.txt')

# max, min, medelvärde för 1981-90, avrunda till en decimal
print()          # ger en tom rad
print('Maximal temperatur under 10 år', np.round( np.max(T),1 ) )
print('Minimal temperatur under 10 år', np.round( np.min(T),1 ) )
print('Medeltemperatur under 10 år   ', np.round( np.mean(T),1 ) )
```

```
# medeltemp. för jan. (dag 1-31) och jun. (dag 152-181), avrunda till en decimal
print()
print('Medeltemperatur januari ', np.round( np.mean(T[:,0:31]),1 ) )
print('Medeltemperatur juni     ', np.round( np.mean(T[:,151:181]),1 ) )

# årsmedelvärde för 1981 till 1990, avrunda till en decimal
print()
print('Årsmedeltemperatur för 1981 ', np.round( np.mean(T[0,:]),1 ) )
print('Årsmedeltemperatur för 1990 ', np.round( np.mean(T[9,:]),1 ) )
```

Vid noterar att dag 1–31 (januari) motsvarar index 0–30 och eftersom man ska addera ett i indexnotationen blir det `np.mean(T[:,0:31])`. På samma sätt har vi att dag 152–181 (juni) motsvarar index 151–180 och vi skriver därför `np.mean(T[:,151:181])`. Man behöver alltid vara vaksam på index och det är mycket lätt att göra fel. Vi har lagt in `print()` för att skriva ut en tom rad. Vidare har vi lagt in extra blanktecken i kommandona för att det ska bli lättare att läsa.

För att köra kommandona i filen klickar vi på den gröna triangeln i Spyders verktygsfält, se figur 2.2, eller skriver

```
runfile('temperatur.py')
```

vid kommandoprompten varvid vi får följande utskrift på skärmen

```
Maximal temperatur under 10 år 25.0
Minimal temperatur under 10 år -16.9
Medeltemperatur under 10 år    7.5

Medeltemperatur januari -0.8
Medeltemperatur juni     14.7

Årsmedeltemperatur för 1981  7.7
Årsmedeltemperatur för 1990  7.8
```

Kapitel 6

Plottnings och grafik

Grafik är viktigt för att tolka och presentera resultat från beräkningar och simuleringar. Python har en mängd kommandon för att plotta och visualisera olika typer av data. Genererade figurer kan sparas som filer i olika grafiska format och importeras i till exempel MSWord- eller LaTeX-dokument.

6.1 Matplotlib – pyplot

Matplotlib är ett paket för att plotta och visualisera olika typer av data. Paketet är mycket stort med en mängd moduler. Den modul vi ska använda här är `pyplot`. Det har blivit standard att modulen `pyplot` importeras genom kommandot

```
import matplotlib.pyplot as plt
```

Funktioner från modulen blir nu tillgängliga via

```
plt.funktionsnamn
```

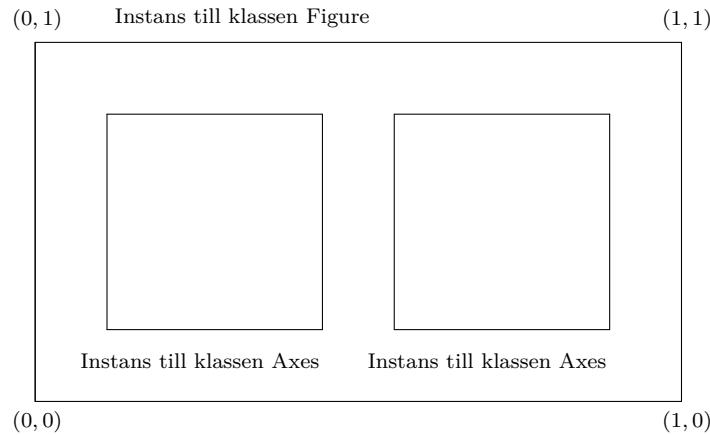
Plottnings objektorienterad, och i `pyplot` definieras klasserna `Figure` och `Axes`. Instanser till dessa två klasser skapas genom

```
fig, ax = plt.subplots()    # fig, ax instanser till klasserna Figure och Axes
```

Instansen `fig` är en rityta, medan instansen `ax` är ett koordinatsystem som kan placeras på ritytan. Formen och storleken på ritytan kan ändras, och till varje rityta kan man placera ett eller flera koordinatsystem. Funktioner och data plottas genom att verka på instansen `ax` med olika metoder

```
ax.plotmetod(data,optioner)
```

På samma sätt som det finns metoder för att plotta, finns det metoder för att sätta ut text, ändra skala på axlarna, addera teckenförklaringar och titel och så vidare. En rityta med koordinatsystem illustreras i figur 6.1.



Figur 6.1: Illustration av instanserna *fig* och *ax*. Instansen *fig* definierar en rityta med ett koordinatsystem sådant att (0,0) definierar nedre vänstra hörnet medan (1,1) definierar övre högra hörnet. Instansen *ax* definierar ett koordinatsystem i vilket vi kan plotta funktioner och data. En eller flera instanser till klassen *Axes* kan placeras på ritytan.

6.2 Exempelgalleriet

Det bästa sättet att få en uppfattning om Pythons grafiska möjligheter är att ta del av exempelgalleriet på <https://matplotlib.org/stable/gallery/index>. Exempelgalleriet visar en mängd olika typer av plottar, och under varje plot ges kommandona som användes för att generera plotten. Alla kommandona som visas är klickbara och leder vidare till Pythons dokumentation. Exempelgallret är alltså utmärkt för ett undersökande arbetssätt: hitta den eller de plottar i exempelgalleriet som liknar det du själv vill åstadkomma, modifiera motsvarande kommando så att din plot blir som du vill ha den. I exempelgalleriet används ett objektorienterat arbetssätt, med metoder som verkar på instanser.



Figur 6.2: Exempelgalleriet som hör till matplotlib är en bra utgångspunkt för att lära sig mer om grafik. Gå in och titta på exemplen.

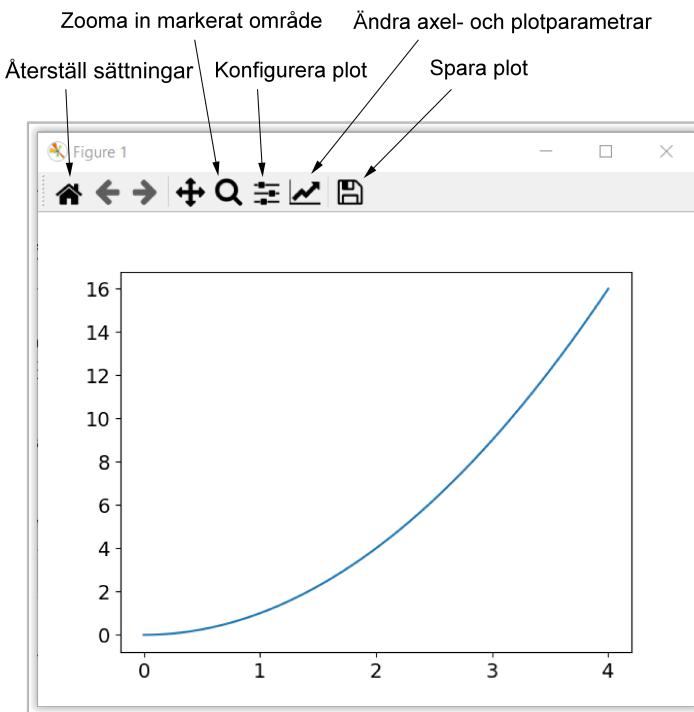
6.3 Skapa och spara en figur

I följande exempel använder vi kommandot `plt.subplots()` för att öppna ett grafikfönster och skapa instanserna `fig` och `ax` till klasserna `Figure` och `Axes`. För att göra själva plotten användar vi metoder som verkar på `ax`.

Exempel 6.1. Vi ska plotta funktionen $y = x^2$ i intervallet $0 \leq x \leq 4$. Den önskade plotten fås genom följande kommandon

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0,4)           # vektor med x-värden
y = x**2                      # vektor med y-värden
fig, ax = plt.subplots()        # skapa instanserna fig och ax
ax.plot(x,y)                  # plotta y mot x
ax.tick_params(labelsize=14)    # sätt storlek på griddmarkeringarna
```

Tredje kommandot genererar 50 stycken x -värden jämnt fördelade mellan 0 och 4. Fjärde kommandot genererar y -värden för funktionen. Femte kommandot öppnar ett grafikfönster och skapar instanserna `fig` och `ax` till klasserna `Figure` och `Axes`. Instansen `ax` placeras automatiskt ut på ritytan. I det näst sista kommandot verkar metoden `plot` på instansen `ax` och plottar funktionen. Plotta, i detta sammanhang, innebär att Python förbindrar punkterna, eller talparen, $(x_0, y_0), (x_1, y_1), \dots$ med räta linjer. Om vi har tillräckligt många x - och y -värden ser funktionskurvan jämnt ut. I det sista kommandot verkar metoden `tick_params` på `ax` och sätter fontstorleken på griddmarkeringarna. Resultatet visas i Spyders grafikfönster (se figur 6.3).



Figur 6.3: Spyders grafikfönster. Ritytan med plotten sparas till fil genom att klicka på ikonen för spara och ange namn och filformat.

Överst i grafikfönstret är namnlistan, som talar om att det är figur 1. Längst upp till höger är knapparna `Minimera`, `Återställ` och `Avsluta`. Under namnlisten finns ett antal verktyg som kan användas för att bland annat zooma in delar av plotten, ändra linjetyp för plotten, ändra

placeringen av koordinatsystemet på ritytan etc. Slutligen finns en box med skalmarkeringar, vilken innehåller grafen. Det är figuren (ritytan) och dess innehåll, och inte hela grafikfönstret, som kan sparas i olika grafiska format.

För att spara figuren klickar man på ikonen för att *spara* och anger namn och format på filen. Alternativt kan vi spara figuren genom att ge kommandot

```
fig.savefig('namn.format')
```

Python stöder bland annat vektorgrafikformaten *svg*, *eps*, *pdf* och rasterformaten *bmp*, *emf*, *jpg*, *tif*, *png*. Vektorgrafiken är skalbar och vi rekommenderar att figurer sparas som *eps* eller *pdf* då de ska importeras i ett LaTeX-dokument.

6.4 Grundläggande plottmetoder

Kommandot `plt.subplots` öppnar ett grafikfönster och skapar instanser till klasserna `Figure` och `Axes`. Öppnade grafikfönster numreras 1, 2, osv. och ligger kvar tills de stängs. Efter det att vi har skapat instanser fås plottar genom att verka på dessa med olika metoder. Det finns en stor mängd metoder, och vi ska här endast titta på några av de vanligaste och mest användbara.

<code>fig,ax=plt.subplots()</code>	öppnar ett grafikfönster. Skapar instanserna <i>fig</i> och <i>ax</i> , till klasserna <code>Figure</code> och <code>Axes</code> . Ritytan har en förutbestämd storlek.
<code>fig,ax=plt.subplots(figsize=(b,h))</code>	öppnar ett grafikfönster. Skapar instanserna <i>fig</i> och <i>ax</i> , till klasserna <code>Figure</code> och <code>Axes</code> . Ritytan får storleken (b, h) .
<code>fig,ax=plt.subplots(m,n,figsize=(b,h))</code>	öppnar ett grafikfönster. Skapar instanserna <i>fig</i> och <i>ax</i> , till klasserna <code>Figure</code> och <code>Axes</code> . Instansen <i>ax</i> är en matris med dimensionen $m \times n$, där <i>vart</i> och <i>ett</i> av elementen definierar ett koordinatsystem. Ritytan får storleken (b, h) .
<code>fig.savefig('namn.format')</code>	sparar ritytan i angiven fil.
<code>plt.clf()</code>	rensar det senaste grafikfönstret.
<code>plt.close()</code>	stänger det senaste grafikfönstret.
<code>plt.close(n)</code>	stänger grafikfönster <i>n</i> .
<code>plt.close('all')</code>	stänger alla grafikfönster.
<code>ax.plot(y)</code>	plottar vektorn <i>y</i> mot elementens index. En heldragen linje förbinder punkterna.
<code>ax.plot(x,y)</code>	plottar vektorn <i>y</i> mot vektorn <i>x</i> . En heldragen linje förbinder punkterna.
<code>ax.plot(x,y,str)</code>	plottar vektorn <i>y</i> mot vektorn <i>x</i> . <i>str</i> är en teckensträng, som talar om vilken linjetyp, punkttyp eller färg man önskar på kurvan. De olika möjligheterna ges i tabell 6.1.
<code>ax.plot(x1,y1,str1, x2,y2,str2,...)</code>	plottar vektorn <i>y1</i> mot vektorn <i>x1</i> enligt <i>str1</i> , vektorn <i>y2</i> mot vektorn <i>x2</i> enligt <i>str2</i> osv.
<code>ax.errorbar(x,y,dy)</code>	plottar vektorn <i>y</i> mot vektorn <i>x</i> med felgränserna $[y - dy, y + dy]$ i <i>y</i> .
<code>ax.errorbar(x,y, dy,dx)</code>	plottar vektorn <i>y</i> mot vektorn <i>x</i> med felgränserna $[y - dy, y + dy]$ och $[x - dx, x + dx]$.
<code>ax.semilogx(x,y)</code>	plottar vektorn <i>y</i> mot vektorn <i>x</i> med en \log_{10} -skala för <i>x</i> och en linjär skala för <i>y</i> .
<code>ax.semilogy(x,y)</code>	plottar vektorn <i>y</i> mot vektorn <i>x</i> med en linjär skala för <i>x</i> och en \log_{10} -skala för <i>y</i> .
<code>ax.loglog(x,y)</code>	plottar vektorn <i>y</i> mot vektorn <i>x</i> med \log_{10} -skalar för både <i>x</i> och <i>y</i> .
<code>ax.grid('on')</code>	ritar ett rutnät.

<code>ax.grid('off')</code> tar bort rutnät. <code>ax.tick_params</code> ändra egenskaperna för griddmarkeringar (eng. ticks), axeltext och griddlinjer.

Genom att i `ax.plot` ange en teckensträng kan man specificera den linjetyp, punkttyp eller färg man önskar på kurvan. En del av de olika möjligheterna är listade i tabell 6.1 Olika typer kan kombineras. Så ger till exempel '`ro`' röda ringar medan '`g--`' ger en grön streckad linje.

Tabell 6.1: *Punkt-, linje- och färgtyper för metoden `ax.plot`.*

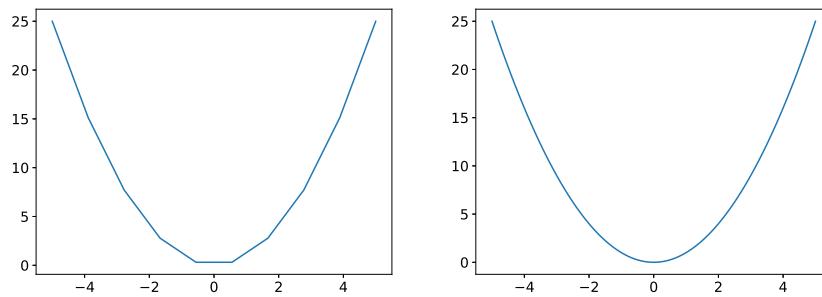
Punkttyper	Linjetyper	
.	- heldragen linje (solid)	
*	-- streckad linje (dashed)	
o	-. punkt-streckad linje (dashdot)	
+	:	prickad linje (dotted)
x		
<		
>		
^		
v		
Färgtyper		
g	grön	
m	magenta	
b	blå	
c	cyan	
k	svart	
y	gul	
r	röd	

Vi illustrerar de grundläggande plottmetoderna med ett antal exempel.

Exempel 6.2. För att plotta funktionen $y = x^2$ i intervallet $-5 \leq x \leq 5$ med heldragen linje skriver man

```
x = np.linspace(-5,5,10)      # för få x-värden, kantig plot
y = x**2
fig, ax = plt.subplots()
ax.plot(x, y)
ax.tick_params(labelsize=14)
```

Den skapade plotten, funktionskurvan, visas till vänster i figur 6.4. Plotten är kantig eftersom vi har få värden på x . Om vi i stället definierar x genom kommandot `x = np.linspace(-5,5,100)`, dvs. tar hundra värden jämnt fördelade mellan 0 och 10, så får vi den jämma plotten till höger i figur 6.4. \square



Figur 6.4: Funktionen $y = x^2$ plottad med olika antal punkter.

Exempel 6.3.

(a) Man kan plotta med olika punkttyper. Kommandona

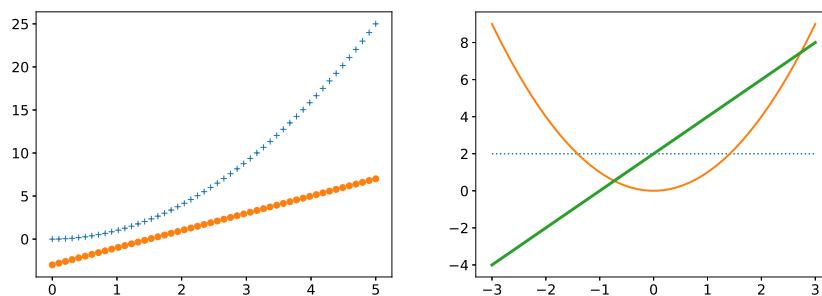
```
x = np.linspace(0,5)
y1 = x**2
y2 = 2*x - 3
fig, ax = plt.subplots()
ax.plot(x,y1,'+',x,y2,'o')
ax.tick_params(labelsize=14)
```

ger plotten till vänster i figur 6.5.

(b) Det går bra att använda skilda linjetyper och linjevidder. Kommandona

```
x = np.linspace(-3,3)
y1 = 2 + 0*x      # vektor där varje element är 2, bara y1 = 2 fungerar inte
y2 = x**2
y3 = 2*x + 2
fig, ax = plt.subplots()
ax.plot(x,y1,:')          # prickad
ax.plot(x,y2,linewidth=2)
ax.plot(x,y3,linewidth=3)
ax.tick_params(labelsize=14)
```

ger plotten till höger i figur 6.5. Genom att använda optionen `linewidth` kan man ändra linjevidden och på så sätt skilja olika kurvor åt. \square

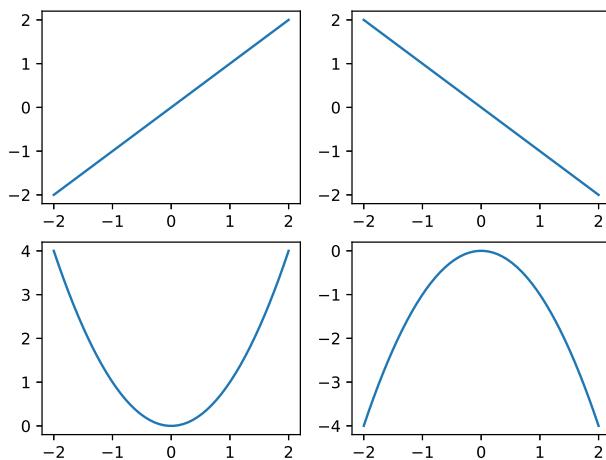


Figur 6.5: Funktioner med olika punkt- och linjetyper.

Exempel 6.4. Då vi skriver `fig, ax = plt.subplots(m,n)` blir instansen `ax` en matris med dimension $m \times n$, där varje vart och ett av elementen definierar ett koordinatsystem. Kommandona

```
fig, ax = plt.subplots(2,2)    # ax blir 2 x 2 matris
x = np.linspace(-2,2)
y0 = x
y1 = -x
y2 = x**2
y3 = -x**2
ax[0,0].plot(x,y0)          # uppe till vänster, index (0,0)
ax[0,1].plot(x,y1)          # uppe till höger, index (0,1)
ax[1,0].plot(x,y2)          # nere till vänster, index (1,0)
ax[1,1].plot(x,y3)          # nere till höger, index (1,1)
```

ger plotten i figur 6.6. □



Figur 6.6: Grafikfönstret uppdelat i 2×2 delfönster.

6.5 Axlar och skalning

Python skalar automatiskt axlarna och anpassar dem efter den punktmängd eller kurva som ska plottas. Ibland har man dock anledning att själv välja skala på axlarna. Man vill kanske också bestämma vilka skalmarkeringar som ska visas.

<code>ax.axis('on')</code>	ritar ut axlar.
<code>ax.axis('off')</code>	ritar inte ut några axlar.
<code>ax.axis('equal')</code>	skaldelarna blir lika stora på x - och y -axlarna.
<code>ax.axis([x1,x2,y1,y2])</code>	ger användaren möjlighet att själv välja skalning på axlarna.
<code>ax.set_xlim([x1,x2])</code>	ger användaren möjlighet att själv välja skalning på x -axeln.
<code>ax.set_ylim([y1,y2])</code>	ger användaren möjlighet att själv välja skalning på y -axeln.
<code>ax.set_xticks([x1,...,xn])</code>	ger användaren möjlighet att själv välja skalmarkeringar på x -axeln.
<code>ax.set_yticks([y1,...,yn])</code>	ger användaren möjlighet att själv välja skalmarkeringar på y -axeln.

```

ax.set_xticklabels([str1,    ger användaren möjlighet att själv välja text på skal-
... ,strn])          markeringar på  $x$ -axeln.
ax.set_yticklabels([str1,    ger användaren möjlighet att själv välja text på skal-
... ,strn])          markeringar på  $y$ -axeln.

```

Exempel 6.5. Låt oss plotta den rationella funktionen

$$y = \frac{1}{x}, \quad x \neq 0.$$

Detta är intressant eftersom funktionsvärdena kommer att gå mot oändligheten då vi närmar oss $x = 0$. Kommandona

```

x = np.linspace(-10,10,500)
y = 1/x
fig, ax = plt.subplots()
ax.plot(x,y)
ax.grid()
ax.tick_params(labelsize=14)

```

ger den vänstra plotten i figur 6.7. Denna plot innehåller ett minimum av information då vi har för stor skala på både x - och y -axeln. Dessutom förbinds punkten precis till vänster om $x = 0$ med punkten precis till höger om $x = 0$, vilket resulterar i ett felaktigt vertikalt streck.

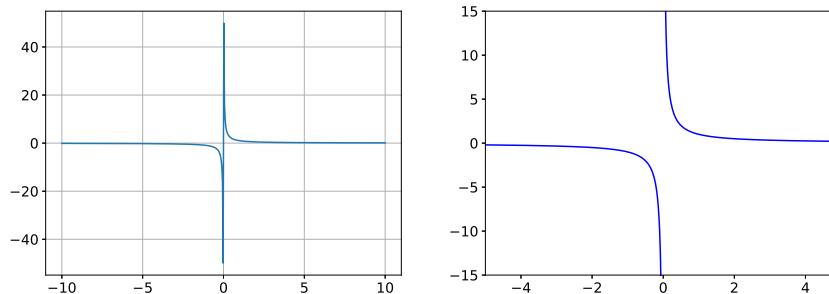
Vi rättar till problemen genom att dela upp plottintervallet i två delar: en del från -10 till en punkt precis till vänster om $x = 0$ och en del från en punkt precis till höger om $x = 0$ till 10 . Lite tester visar sedan vilka gränser, vilka sätts med `axis`, vi ska ha för x och y .

```

x1 = np.linspace(-10,-1.e-10,500)
y1 = 1/x1
x2 = np.linspace(1.e-10,10,500)
y2 = 1/x2
fig, ax = plt.subplots()
ax.plot(x1,y1,'b',x2,y2,'b')    # dela graf i två delar
ax.axis([-5,5,-15,15])           # justera skala (axel gränser)
ax.tick_params(labelsize=14)

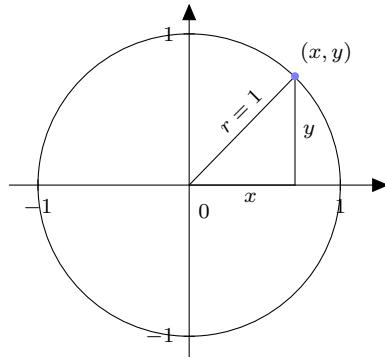
```

Vi får den högra plotten i figur 6.7. □



Figur 6.7: En rationell funktion plottad med olika skalor på axlarna.

Exempel 6.6. Vi ska plotta en cirkel med radie $r = 1$ och centrum i origo. Låt (x, y) vara en punkt på cirkeln. Pythagoras sats ger då att ekvationen $x^2 + y^2 = r^2 = 1$ är uppfylld. Omvänt kan man visa att alla punkter (x, y) som uppfyller $x^2 + y^2 = 1$ ligger på en cirkel med radie 1 och medelpunkt i origo, se figur 6.8.



Figur 6.8: Punkterna (x, y) på en cirkel med radie = 1 och medelpunkt i origo uppfyller ekvationen $x^2 + y^2 = 1$.

Vi löser ut y från ekvationen $x^2 + y^2 = 1$ och har

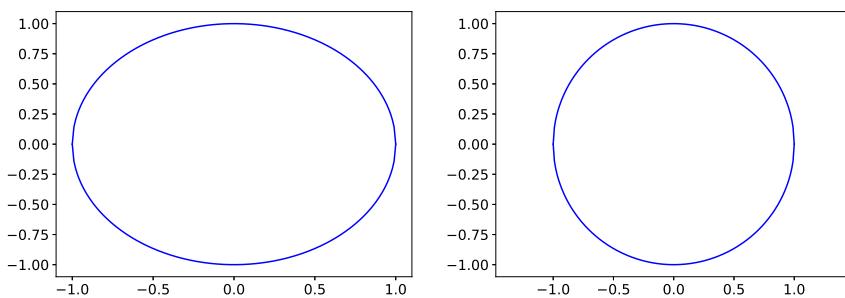
$$y = \pm\sqrt{1 - x^2}, \quad -1 \leq x \leq 1.$$

Vi kan alltså plotta cirkeln genom följande kommandon

```
x = np.linspace(-1,1,200)      # tag många x-värden för jämn plott
y = np.sqrt(1 - x**2)

fig, ax = plt.subplots()
ax.plot(x,y,'b')                # övre delen av cirkeln
ax.plot(x,-y,'b')               # undre delen av cirkeln
ax.tick_params(labelsize=14)
```

Vi får den vänstra plotten i figur 6.9. Det som skall vara en cirkel ser ut som en oval. Anledningen är att vi har olika skaldelar på x - och y -axeln. Om vi ger kommandot `ax.axis('equal')` får vi samma skaldelar på axlarna och plotten är den som visas till höger i figur 6.9. Vi kommer att återkomma till cirklar i exempel 7.5, där vi ska rita en piltavla. \square



Figur 6.9: En cirkel plottad med olika respektive samma skaldelar på x - och y -axeln.

6.6 Text och teckenförklaring

Förutom metoder för att plotta, välja linjetyp och sätta skalning, finns det metoder för att addera titel samt text på x - och y -axlarna. Det går också att placera text på valfria ställen inne i plotten. För att särskilja olika kurvor kan man också lägga till teckenförklaringar (eng. legend).

<code>ax.set_title(txt)</code>	skriver strängen txt överst i grafikfönstret.
<code>ax.set_xlabel(txt)</code>	skriver strängen txt under x -axeln.
<code>ax.set_ylabel(txt)</code>	skriver strängen txt längs y -axeln.
<code>ax.set_text(x,y,txt)</code>	skriver strängen txt i pos. (x,y) på skärmen.
<code>ax.legend(txt)</code>	skriver en ruta med förklaringar till kurvorna.

I den text som placeras ut kan man inkludera grekiska bokstäver och en del matematiska symboler. De grekiska bokstäderna och de matematiska symbolerna är listade i tabell 6.2 och följer LaTeX-konventionerna. Super- och subskript kan också användas och fås med hjälp av $\hat{}$ och $_$ följt av det som ska upphöjas eller nedsänkas inom klammerparentes. Till exempel ger $x^{\{-3\}}$ texten x^{-3} medan $x_{\{1\}}$ ger x_1 . Strängar, vilka Python ska tolka enligt LaTeX-konventionerna, ska föregås av ett `r`. I själva strängen ska de matematiska symbolerna omslutas av dollarstecken, t.ex. `r'$y = x^{-3}$'`.

Tabell 6.2: Grekiska bokstäver och matematiska symboler i LaTeX.

Grekiska bokstäver			
α	<code>\alpha</code>	λ	<code>\lambda</code>
β	<code>\beta</code>	μ	<code>\mu</code>
γ	<code>\gamma</code>	ν	<code>\nu</code>
δ	<code>\delta</code>	ω	<code>\omega</code>
ϵ	<code>\epsilon</code>	ϕ	<code>\phi</code>
κ	<code>\kappa</code>	π	<code>\pi</code>
Matematiska symboler			
\leftarrow	<code>\leftarrow</code>	\Leftarrow	<code>\Leftarrow</code>
\rightarrow	<code>\rightarrow</code>	\Rightarrow	<code>\Rightarrow</code>
\uparrow	<code>\uparrow</code>	\Leftrightarrow	<code>\Leftrightarrow</code>
\downarrow	<code>\downarrow</code>	\geq	<code>\geq</code>
		\leq	<code>\leq</code>
		\pm	<code>\pm</code>
		∂	<code>\partial</code>
		∞	<code>\infty</code>
		\int	<code>\int</code>
		\times	<code>\times</code>

Exempel 6.7.

(a) Följande kommandon plottar funktionerna $y = x^2$ och $y = \sqrt{x}$ i intervallet $0 \leq x \leq 2$. Förutom titel och text på x - och y -axlarna skrivs även en teckenförklaring genom metoden `legend`.

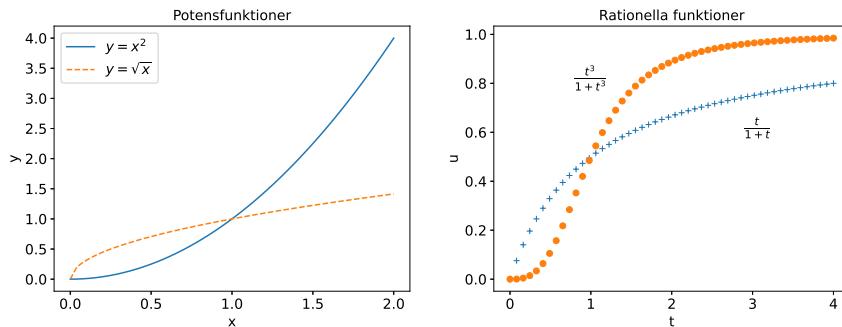
```
x = np.linspace(0,2)
y1 = x**2
y2 = np.sqrt(x)
fig, ax = plt.subplots()
ax.plot(x,y1,label=r'$y=x^2$')           # r gör att sträng tolkas som
                                             # LaTeX-kod
ax.plot(x,y2,'--',label=r'$y=\sqrt{x}$')
ax.set_title('Potensfunktioner',fontsize=14)
ax.set_xlabel('x',fontsize=14)
ax.set_ylabel('y',fontsize=14)
ax.tick_params(labelsize=14)
ax.legend(fontsize=14)
```

Den genererade plotten visas till vänster i figur 6.10.

- (b) I det följande exemplet använder vi olika punkttyper för att plotta funktionerna $u = \frac{t}{1+t}$ och $u = \frac{t^3}{1+t^3}$ i intervallet $0 \leq t \leq 4$. Vi placerar också förklarande text i plotten.

```
t = np.linspace(0,4)                      # generera t-värden
u1 = t/(1 + t)                            # generera u = t/(1 + t)
u2 = t**3/(1 + t**3)                      # generera u = t^3/(1 + t^3)
fig, ax = plt.subplots()
ax.plot(t,u1,'+')                         # plotta (plus)
ax.plot(t,u2,'o')                          # plotta (cirklar)
ax.set_title('Rationella funktioner', fontsize=14)
ax.text(2.9,0.6,r'$\frac{t}{1+t}$', fontsize=16)
ax.text(0.8,0.8,r'$\frac{t^3}{1+t^3}$', fontsize=16)
ax.set_xlabel('t', fontsize=14)  # skriv text på x-axeln
ax.set_ylabel('u', fontsize=14)  # skriv text på y-axeln
ax.tick_params(labelsize=14)
```

Den genererade plotten visas till höger i figur 6.10. □



Figur 6.10: Funktionsplottar med titel, teckenförklaring och text.

Exempel 6.8.

- (a) Följande kommandon plottar $y = 1/(x^\alpha + 1)$ i intervallet $0 \leq x \leq 2$ för några olika värden på α . Observera att för titeln måste vi ha r framför teckensträngen då vi använder oss av LaTeX-symboler. Teckenförklaring skrivas med hjälp av metoden `ax.legend`.

```
x = np.linspace(0,2)
y1 = 1/(x**2 + 1)
y2 = 1/(x**4 + 1)
fig, ax = plt.subplots()
ax.plot(x,y1,label=r'$\alpha = 2$')
ax.plot(x,y2,'--',label=r'$\alpha = 4$')
ax.legend(fontsize=14)
ax.set_title(r'Funktionen $y = \frac{1}{x^{\alpha} + 1}$,
            $0 \leq x \leq 2$', fontsize=14)
ax.tick_params(labelsize=14)
```

Den genererade plotten visas till vänster i figur 6.11.

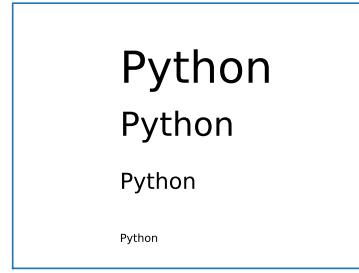
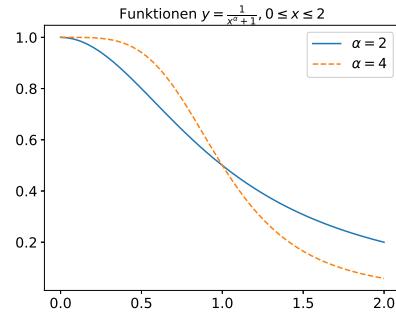
- (b) Det går bra att justera fontstorleken på adderad text med hjälp av optionen `'fontsize'`. Följande kommandon ritar en fyrkant. I fyrkanten skrivas texten Python ut med olika fontstorlek. Kommandot `ax.axis('off')` tar bort axlarna.

```

fix, ax = plt.subplots()
ax.plot([0,10,10,0],[0,0,10,10,0]) # rita fyrkant
ax.text(3,1,'Python',fontsize=10)
ax.text(3,3,'Python',fontsize=20)
ax.text(3,5,'Python',fontsize=30)
ax.text(3,7,'Python',fontsize=40)
ax.axis('off')

```

Den utskrivna texten visas till höger i figur 6.11. □



Figur 6.11: Exempel på typsättning med LaTeX och text med olika fontstorlek. Vi använder kommandon som `frac` och `sqrt` för att typsätta kvoter och kvadratrötter i matematisk font.

6.7 Pilar och förklarande text

Det är ofta värdefullt att inkludera pilar med förklarande text i en plot. I Python finns metoden `ax.annotate`, som har en mängd olika optioner. Först kommer texten och dess storlek, sedan kommer koordinaterna för spetsen och för ändpunkten. Sist kommer optioner som definierar pilens utseende.

<code>ax.annotate(str,spets, slut, pilegenskaper)</code>	sätter ut en pil med angiven position för spets och ändpunkt. Sist anges pilens egenskaper.
--	--

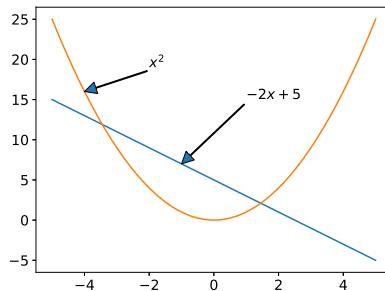
Exempel 6.9. Två funktioner och pilar med förklarande text.

```

x = np.linspace(-5,5)
y1 = -2*x + 5
y2 = x**2
fig, ax = plt.subplots()
ax.plot(x,y1)
ax.plot(x,y2)
ax.tick_params(labelsize=14)
ax.annotate(r'$-2x + 5$', fontsize=14, xy=(-1,7), \ # snedstreck \ bryter raden
            xytext=(1,15), arrowprops=dict(width=1))
ax.annotate(r'$x^2$', fontsize=14, xy=(-4,16), \
            xytext=(-2,19), arrowprops=dict(width=1))

```

Resultatet visas i figur 6.12. Notera hur vi använt snedstrecket \ för radbrytning. □



Figur 6.12: Exempel som visar hur vi kan addera pilar och kommentarer.

6.8 Histogram och stolpdiagram

Innehållet i en vektor kan åskådliggöras med hjälp av stapeldiagram eller histogram. För att skapa ett histogram gör man en klassindelning och räknar antalet element som faller i varje klass. Det hela åskådliggörs sedan med rektanglar, vars höjd är lika med antalet element i varje klass. Normalt använder man en likformig indelning med n klasser som sträcker sig från det minsta till det största elementet i vektorn. I Python finns följande kommandon.

<code>ax.bar(x,y)</code>	ritar ett stapeldiagram över elementen i y .
<code>ax.stem(x,y)</code>	ritar ett stolpdiagram över elementen i y .
<code>ax.hist(x,n)</code>	ritar ett histogram med n klasser över elementen i x .
<code>m, bins, patch = ax.hist(x,n)</code>	ger ett histogram med n klasser över elementen i x . <code>bins</code> är en vektor som definierar intervallen, och <code>m</code> är en vektor med antalet element, frekvensen, i varje klass.

Exempel 6.10. I filen **T.txt**, vilken som vanligt kan laddas ner från Canvas, finns vektorn T med uppmätta temperaturvärden under 23 år för en mätstation i Skåne. Filen **T.txt** läses med

```
T = np.loadtxt('T.txt')
```

Kommandot

```
fig, ax = plt.subplots()  
ax.hist(T,5,edgecolor='black')    # histogram med 5 klasser  
ax.tick_params(labelsize=14)
```

ger histogrammet till vänster i figur 6.13. För att få antalet element i varje klass tillsammans med klassindelningen ger man kommandot

```
m, bins, patches = ax.hist(T,bins=5,edgecolor='black')  
print(m)                      # antalet element (frekvensen) i varje klass  
print(bins)                    # klassgränser
```

varvid Python svarar

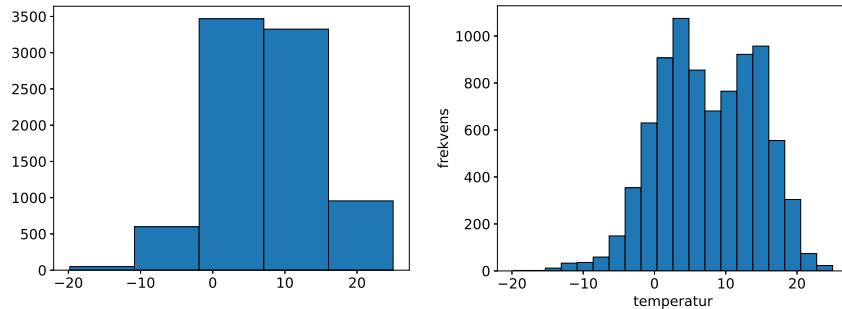
```
[ 49.  598. 3467. 3325.  956.]  
[-19.8 -10.84 -1.88  7.08 16.04 25. ]
```

Första klassen ligger mellan -19.8 och -10.84 och innehåller 49 element. Andra klassen ligger mellan -10.48 och -1.88 och innehåller 598 element och så vidare.

Kommandot

```
ax.hist(T,20,edgecolor='black')    # histogram med 20 klasser
ax.tick_params(labelsize=14)
```

ger histogrammet till höger i figur 6.13. Detta histogram visar tydligt att vår- och hösttemperaturer i intervallet 7°C till 10°C är mindre vanliga än sommar- och vintertemperaturer. Temperatur är ett bra exempel på data som inte är normalfördelad. \square



Figur 6.13: Histogram över observerade temperaturer under 23 år för en mätstation i Skåne. Histogrammet till höger innehåller 20 klasser och visar att vår- och hösttemperaturer i intervallet 7°C till 10°C är mindre vanliga än sommar- och vintertemperaturer.

Exempel 6.11. Kommandot

```
x = np.random.rand(1000000)      # likformigt fördelade mellan 0 och 1
```

ger en vektor med en miljon slumptal likformigt fördelade mellan 0 och 1. För att generera ett histogram med 10 klasser ger vi kommandot

```
fig, ax = plt.subplots()
ax.hist(x,10,edgecolor='black')
ax.tick_params(labelsize=14)
```

varvid vi får plotten till vänster i figur 6.14.

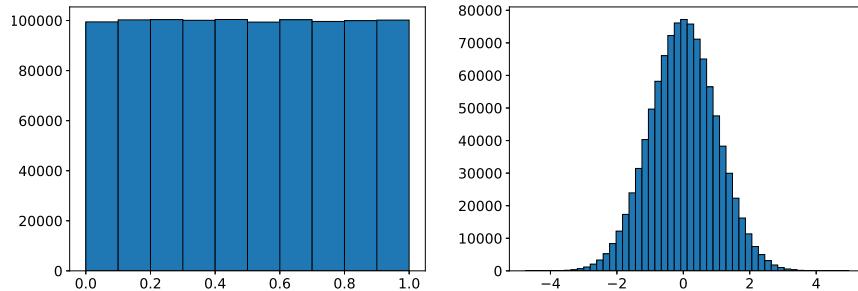
Kommandot

```
x = np.random.randn(1000000)    # normalfördelade med medelv 0 och std 1
```

ger en vektor med en miljon slumptal som följer en normalfördelning med medelvärdet $\mu = 0$ och standardavvikelsen $\sigma = 1$ (se avsnitt 5.9). För att generera ett histogram med 50 klasser ger vi kommandot

```
fig, ax = plt.subplots()
ax.hist(x,50,edgecolor='black')
ax.tick_params(labelsize=14)
```

vilket ger plotten till höger i figur 6.14. För normalfördelade slumptal ska 68 % av talen falla i intervallet $\mu \pm \sigma$, lite mer än 95 % i intervallet $\mu \pm 2\sigma$ och 99.7 % i intervallet $\mu \pm 3\sigma$. \square



Figur 6.14: Till vänster är ett histogram över likformigt fördelade slumptal mellan 0 och 1. Till höger är ett histogram över normalfördelade slumptal med medelvärdet 0 och standardavvikelsen 1.

Exempel 6.12. För att illustrera stapel- och stolpdiagram definierar vi vektorerna x och y med vardera tio element

```
x = np.array([1,2,3,4,5,6,7,8,9,10])
y = np.array([1,-1,0.5,2,3,5,-2,-4,4,9])
```

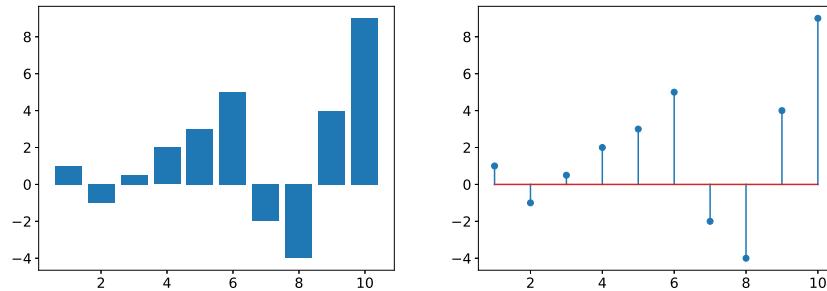
Då vi sedan ger kommandona

```
fig, ax = plt.subplots()
ax.bar(x,y)
ax.tick_params(labelsize=14)
```

respektive

```
fig, ax = plt.subplots()
ax.stem(x,y)
ax.tick_params(labelsize=14)
```

får vi plottarna i figur 6.15. □

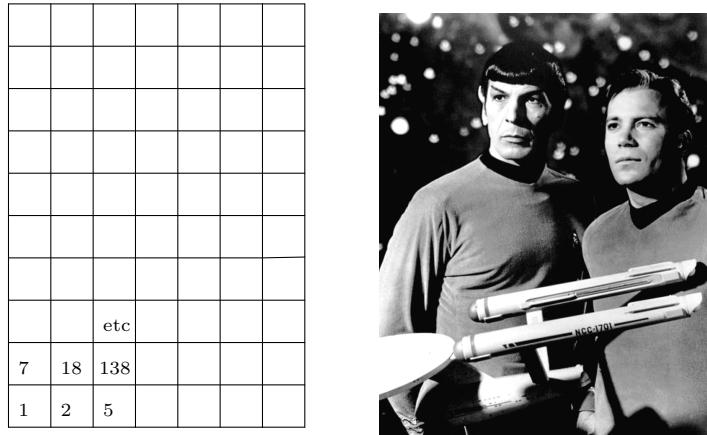


Figur 6.15: Stapel- och stolpdiagram.

6.9 Bilder

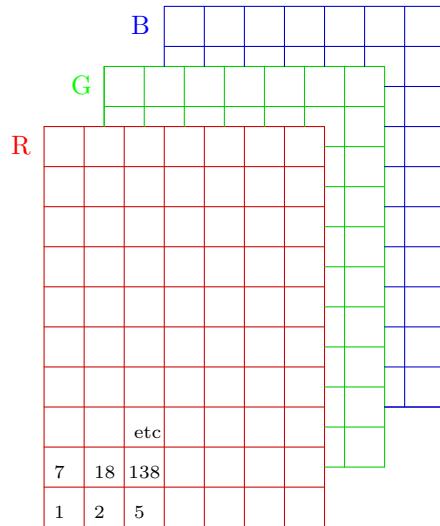
En digital bild består av pixlar, eller bildpunkter. Ju fler pixlar desto bättre upplösning. I en svartvit bild består pixlarna av talvärden lagrade i en matris med index i,j , där i anger raden

och j kolonnen. Normalt ligger talvärdena mellan 0 och 255 och representerar olika nyanser av grått. Talvärdet 0 representerar svart medan talvärdet 255 representerar vitt. Den digitala svart-vita bilden **starwars.jpg** visas till höger figur 6.16. Motsvarande matris med talvärdien visas till vänster.



Figur 6.16: En svartvit digital bild består av talvärden mellan 0 och 255 lagrade i en matris. Talvärdena representerar olika nyanser av grått. I matrisen anger index i raden och index j kolonnen.

I en färgbild (rgb) består varje pixel av tre värden för respektive röd, grön och blå lagrade i tre stackade matriser. För Python är stackade matriser ett fält med tre index i, j, k , där index i anger vilken vilken rad, index j anger vilken kolonn och index k vilken av de tre stackade matriserna. Lagringen av en färgbild som tre stackade matriser visas i figur 7.13.



Figur 6.17: En färgbild består av tre lager av talvärden, normalt mellan 0 och 255, lagrade i tre stackade matriser. Talvärdena i den första matrisen anger andelen rött (R), talvärdena i den andra andelen grönt (G), och talvärdena i den tredje anger andelen blått (B). Tre stackade matriser utgör fält med tre index i, j, k , där index i anger vilken vilken rad, index j anger vilken kolonn och index k vilken av de tre stackade matriserna.

6.10 Matriser och bilder

Tvådimensionella matriser A kan åskådliggöras som bilder (eng. images) genom kommandot `imshow`. Matriselementen a_{jk} specificerar därvid färgerna på pixlarna i bilden. Färgerna är beroende av matriselementens värden och anges av en så kallad färgkarta (eng. colormap), se tabell 6.3

Tabell 6.3: *Olika fördefinierade färgkartor.*

hsv	röd-gul-grön-cyan-blå-magenta och tillbaka till rött.
hot	svart-röd-gul-vit färgskala.
gray	linjär gråskala.
bone	gråskala med lite blått.
copper	färgskala i koppartoner.
pink	färgskala i rosa pastellfärgar.
jet	variant av hsv.
cool	färgskala i cyan och magenta.
autumn	färgskala i rött och gult.
spring	färgskala i magenta och gult.
winter	färgskala i blått och grönt.
summer	färgskala i grönt och gult.

Man kan även läsa in bilder från internet eller från digitalkameror och lagra dessa bilder som matriser. Om bilden är svartvit erhålls en vanlig tvådimensionell matris, där elementvärdena motsvarar andelen grått i pixlarna. Om bilden är i färg erhålls ett fält med djupet 3 (tre stackade matriser). Bildmatriser kan processeras för att till exempel upptäcka mönster eller föremål. Processerade matriser plottas med hjälp av kommandot `imshow`.

<code>ax.imshow(A)</code>	visar matrisen A som en bild. Alla elementvärden, från det minsta till det största, sprids ut i färgkartan. Om ingen färgkarta anges används jet.
<code>ax.imshow(A,cmap, vmin=low,vmax=high)</code>	visar matrisen A som en bild. Elementvärden i intervallet <code>[low, high]</code> sprids ut i färgkartan. Värden utanför intervallet får den första eller sista färgen i färgkartan.
<code>fig.colorbar A = plt.imread('filnamn(fmt)'</code>	ger en förklarande färgkarta i form av en vertikal stapel. läser in en bild från en fil med namnet <i>filnamn</i> och format <i>fmt</i> . Bland filformat som accepteras är bmp , hdf , jpg , pcx , tiff . Bilden lagras i matrisen A som åtta bitars heltal utan tecken (<code>uint8</code> heltal från 0 till 255 i steg om 1).

Exempel 6.13. Vi definierar en 8×10 -matris A enligt

```
A = np.array([[44, 50, 52, 50, 44, 36, 27, 19, 12, 7 ], \
[58, 66, 69, 66, 58, 48, 36, 25, 16, 9 ], \
[71, 81, 84, 81, 71, 58, 44, 31, 19, 11], \
[81, 91, 95, 91, 81, 66, 50, 34, 22, 13], \
[84, 95, 99, 95, 84, 69, 52, 36, 23, 13], \
[81, 91, 95, 91, 81, 66, 50, 34, 22, 13], \
[71, 81, 84, 81, 71, 58, 44, 31, 19, 11], \
[58, 66, 69, 66, 58, 48, 36, 25, 16, 9]])
```

(a) Kommandona

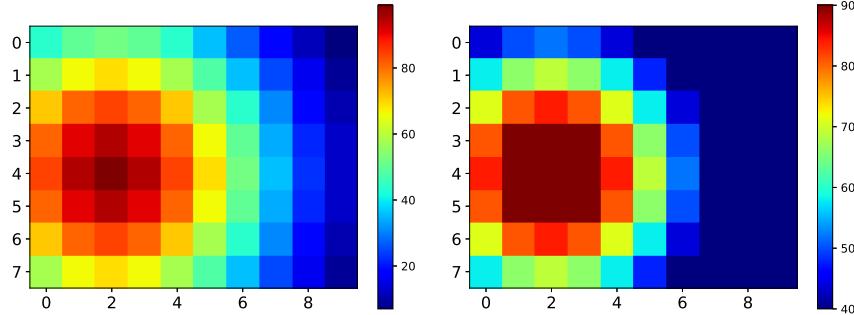
```
fig,ax = plt.subplots()
pos = ax.imshow(A,cmap='jet')
fig.colorbar(pos)
ax.tick_params(labelsize=14)
```

ger figuren 6.18 till vänster där alla elementvärden, från det minsta till det största, har spridits ut i färgkartan.

(b) Kommandona

```
fig,ax = plt.subplots()
pos = ax.imshow(A,vmin=40,vmax=90,cmap='jet') # färgkarta jet
fig.colorbar(pos)
ax.tick_params(labelsize=14)
```

ger figuren 6.18 till höger där elementvärden mellan 40 och 90 har spridits ut i färgkartan. \square



Figur 6.18: Matris åskådliggjord med `imagedesc`.

Exempel 6.14. I den grafiska filen **Lions.jpg** på Canvas finns en bild på jagande grottlejon. Bilden läses in och sparas i en matris A genom

```
A = plt.imread('Lions.jpg')
```

Storleken och typen av matrisen A fås genom kommandot

```
print(A.shape)
print(A.dtype)
```

och vi får utskriften

```
(1721, 2015, 3)
uint8
```

Vi ser att A är en matris med 1 721 rader, 2 015 kolonner och djup 3, där elementen är åtta bitars heltal utan tecken (`uint8`). Bildmatrisen A plottas med

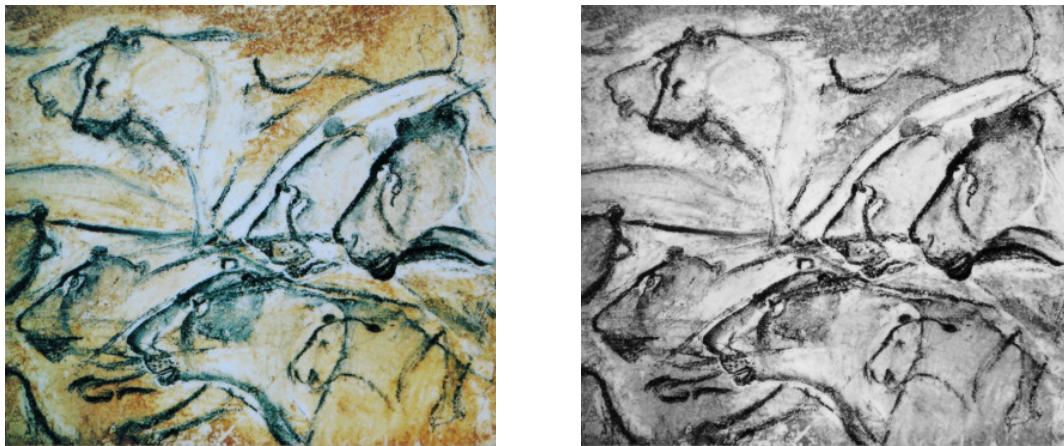
```
fig, ax = plt.subplots()
ax.imshow(A)
ax.axis('equal')      # samma skala på axlarna
ax.axis('off')        # ta bort axlar
```

och ger figuren 6.19 till vänster. □

Exempel 6.15. Som ett exempel på enkel bildbehandling skall vi göra om färgbilden på grottalejonen till svartvitt genom att medelvärdesbilda över de tre färgerna. Då vi bildbehandlar är det viktigt att vi omvandlar talvärdena i matrisen från åtta bitars heltal utan tecken (`uint8`) till flyttal. Pythonkommandona är

```
A = A.astype(dtype='float')           # obs, omvandla från uint8 till flyttal
B = (A[:, :, 0] + A[:, :, 1] + A[:, :, 2])/3 # medelvärdesbilda
fig, ax = plt.subplots()
ax.imshow(B,cmap='gray')           # plotta matrisen med färgskalan gray
ax.axis('equal')                  # samma skala på axlarna
ax.axis('off')                   # ta bort axlarna
```

Den svarvita bilden visas i figur 6.19 till höger. □



Figur 6.19: Bild som har importerats i Python. Den svartvita bilden till höger har fåtts genom att medelvärdesbilda över färgerna.

6.11 Plotta som i matteboken

Vi avslutar kapitlet om grafik med att visa hur du kan plotta funktioner som i matteboken med två axlar som går genom origo. I Python får man normalt fyra linjer som omsluter en figur. Dessa fyra linjer refereras till som `top spine`, `bottom spine`, `left spine` och `right spine`, där spine betyder ryggrad på svenska. För att få två axlar som går genom origo stänger man först av `top spine` och `right spine` och sedan flyttar man `left spine` och `bottom spine` så att de går genom origo. Detta görs genom

```
ax.spines['top'].set_visible(False)      # stäng av top spine
ax.spines['right'].set_visible(False)    # stäng av right spine
ax.spines['left'].set_position('zero')   # låt left spine gå genom origo
ax.spines['bottom'].set_position('zero') # låt bottom spine gå genom origo
```

Om du är mera bekväm med att använda denna typ av plottar, så för all del, använd dem.

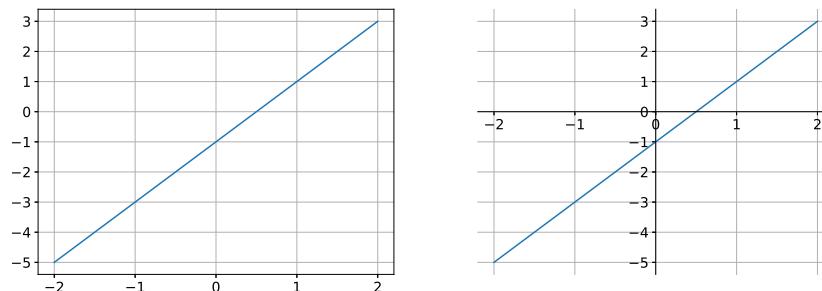
Exempel 6.16. Vi ska plotta funktionen $f(x) = 2x - 1$ i intervallet $[-2, 2]$ tillsammans med två axlar som går genom origo. I bägge fallen använder vi oss av gridlinjer. Kommandona blir

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(-2,2)
y = 2*x - 1
# normal python plot
fig, ax = plt.subplots()
ax.plot(x,y)
ax.tick_params(labelsize=14)
ax.grid('on')

# plot med två-axlar genom origo
fig, ax = plt.subplots()
ax.plot(x,y)
ax.tick_params(labelsize=14)
ax.grid('on')

# dessa kommandon ger axlar genom origo
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_position('zero')
ax.spines['bottom'].set_position('zero')
```

Pythons normala plot visas till vänster i figur 6.20 medan plotten med två axlar genom origo visas till höger. \square



Figur 6.20: Pythons normala plot till vänster och med två axlar genom origo till höger.

Kapitel 7

Programmering

Vi ska här titta på hur vi kan skriva program som löser större uppgifter och problem. I program används väsentligen tre huvudkonstruktioner: sekvens, alternativ och repetition. En sekvens är en följd av kommandon (ibland säger vi också satser) som utförs i ordning uppifrån och ned. Alternativ eller vägvalsfunktioner (if-satser) gör det möjligt att utföra olika kommandon beroende på om logiska uttryck är sanna eller falska. Repetitionskonstruktionerna, slutligen, används för att upprepa kommandon. Vi kan upprepa kommandon ett fixt antal gånger (for-loop) eller så länge ett visst logiskt villkor är uppfyllt (while-loop).

7.1 Logiska uttryck

Ett logiskt uttryck har antingen värdet sant eller falskt. I programmering används dessa uttryck i if-satser för att styra en beräkning. Den vanligaste formen av ett logiskt uttryck är en jämförelse mellan två tal. I Python finns följande jämförelseoperatorer.

jämförelseoperator	betydelse
<	mindre än
<=	mindre än eller lika med
==	liko med
>	större än
>=	större än eller lika med

Om ett logiskt uttryck är sant får uttrycket värdet `True`. Om det logiska uttrycket är falsk får uttrycket värdet `False`.

Exempel 7.1. Låt $x = 0$, $y = 2$ och $z = 3$.

(a) Jämförelsen

```
x == y
```

ger resultatet `False`.

(b) Det logiska uttrycket

```
x + y <= z
```

ger resultatet `True`.

(c) Vi kan ta reda på om ett heltal är jämnt eller udda genom att dividera med två och se om resten är noll. Då vi gör jämförelsen

```
5%2 == 0
```

blir resultatet `False` och 5 är alltså udda. \square

Förutom jämförelseoperatorerna har Python ett antal logiska operatorer. De tre vanligaste operatorerna är logiskt och, logiskt eller och logiskt inte

operator	betydelse
<code>and</code>	logiskt och
<code>or</code>	logiskt eller
<code>not</code>	logiskt inte

Med de logiska operatorerna binder man samman logiska uttryck så att ett nytt logiskt uttryck bildas. Följande tabell visar det nya logiska uttryckets sanningsvärde.

logiskt uttryck	utläses	sanningsvärde
<code>A and B</code>	<code>A och B</code>	uttrycket är sant om både <code>A</code> och <code>B</code> är sanna.
<code>A or B</code>	<code>A eller B</code>	uttrycket är sant om antingen <code>A</code> , <code>B</code> eller båda är sanna.
<code>not A</code>	<code>inte A</code>	uttrycket är sant om <code>A</code> är falskt och falskt om <code>A</code> är sant.

Exempel 7.2. Låt $x = 4$, $y = -6$ och $z = -5$.

(a) Det logiska uttrycket

```
0 <= x and x <= 10
```

ger resultatet `True`. Notera att Python till skillnad från de flesta andra programspråk tillåter att man skriver det logiska uttrycket ovan som

```
0 <= x <= 10
```

(b) Det logiska uttrycket

```
0 <= x or 0 <= y
```

ger resultatet `True`.

(c) Det logiska uttrycket

```
0 <= y or 0 <= z
```

ger resultatet `False`.

(d) Det logiska uttrycket

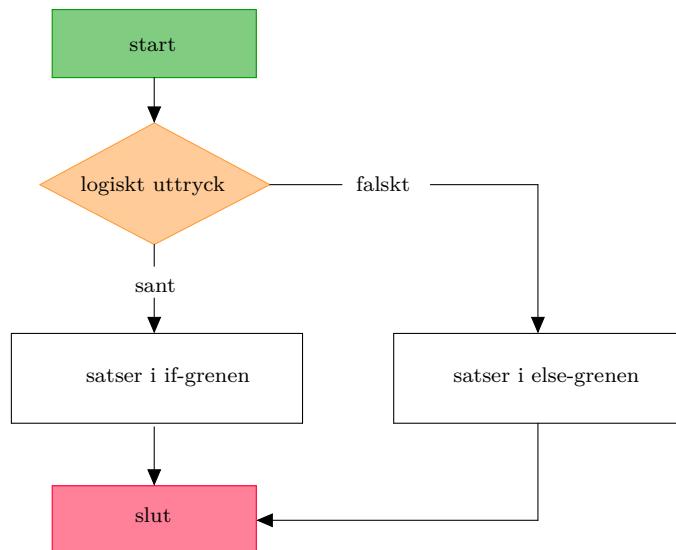
```
not x == y
```

ger resultatet `True`. \square

7.2 If-satser

Alternativ uttrycks i programmering med hjälp av if-satser. Den enklaste formen av en if-sats kan beskrivas som följer: om det logiskt villkoret är sant, utför en eller flera satser i den så kallade

if-grenen, annars om det logiska villkoret är falskt, utför en eller flera satser i den så kallade else-grenen. Denna if-sats illustreras i flödesschemat i figur 7.1.



Figur 7.1: Om det logiska uttrycket är sant, utför en eller flera satser i if-grenen, annars om det logiska uttrycket är falskt, utför en eller flera satser i else-grenen.

I Python uttrycks skriver vi denna if-sats enligt följande

```

if logiskt uttryck:
    satser1      # if-grenen, en eller flera indragna satser
else:
    satser2      # else-grenen, en eller flera indragna satser
satsen efter if-satsen dras inte in
  
```

Om uttrycket är sant utförs **satsen1** annars om det logiska uttrycket är falskt utförs **satsen2**. If-villkor slutar alltid med kolon och följs av indragna (indenterade) satser. Satser efter if-satsen dras inte in. Mera omfattande villkorskonstruktioner skrivs som elseif-satser

```

if uttryck1:
    satser1      # en eller flera indragna satser
elif uttryck2:
    satser2      # en eller flera indragna satser
elif uttryck3:
    satser3      # en eller flera indragna satser
else:
    satser4      # en eller flera indragna satser
satsen efter if-satsen dras inte in
  
```

Om **uttryck1** är sant utförs **satsen1** följt av **satsen5**, annars om **uttryck2** är sant utförs **satsen2** följt av **satsen5**, annars om **uttryck3** är sant utförs **satsen3** följt av **satsen5**. Om inget av de logiska uttrycken ovan är sanna utförs **satsen4**. Satser efter if-satsen dras inte in. Ovanstående konstruktion är generell, och kan byggas på med fler villkor om det är nödvändigt.

Exempel 7.3.

(a) Programmet **evenodd.py** läser in ett heltalet n från terminalen och bestämmer och skriver ut om talet är jämnt eller udda

```
# evenodd.py
n = int(input('Ge ett heltalet '))
if n%2 == 0:
    print('Heltalet är jämnt')
else:
    print('Heltalet är udda ')
```

(b) Programmet **intervall.py** kontrollerar om ett positivt tal x ligger mellan 0 och 10. Om användaren matar in ett tal som inte är positivt skrivas en varning ut.

```
# intervall.py
x = float(input('Ge ett positivt tal '))
if x < 0:
    print('Talet är inte positivt')
elif 0 <= x <= 10:
    print('Talet ligger mellan 0 och 10 ')
else:
    print('Talet är större än 10')
```

Programmet ovan kan även skrivas

```
# intervall.py
x = float(input('Ge ett positivt tal '))
if x < 0:
    print('Talet är inte positivt')
elif 0 <= x <= 10:
    print('Talet ligger mellan 0 och 10 ')
elif x > 10:
    print('Talet är större än 10')
```

Både if-satserna gör samma sak. Detta gäller allmänt: det finns ofta många olika sätt att göra samma sak och vad man väljer är ofta beroende på tycke och smak. \square

Exempel 7.4. En andragradsekvation av typen

$$x^2 + px + q = 0$$

kan ha två lösningar, en lösning eller sakna lösningar beroende på värdena på p och q . Enligt den så kallde pg -formeln ges lösningarna av

$$x = -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q}.$$

Uttrycket under rottecknet kallas för ekvationens diskriminant och talar om hur många lösningar andragradsekvationen har. De olika fallen sammanfattas nedan:

$$\left(\frac{p}{2}\right)^2 - q > 0 \quad \text{ekvationen har två lösningar}$$

$$\left(\frac{p}{2}\right)^2 - q = 0 \quad \text{ekvationen har en lösning (dubbelrot)}$$

$$\left(\frac{p}{2}\right)^2 - q < 0 \quad \text{ekvationen saknar lösningar.}$$

Programmet **pq.py** låter användaren mata in värden på p och q och skriver sedan ut lösningarna till ekvationen

```
# pq.py
import numpy as np
p = float(input('Ge p '))
q = float(input('Ge q '))
# Beräkna diskriminanten
d = (p/2)**2 - q
if d > 0:
    print('Två lösningar: ', -p/2 + np.sqrt(d), -p/2 - np.sqrt(d))
elif d == 0:
    print('En lösning: ', -p/2)
else:
    print('Saknar reella lösningar')
```

Då vi provkör programmet och matar in $p = 2$ och $q = 1$ får vi

En lösning: -1.0

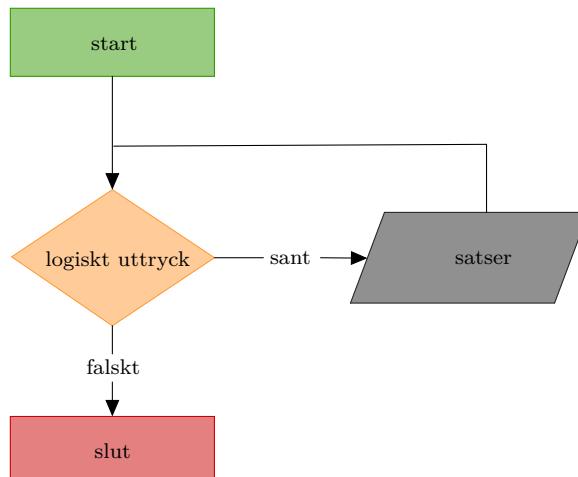
Om vi provkör med $p = -3$ och $q = 2$ blir svaret

Två lösningar: 2.0 1.0

En kontroll visar att bågge svaren är korrekta. □

7.3 While-loopar

I en while-loop repeteras satser så länge ett logiskt uttryck är sant. En while-loop illustreras i flödesschemat i figur 7.2.



Figur 7.2: I en while-loop repeteras ett antal satser så länge ett logiskt uttryck är sant.

En while-loop har formen

```
while uttryck:
    satser      # en eller flera indragna satser
    satser efter while-loopen dras inte in
```

Först beräknas värdet av det logiska uttrycket. Om detta är sant utförs `satsen` varefter det logiska uttrycket beräknas igen och så vidare tills det att det logiska uttrycket blir falskt. Python hoppar då ur while-loopen och fortsätter att exekvera satserna som kommer under loopen och är utan indrag. Om det logiska uttryckets värde är falskt redan från början utförs `satsen` inte någon gång. Ett repetitionsmoment brukar kallas en iteration, och en while-loop kan beskrivas som att man itererar så länge ett logiskt villkor är sant. Satsen som följer efter det logiska uttrycket i while-loopen måste dras in. Satsen efter while-loopen har inget indrag.

Exempel 7.5. Vi kastar pil på en $2 \text{ m} \times 2 \text{ m}$ stor tavla som innesluter in cirkelskiva med radie 1 m och centrum i origo. Vi håller på och kasta så länge (while) vi hamnar utanför 'bulls eye', dvs en cirkelskiva med radien 0.1 m. Då vi väl träffat 'bulls eye' sluta vi kasta. Vi räknar hur många kast vi behöver.

Ett kast går till på det sättet att vi slumpar ut en x - och en y -koordinat mellan -1 och 1 , markerar positionen på tavlan, och beräknar avståndet $d = \sqrt{x^2 + y^2}$ till tavlans centrum. Vi inför en variabel `hit` som signalerar om vi träffat 'bulls eye', dvs om $d < 0.1$ eller inte och en variabel `n` som räknar antalet kast vi gjort. Båda variablerna sätts till noll innan vi går in i while-loopen. Inne i loopen stegar vi upp räknaren genom `n = n + 1`. Om vi träffar 'bulls eye' sätter vi `hit` till 1. Vi har följande Python kod, som även ritar piltavlan och markerar var på tavlan kasten hamnar. Gå tillbaka till exempel 6.6 för en beskrivning av hur vi ritar cirklar.

```
# pilkast.py
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

# rita tavla
ax.plot([-1,1,1,-1,-1],[-1,-1,1,1,-1])

# rita stor och liten cirkel
x = np.linspace(-1,1,100)
y = np.sqrt(1-x**2)
ax.plot(x,y,'k')
ax.plot(x,-y,'k')
ax.plot(0.1*x,0.1*y,'r')
ax.plot(0.1*x,-0.1*y,'r')

# samma skala på x- och y-axeln
# ta bort axelmarkeringarna
ax.axis('equal')
ax.axis('off')

hit = 0      # initialisera
n    = 0      # initialisera
while hit < 1:                      # iterera så länge ingen träff
    x = -1 + 2*np.random.rand()       # slumpad x-koord. mellan -1 och 1
    y = -1 + 2*np.random.rand()       # slumpad y-koord. mellan -1 och 1
```

```

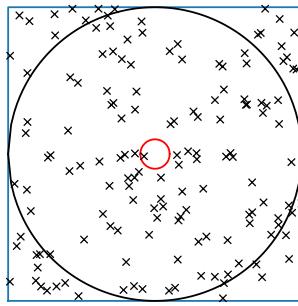
ax.plot(x,y,'kx')          # plotta var kastet hamnar
n = n + 1                  # antal kast ökar med 1
if np.sqrt(x**2 + y**2) < 0.1: # träff - ändra värdet på hit
    hit = 1
print('Antal kast ',n)      # utanför loopen, inget indrag

```

Eftersom kasten är slumpmässiga, blir antalet kast som behövs olika varje gång vi kör programmet. När jag körde programmet en gång fick jag

Antal kast 149

och träffbilden som visas i figur 7.3. □



Figur 7.3: Vi kastar pil på en tavla och håller på att kasta så länge vi inte har träffat 'bulls eye'.

Exempel 7.6. Vi ska ta reda på hur många 25-litersspannar man kan fylla i en tank som rymmer 4990 liter utan att den svämmar över (ja, man kan lösa det utan programmering också).



Figur 7.4: Vi ska ta reda på hur många 25-litersspannar man kan fylla i en tank som rymmer 4990 liter utan att den svämmar över.

Vi inför en variabel v som talar om hur mycket vatten vi har fyllt i tanken och en variabel n som talar om hur många spannar vi fyllt i. Båda variablerna sätts till noll innan vi går in i while-loopen och stegas upp genom $v = v + 25$ och $n = n + 1$ då vi fyller i en spann. Vi fyller i så länge volymen är mindre än 4990 (while-satsen). När vi går ur loopen inser vi att v har värdet 25 för mycket och n har värdet 1 för mycket och vi måste subtrahera med 25 respektive 1 vid utskriften. Utskriften sker utanför loopen och har inget indrag.

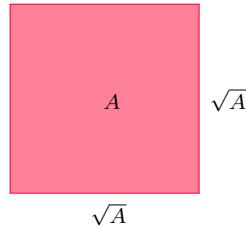
```
# volym.py
v = 0 # volymen 0 då vi börjar (initialisering)
n = 0 # antalet spannar 0 då vi börjar
while v < 4990: # iterera så länge v < 4990
    v = v + 25 # fyller i en spann, volymen ökar med 25
    n = n + 1 # antalet ifyllda spannar ökar med 1
print('Antal spannar ',n - 1) # utanför loopen, inget indrag
print('Volym vatten ',v - 25)
```

Då vi kör programmet får vi

```
Antal spannar 199
Volym vatten 4975
```

En kontroll visar att $199 \times 25 = 4975$, och det är den största volym vi kan fylla på utan att det rinner över. \square

Exempel 7.7. Vi ska beräkna kvadratrotten \sqrt{A} ur det positiva talet A . Geometriskt är detta samma som att konstruera en kvadrat vars area är A .



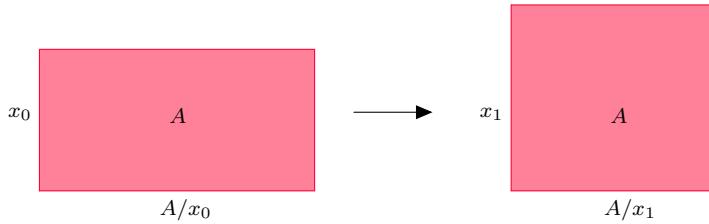
Figur 7.5: En kvadrat med arean A har sidor med längd \sqrt{A} eftersom $\sqrt{A} \cdot \sqrt{A} = A$.

Antag att vi startar med ett gissat värde x_0 på kvadratrotten till A . Detta motsvarar geometriskt, om vi ska ha arean A , en rektangel med sidolängderna x_0 och A/x_0 (vi har $x_0 \times A/x_0 = A$). För att göra rektangeln ”mer kvadratisk” kan vi ersätta x_0 med ett tal x_1 som är medelvärdet av x_0 och A/x_0 dvs.

$$x_1 = \frac{1}{2} \left(x_0 + \frac{A}{x_0} \right) \quad \text{sidolängden } x_1 \text{ gör rektangeln ”mer kvadratisk” än sidolängden } x_0$$

Vi använder sedan x_1 som nytt startvärde, dvs. vi ger x_0 värdet x_1 och beräknar på nytt ett bättre värde x_1 och så vidare (se figur 7.6). Iterationerna avbryts då avståndet $|x_1 - x_0| < tol$, där tol är en given positiv tolerans eller noggrannhet.¹ Den ovan beskrivna metoden kallas Herons metod efter Heron från Alexandria som levde 10 - 70 efter e.Kr. Dock, metoden var känd redan av Babylonierna ca 1700 f.Kr.

¹ Avstånd är alltid positiva. Avståndet mellan talet x_0 och x_1 skrivs $|x_0 - x_1|$ och beräknas genom `abs(x0 - x1)`, se exempel 2.12.



Figur 7.6: Rektangeln med sidolängder x_0 och A/x_0 görs ”mera kvadratisk” genom att ersätta x_0 med ett tal x_1 som är medelvärdet av x_0 och A/x_0 .

Programmet **kvadratrot.py** genererar en serie med allt bättre approximationer till \sqrt{A} . Iterationerna avbryts då avståndet dx mellan x_1 och x_0 blir mindre än en given tolerans tol . I programmet räknas även antalet iterationer med hjälp av variabeln n som stegas upp genom $n = n + 1$ i varje iteration.

```
# kvadratrot.py
A    = float(input(' Ge ett värde på A '))
x0   = float(input(' Ge ett startvärde för roten '))
tol = float(input(' Ge toleransen '))
n = 0                      # initialisering
dx = tol + 1                # se till att vi går in i loopen
while dx > tol:            # iterera så länge dx > tol
    x1 = 0.5*(x0 + A/x0)    # bättre värde
    dx = abs(x1 - x0)       # avståndet x0 och x1
    x0 = x1                 # x1 nytt startvärde, dvs. x0 = x1
    n = n + 1                # stega upp iterationsräknaren
print('Roten är ',x0)        # utskrift
print('Iterationer ',n)      # utskrift
```

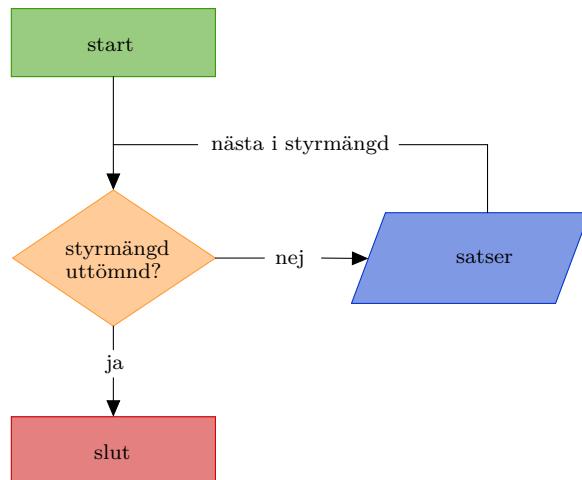
Då vi testkör programmet med $A = 2$, $x_0 = 1$ och $tol = 10^{-14}$ får vi

```
Roten är      1.414213562373095
Iterationer  6
```

En kontroll visar att det beräknade värdet på roten är korrekt i alla femton decimaler. □

7.4 For-loopar

I en for-loop repeteras satser så länge en styrvariabel löper i en styrmängd. En for-loop illustreras i flödesschemat i figur 7.7



Figur 7.7: I en for-lopp repeteras satser så länge en styrvariabel löper i en styrmängd. Då styrmängden är uttömd fortsätter exekveringen med satserna som kommer under for-loopen.

Den generella formen av en for-sats är

```

for styrvariabel in styrmängd:
    satser      # en eller flera indragna satser
    satser efter for-loopen dras inte in
  
```

Styrvariabeln sätts till det första värdet i styrmängden och **satser** utförs. Styrvariabeln stegas upp och sätts till det andra värdet i styrmängden varefter **satser** utförs och så vidare tills det styrvariabeln har löpt igenom hela styrmängden. Python hoppar då ur for-loopen och fortsätter att exekvera satserna som står under loopen och som är utan indrag. Om styrmängden är tom utförs **satser** inte någon gång. I Python har vi stor frihet vad gäller styrmängden och vi ger några exempel.

Styrmängd med hjälp av lista

Den enklaste styrmängden är en lista med tal. Ett exempel är

```

for i in [1,2,3,6]:
    satser
    satser efter for-loopen dras inte in
  
```

där *i* successivt sätts till 1, 2, 3 och 6.

Styrmängd med hjälp av range

Ofta används funktionen `range(a,b,h)` för att definiera styrmängden. Här är *a*, *b* och *h* heltal som anger start, slut och steg för styrmängden. Sista talet i styrmängden är alltid mindre än *b*. Om vi utelämnar *h* blir steglängden automatiskt lika med ett. Om vi utelämnar *a* och *h* startar styrmängden i 0 och går i steg om 1 till *b* – 1. Ett exempel är

```
for i in range(1,8,2):
    satser
    satser efter for-loopen dras inte in
```

där i successivt sätts till 1, 3, 5 och 7.

Styrmängd med hjälp av np.arange

En annan variant är att använda funktionen `np.arange(a,b,h)`. Skillnaden mot `range` är att a , b och h nu inte behöver vara heltal. Ett exempel är

```
for x in np.arange(0,2.1,0.5):
    satser
    satser efter for-loopen dras inte in
```

där i successivt sätts till 0, 0.5, 1.5 och 2.0.

Exempel 7.8. Vi kastar pil på en $2 \text{ m} \times 2 \text{ m}$ stor tavla som innesluter in cirkelskiva med radie 1 m och centrum i origo. Vi kastar 200 gånger och räknar hur många gånger vi träffar 'bulls eye', dvs. en cirkelskiva med radien 0.1 m.

Ett kast går till på det sättet att vi slumpar ut en x - och en y -koordinat mellan -1 och 1 , markerar positionen på tavlan, och beräknar avståndet $d = \sqrt{x^2 + y^2}$ till tavlans centrum. Vi inför en variabel n som räknar hur många gånger vi träffat 'bulls eye'. Variabeln sätts till noll innan vi går in i for-loopen. Inne i loopen stegar vi upp räknaren genom $n = n + 1$ varje gång vi träffar 'bulls eye', dvs. om $d < 0.1$. Koden är som följer

```
# pilkast.py
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

# rita tavla
ax.plot([-1,1,1,-1,-1],[-1,-1,1,1,-1])

# rita stor och liten cirkel
x = np.linspace(-1,1,100)
y = np.sqrt(1-x**2)
ax.plot(x,y,'k')
ax.plot(x,-y,'k')
ax.plot(0.1*x,0.1*y,'r')
ax.plot(0.1*x,-0.1*y,'r')

# samma skala på x- och y-axeln
# ta bort axelmarkeringarna
ax.axis('equal')
ax.axis('off')

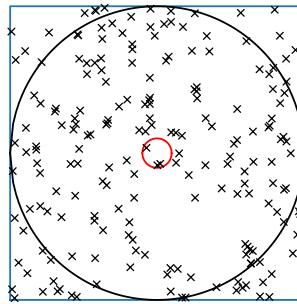
n = 0                                # initialisera
for i in range(200):                   # loopa 200 gånger, i från 0 till 199
    x = -1 + 2*np.random.rand()        # slumpad x-koord. mellan -1 och 1
    y = -1 + 2*np.random.rand()        # slumpad y-koord. mellan -1 och 1
    ax.plot(x,y,'kx')                # plotta var kastet hamnar
    if np.sqrt(x**2 + y**2) < 0.1:   # träff - ökar n med 1
```

```
n = n + 1
print('Antal träffar ',n)           # utanför loopen, inget indrag
```

Eftersom kasten är slumpmässiga, blir antalet gånger vi träffar 'bulls eye' olika varje gång. När jag körde programmet en gång fick jag

Antal träffar 3

och träffbilden som visas i figur 7.8. □



Figur 7.8: Vi kastar pil 200 gånger på en tavla och räknar hur många gånger vi träffar 'bulls eye'.

En mycket vanlig uppgift är att loopa över elementen i en vektor v . Styrmängden anges då av `range(len(v))`, där `len(v)` är längden (antalet element) i vektorn.

Exempel 7.9. I filen **T.txt** finns vektorn T med dagliga temperaturvärden under 23 år för en mätstation i Skåne. Programmet **temperatur.py** läser temperaturvärdena i filen och bestämmer medeltemperatur under de 23 åren.

```
# temperatur.py
import numpy as np
T = np.loadtxt('T.txt') # läs in och lagra i T
Tsum = 0               # initialisera summan
for i in range(len(T)):
    Tsum = Tsum + T[i] # addera term till summan
Tmedel = Tsum/len(T)   # beräkna medeltemperaturen
print('Medeltemperatur: ', round(Tmedel,2))
```

För att beräkna medeltemperaturen inför vi T_{sum} , vilken vi sätter till 0 utanför loopen. Sedan adderar vi successivt dagliga temperaturer. Efter loopen delar vi med antalet element för att få medelvärdet. Då vi kör programmet får vi

Medeltemperatur: 7.59

Minsta och största värde samt medelvärde kan också fås genom att använda de inbyggda funktionerna i avsnitt 5.20. □

7.5 Nästlade loopar

For- och while-loopar kan vara nästlade på olika sätt. I konstruktionen nedan har vi två nästlade for-loopar

```

for styrvariabel1 in styrmängd1:
    for styrvariabel2 in styrmängd2:
        satser
    satser efter for-looparna dras inte in

```

`styrvariabel1` sätts till det första värdet i `styrmängd1`. Vi går sedan in i den andra for-loopen. `styrvariabel2` sätts till det första värdet i `styrmängd2` och `satsen` utförs. `styrvariabel 2` stegas upp och sätts till det andra värdet i `styrmängd2` varefter `satsen` utförs och så vidare tills `styrvariabel2` har löpt igenom hela `styrmängd2`. Den inre for-loopen har nu avslutats och `styrvariabel1` stegas upp och sätts till det andra värdet i `styrmängd1`. Vi går på nytt in i den andra for-loopen och utför den osv.

Exempel 7.10. Betrakta matrisen

$$A = \begin{pmatrix} 1 & 1/2 & 1/3 & 1/4 & 1/5 & 1/6 \\ 1/2 & 1/3 & 1/4 & 1/5 & 1/6 & 1/7 \\ 1/3 & 1/4 & 1/5 & 1/6 & 1/7 & 1/8 \end{pmatrix}.$$

Programmet **matris.py** genererar matrisen A med hjälp av två nästlade for-loopar. Programmet beräknar också summan av alla matriselement. Notera hur vi måste börja med att fördefiniera matrisen A innan vi tilldelar elementen värdet.

```

# matris.py
import numpy as np
A = np.zeros((3,6))          # fördimensionering
s = 0                         # initialisering
for i in range(3):           # loop över rader
    for j in range(6):         # loop över kolonner
        A[i,j] = 1/(i+j+1)    # generera element
        s = s + A[i,j]          # addera element till summan
print('Summan av elementen är',s)

```

Summan kan också beräknas med det inbyggda kommandot `sum`. □

Exempel 7.11. Vi har en 3×2 -matris

$$A = \begin{pmatrix} 1 & 5 \\ 4 & 0 \\ 2 & 8 \end{pmatrix}.$$

Programmet **radkolum.py** beräknar och skriver ut radsummorna och kolonnsummorna.

```

import numpy as np
# radkolum.py
A = np.array([[1,5],
              [4,0],
              [2,8]])      # 3 x 2 matris
print('Radsummor')
for i in range(3):      # loop över rader
    s = 0                # initialisera
    for j in range(2):    # loop över kolonner
        s = s + A[i,j]    # addera element till summan
    print('Summan av elementen i rad',i,'är',s)

print('Kolonnumsummor')

```

```

for j in range(2):          # loop över kolonner
    s = 0                  # initialisera
    for i in range(3):      # loop över rader
        s = s + A[i,j]      # addera element till summan
    print('Summan av elementen i kolonn',j,'är',s)

```

Då vi kör programmet får vi

```

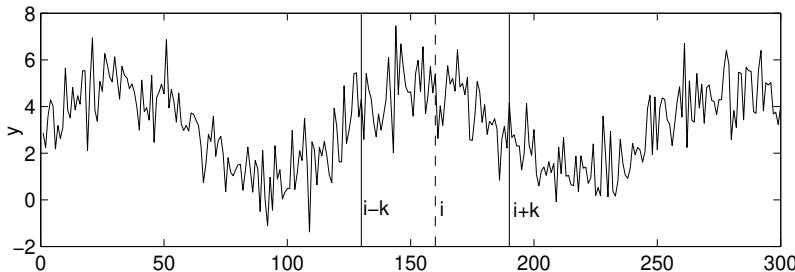
Radsummor
Summan av elementen i rad 0 är 6
Summan av elementen i rad 1 är 4
Summan av elementen i rad 2 är 10
Kolonnumsummor
Summan av elementen i kolonn 0 är 7
Summan av elementen i kolonn 1 är 13

```

För beräkning med hjälp av inbyggda kommandon, se exempel 5.24. \square

7.6 Tillämpning: signalbehandling

Signaler eller tidsserier innehåller ofta slumpmässiga oregelbundenheter. Dessa kan utjämnsas genom att bilda ett glidande medelvärde (se figur 7.9).



Figur 7.9: Tidsserie med slumpmässiga och högfrekventa oregelbundenheter. Vid glidande medelvärde ersätter man varje punkt $y(i)$, $i = 0, 1, \dots, n - 1$ med medelvärdet $y_m(i)$ av punkter i ett fönster från $i - k$ till $i + k$.

Varje punkt $y(i)$, $i = 0, 1, \dots, n - 1$ i den ursprungliga signalen ersätts av medelvärdet $y_m(i)$ av grannvärdena i ett fönster från $i - k$ till $i + k$

$$y_m(i) = \frac{1}{2k+1} \times \underbrace{(y(i-k) + y(i-k+1) + \dots + y(i+k-1) + y(i+k))}_{2k+1 \text{ värden}}.$$

Då i ligger alldeles i början eller slutet av tidsserien finns inte tillräckligt många punkter till vänster respektive till höger och man måste därför krympa fönstret i vilket man medelvärdesbildar.

Som ett exempel ska vi titta på temperaturen T mellan 1981 och 1983 vid en mätstation i det inre av Skåne. Temperaturdata ligger lagrade i filen **T8183.txt** som laddas ner från Canvas. Programmet **glidandemedel.py** läser data och plottar den ursprungliga tidsserien tillsammans med glidande medelvärde. Notera hur man måste kontrollera att medelvärdesbildandet hela tiden sker i ett fönster som inte går utanför den ursprungliga tidsserien.

```

# glidandemedel.py
import numpy as np

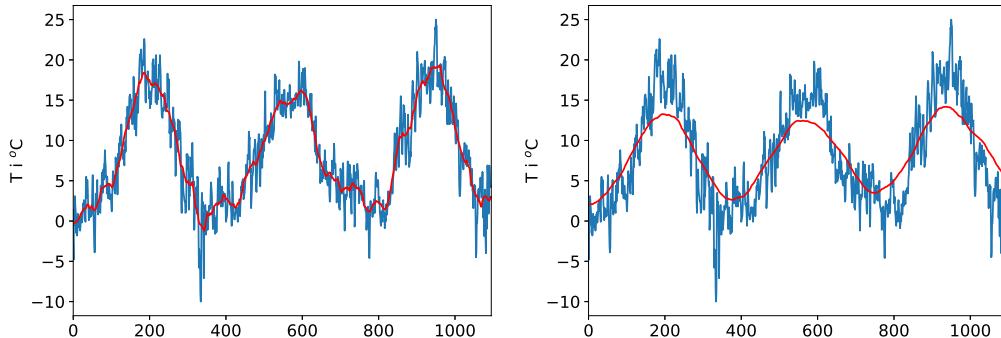
```

```

import matplotlib.pyplot as plt
T = np.loadtxt('T8183.txt')
Tm = np.zeros(len(T))                      # fördimensionering
fig, ax = plt.subplots()
ax.plot(T)
k = int(input('Ge värde på k '))
n = len(T)
for i in range(n):                         # loop över element
    n1 = np.max([0,i-k])                   # n1 aldrig mindre än 0
    n2 = np.min([n,i+k+1])                 # n2 aldrig större än n
    Tm[i] = np.mean(T[n1:n2])              # medelvärde i fönster
ax.plot(Tm,'r')
ax.set_xlim([0,n])
ax.set_ylabel(r'T i $^{\circ}\text{C}$', fontsize=14)
ax.tick_params(labelsize=14)

```

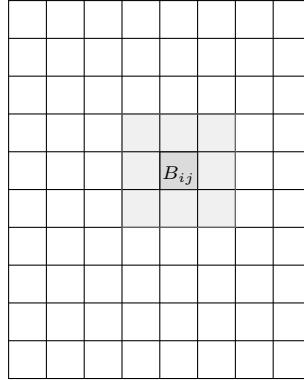
Då vi kör programmet med $k = 15$ och $k = 91$ får vi figur 7.10. Ett fönster för en månad $2 \times 15 + 1 = 31$ ger en utjämnad kurva som väl följer den underliggande signalen. Ett fönster omfattande ett halvår $2 \times 91 + 1 = 183$ ger en mycket jämn kurva. Problemet är att tidsstrukturen blir utsmetad.



Figur 7.10: Temperaturdata tillsammans med en månads respektive ett halvårs glidande medelvärde.

7.7 Tillämpning: bildbehandling

En digitaliserad svartvit bild kan representeras av en $m \times n$ -matris B , där matriselementen anger olika nyanser av grått (svärtningen). Snabba variationer i svärtningen, till exempel vid kanter eller linjer, kan utjämnas genom att bilda medelvärden. Svärtningen $B(i, j)$ i en pixel i, j i den ursprungliga bilden ersätts då av medelvärdet $B_m(i, j)$ av svärtningen i omgivande punkter. Som ett exempel tar vi en omgivning i form av en 3×3 -matris, vilket illustreras i figur 7.11.



Figur 7.11: En bild utjämns genom att bilda medelvärden. Svärtningen $B(i, j)$ i en pixel i, j i den ursprungliga bilden ersätts då av medelvärdet $B_m(i, j)$ av svärtningen i omgivande punkter.

Då (i, j) ligger i närheten av bildkanten finns inte tillräckligt många punkter för att medelvärdesbilda. Dessa punkter brukar utelämnas så att den processade bilden blir lite mindre än originalbilden.

Genom att subtrahera den utjämna bilden från den ursprungliga bilden kommer kanter och linjer att framträda i skillnadsbilden. Efter att kanter och linjer har framhävts på detta sätt kan skillnadsbilden adderas till den ursprungliga bilden för att öka kontrasten.

Som ett exempel ska vi titta på en bild som ligger lagrad i **parism.jpg** (laddas ner från Canvas). Programmet **kant.py** genererar en utjämnad bild som subtraheras från ursprungsbildens för att ge en skillnadsbild. Skillnadsbilden adderas sedan till ursprungsbildens.

```
# kant.py
import numpy as np
import matplotlib.pyplot as plt
B = plt.imread('PARISM.JPG')           # läs in
fig, ax = plt.subplots()                # plotta original
ax.imshow(B,cmap='gray')
ax.axis('equal')
ax.axis('off')
(m,n) = B.shape                       # rader och kolonner

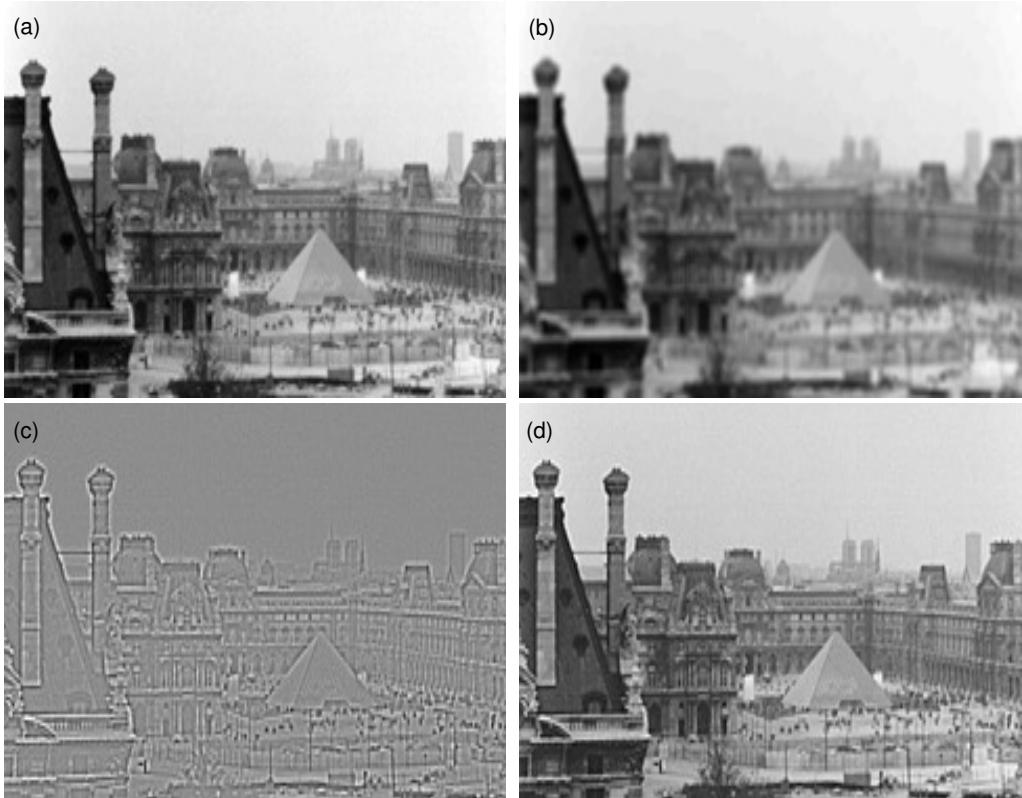
Bm = np.zeros((m,n))                  # fördimensionera
for i in range(1,m-1):                # loop över rader
    for j in range(1,n-1):              # loop över kolonner
        Bm[i,j] = np.mean(B[i-1:i+2,j-1:j+2]) # medelvärde av grannar
fig, ax = plt.subplots()
ax.imshow(Bm[1:m-1,1:n-1],cmap='gray') # plotta
ax.axis('equal')
ax.axis('off')

D = B - Bm                           # skillnadsbild
fig, ax = plt.subplots()
ax.imshow(D[1:m-1,1:n-1],cmap='gray')
ax.axis('equal')
ax.axis('off')

E = B + D                            # kantförstärkt bild
fig, ax = plt.subplots()
```

```
ax.imshow(E[1:m-1,1:n-1],cmap='gray')
ax.axis('equal')
ax.axis('off')
```

Då vi kör programmet får vi de fyra bilderna i figur 7.12.



Figur 7.12: (a) originalbild, (b) utjämnad bild, (c) skillnadsbild, (d) skillnadsbilden adderad till ursprungsbilden.

7.8 Tillämpning: bildbehandling – ändra färger

Där finns gratis verktyg på nätet som man kan använda för att byta ut färger i färgbilder, se till exempel <https://onlinepngtools.com/change-png-color>. Vi skall göra ett eget verktyg!

Färgen på en pixel i en färgbild anges av tre tal (R, G, B) . Talen, som normalt ligger mellan 0 och 255, anger andelen rött (R), andelen grönt (G) och andelen blått (B) som bygger upp färgen. Låt (R, G, B) vara den färg vi är intresserad av och låt $(\bar{R}, \bar{G}, \bar{B})$ definiera en annan färg. Vi kan nu beräkna ett relativt 'färgavstånd' som

$$\sqrt{\frac{(R - \bar{R})^2 + (G - \bar{G})^2 + (B - \bar{B})^2}{R^2 + G^2 + B^2}}.$$

Om färgavståndet är litet, ligger färgerna nära varandra. Om färgavståndet är stort, ligger färgerna lång ifrån varandra. Beträkta färbilden **havsvorn_PO.jpg** till vänster i figur 7.13. Vi kan läsa in bilden och visa den i Python med hjälp av kommandona nedan.

```

import numpy as np
import matplotlib.pyplot as plt

fig,ax = plt.subplots()
A = plt.imread('havsvorn_PO.jpg') # läs bild och spara i A
ax.imshow(A) # bild
ax.axis('equal')
ax.axis('off')

```

Om vi nu 'hoovrar' över bilden med muspekaren får vi uppe till höger i verktygslisten både rad och kolonn för den pixel vi pekar på och dessutom RGB-värdet. Då vi 'hoovrar' över himlen ser vi att den blå färgen ges av $RGB = (1, 122, 195)$.

Följande program läser in bilden **havsvorn_PO.jpg** och lagrar den i en matris A som vi omvandlar till flyttal. Vi anger den blå färgen $RGB = (1, 122, 195)$ och en ny färg, som vi tar till rosa $RGB = (255, 192, 203)$. Vi loopar över rader och kolonner i bilden och beräknar färgavståndet för den aktuella pixeln till den blå färgen. Om avståndet är litet byter vi ut den aktuella färgen till rosa. Sedan plottar vi bilden.

```

import numpy as np
import matplotlib.pyplot as plt

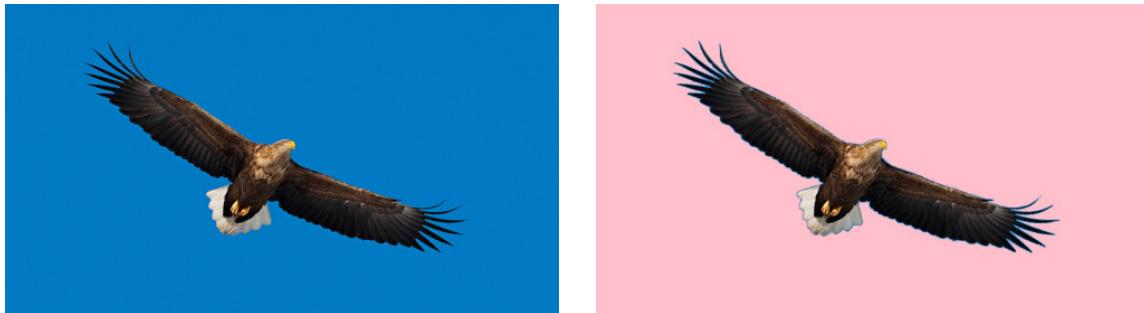
fig,ax = plt.subplots()
A = plt.imread('havsvorn_PO.jpg') # läs bild och spara i A
A = A.astype(dtype='float') # obs, omvandla från uint8 till flyttal

RGB = [1,122,195] # blå färg som vi vill byta ut
RGBNY = [255,192,203] # ny färg, ljus rosa

(m,n,o) = A.shape # dimensionerna på bilden, m antal rader
# n antalet kolonner, o djupet
for i in range(m): # loopa över rader och kolonner
    for j in range(n):
        a = (A[i,j,0]-RGB[0])**2 + (A[i,j,1]-RGB[1])**2 + \
            (A[i,j,2]-RGB[2])**2
        b = RGB[0]**2 + RGB[1]**2 + RGB[2]**2
        dist = np.sqrt(a/b) # relativt färgavstånd
        if dist < 0.1: # om färgavstånd litet, byt ut färgen
            A[i,j,:] = RGBNY # samma som A[i,j,0] = RGBNY[0],
            # A[i,j,1] = RGBNY[1] och A[i,j,2] = RGBNY[2]
A = A.astype(dtype='int') # måste omvandla till heltal innan vi visar
ax.imshow(A) # visa modifierad bild
ax.axis('equal')
ax.axis('off')

```

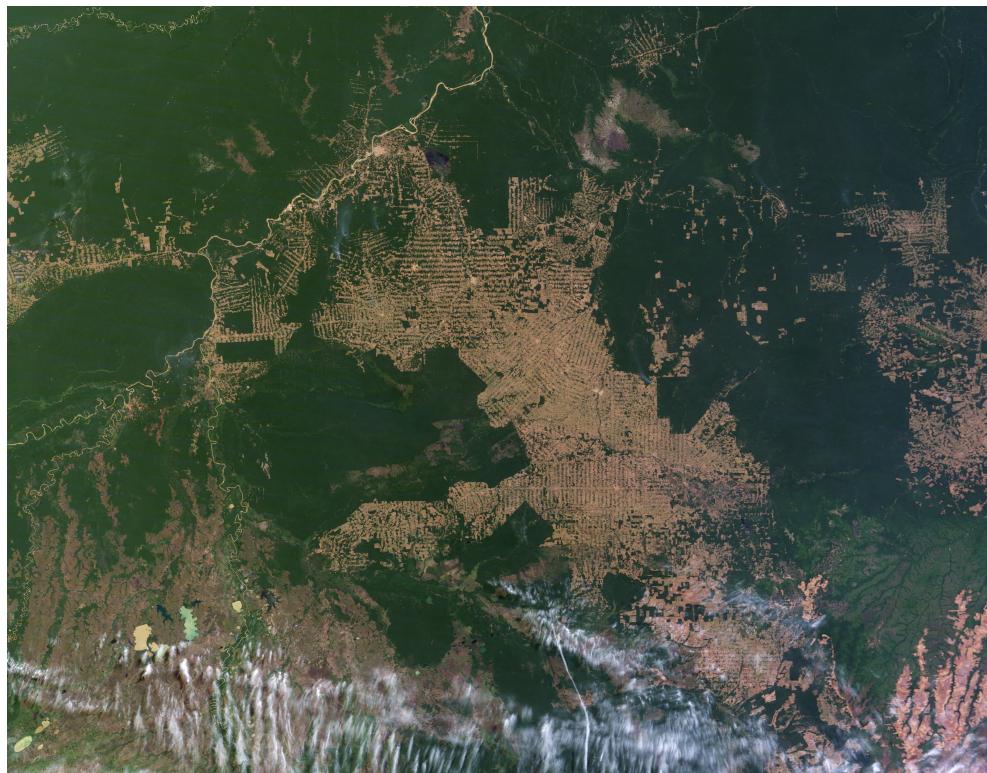
Då vi kör programmet får vi bilden till höger i figur 7.13. Vi ser att färgen på pixlar med ett litet färgavstånd till den blå färgen har bytts ut till rosa. Test gärna och läs in någon egen bild och byt ut en eller flera färger.



Figur 7.13: Till vänster originalbild av Patrik Olofsson/N. Till höger bild där vi ersatt den blå färgen med rosa

7.9 Tillämpning: bildbehandling – avskogning av Amazonas

Avskogning och ändrad markanvändning är ett hot mot biodiversitet och ekosystemens stabilitet, men även mot klimatet och globala hydrologiska cykler. NASA:s jordobservatorium <https://earthobservatory.nasa.gov/world-of-change/Deforestation> visar filmer över hur skogar, glaciärer, istäcke och andra naturtyper ändras i en allt snabbare takt. Vi skall här titta på klassifikation, en viktig uppgift inom databehandling, där vi skall inordna pixlar i en bild över Amazonas, se figur 7.14, i tre klasser: avskogat område, ursprunglig regnskog eller moln. Bilden med namn `amazon_deforestation_20110805.jpg` finns att ladda ner från Canvas.



Figur 7.14: Bild, tagen med NASA:s Terra satellit, som visar avskogning i Amazonas. Vi skall inordna varje pixel i en av de tre breda klasserna regnskog, avverkning och moln.

Tabell 7.1: Typiska RGB-värden för de tre breda klasserna regnskog, avverkning och moln.

klass	<i>R</i>	<i>G</i>	<i>B</i>
regnskog	35	56	37
avverkning	157	126	98
moln	216	221	243

I tabell 7.1 redovisar vi typiska RGB-värden för de tre breda klasserna regnskog, avverkning och moln. RGB-värdena har fåtts genom att 'hoovra' över typiska regnskogs-, avverknings- och molnområden i bilden.

Klassificeringen går till på det sätt att vi loopar över alla pixlar i bilden. För var och en av pixlarna beräknar vi det relativata 'färgavståndet'

$$\sqrt{\frac{(R - \bar{R})^2 + (G - \bar{G})^2 + (B - \bar{B})^2}{R^2 + G^2 + B^2}}.$$

till klasserna regnskog, avverkning och moln. Om färgavståndet till regnskog är minst, klassas pixeln som regnskog, om avståndet till avverkning är minst, klassas pixeln som avverkning osv. För att tydliggöra klassificeringen ersätter vi färgen i en pixel som klassats som regnskog med klargrön, färgen i en pixel som klassats som avverkning med mörkbrun och färgen i en pixel som klassats som moln med helvitt. Då vi är intresserade av hur stor andel av området som har avverkats räknar vi hur många pixlar som faller i de olika klasserna. Antalet redovisas sedan som procent av det totala antalet pixlar. Programmet som genomför klassificeringen visas nedan.

```

a = (A[i,j,0]-RGB[k,0])**2 + (A[i,j,1]-RGB[k,1])**2 + \
     (A[i,j,2]-RGB[k,2])**2
b = RGB[k,0]**2 + RGB[k,1]**2 + RGB[k,2]**2
dist[k] = np.sqrt(a/b)      # färgavstånd

klass = np.argmin(dist)    # bestäm vilken klass som har minsta
                           # färgavstånd till pixeln
antal[klass] = antal[klass] + 1 # öka antalet i denna klass med 1
A[i,:,:] = RGBNY[klass,:,:] # omdefiniera färgen på pixeln i enlighet
                           # med den bestämda klassen
A = A.astype(dtype='int')    # måste omvandla till heltal innan vi visar
ax.imshow(A)                 # visa modifierad bild
ax.axis('equal')
ax.axis('off')

print('regnskog   ',round(100*antal[0]/(m*n),1),'procent')
print('avverkning ',round(100*antal[1]/(m*n),1),'procent')
print('moln       ',round(100*antal[2]/(m*n),1),'procent')

```

Då vi kör programmet får vi

```

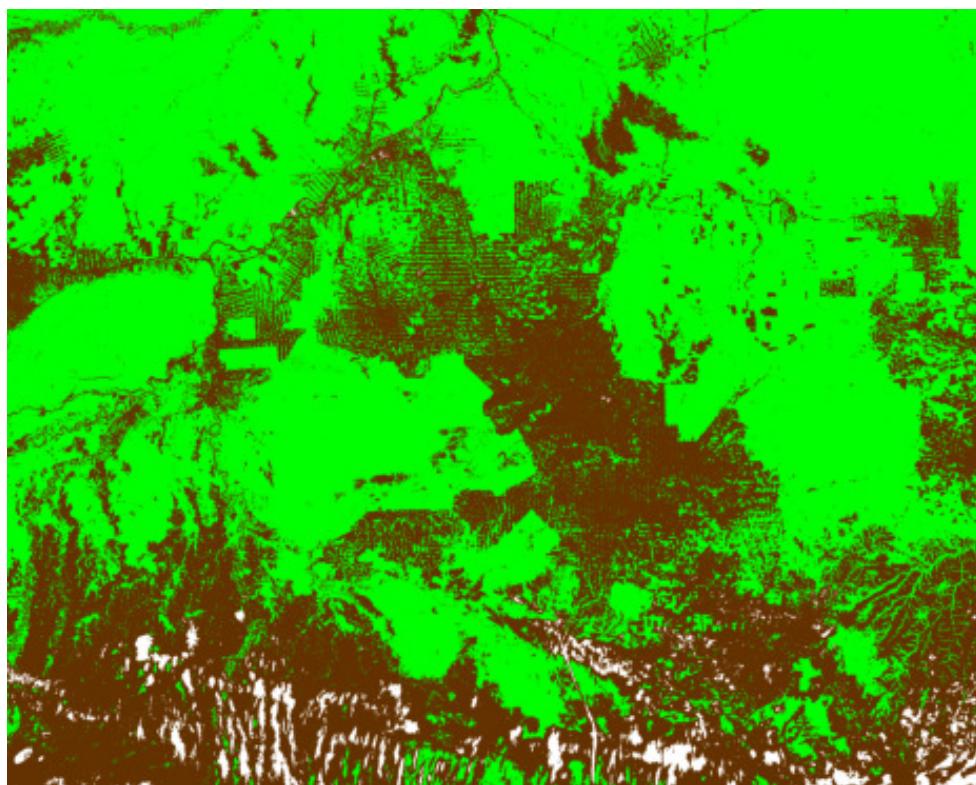
rad  0 av 3000
rad  1 av 3000
...
rad  2998 av 3000
rad  2999 av 3000
regnskog   59.4 procent
avverkning 38.3 procent
moln       2.2 procent

```

Bilden, där de klassificerade pixlarna har fått nya färger, visas i figur 7.15. På det stora hela fungerar klassificeringen väl. En noggrann inspektion av den ursprungliga bilden avslöjar att det finns sekundärskog, dvs. skog med mycket mindre artrikedom som har återplanterats på avverkade områden. Sekundärskogen har RGB-värden som ligger 'mellan' de för regnskog och avverkning och har, som det verkar, till stor del klassats som avverkning i vårt schema. Det är möjligt att öka antalet klasser, men det visar på det generella problemet att man alltid måste göra en avvägning av hur många klasser man skall ta inkludera.

Vid markanvändningsmonitorering klassificerar man regelbundet markanvändningen på lokal, regional och global skala och tittar på förändringar år från år https://en.wikipedia.org/wiki/Land_change_science. Förflyttningshastigheten är i många fall skrämmande.

Som du märker tar det ganska lång tid att loopa igenom bilden och göra en klassning. Programmet skulle kunna snabbas upp en faktor tvåhundra med så kallad vektorisering. Detta ligger dock utanför kursens ram.



Figur 7.15: Bild som visar klassificering av pixlar i ett område i Amazonas. Klargrön är ursprunglig regnskog, mörkbrun är avskogat område och vitt är moln.

Kapitel 8

Programstruktur

För att få överblick och struktur delar man ofta upp uppgifterna i ett program i mindre delar, vilka utförs av funktioner. Funktioner, som anropas i programmet med ett antal invariabler, utför operationer på variablene och returnerar ett antal utvariabler. Funktioner kan vara inbyggda och samlade i moduler, som i `numpy` eller `matplotlib`, men de kan också vara egenutvecklade. Vi ska här titta närmare på hur man skriver egna funktioner och anropar dem från ett program.

8.1 Program, funktioner och moduler

Ett program består av ett antal kommandon samlade i en fil. Programmet får samma namn som namnet på filen. Funktioner har väldefinierade namn, tar ett antal invariabler, utför operationer på variablene och returnerar ett antal utvariabler. Funktioner kan placeras i samma fil som programmet, och då placerar man dem normalt överst. I programmet anropas sedan funktionerna med funktionsnamnen följt av variablene inom parentes, se figur 8.1.

```
# program.py
def. funktion 1
def. funktion 2
:
def. funktion n
# programstart
kommandon och
funktionsanrop
```

Figur 8.1: Funktioner definierade överst i programmet. I själva huvudprogrammet anropas funktionerna med funktionsnamnen.

8.2 Funktioner

En funktion tar ett antal invariabler (inparametrar) x_0, x_1, \dots, x_m , utför operationer på variablene och returnerar en eller flera utvariabler (utparametrar) y_0, y_1, \dots, y_n . En funktion har den allmänna formen

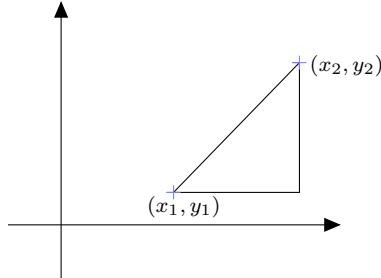
```
def funktionsnamn(x0,x1,...,xm):
    """
    dokumentation
    """
    satser
    return y0,y1,...,yn
```

En funktion behöver inte returnera några utvariabler, och då utelämnar man bara sista raden.

Det reserverade ordet `def` markerar starten av funktionen. Sedan följer funktionsnamnet och listan av invariabler. Första raden avslutas med ett kolon. Dokumentation och satser som följer ska, precis som vid if-, while- och for-satser, dras in fyra steg. Även om det inte är obligatoriskt brukar man börja en funktion med en så kallad dokumentationssträng (doc-string) inom trippla citationsstecken. Dokumentationssträngen beskriver vad funktionen gör, vilka invariabler den tar och vilka utvariabler den producerar. Då man på kommandoraden skriver `funktionsnamn?` visas dokumentationssträngen. Efter dokumentationssträngen kommer satserna som utförs av funktionen. Sist kommer `return` följt av funktionens utvariabler. Satser som följer funktionen dras inte in.

Förutom invariablerna och utvariablerna använder man i funktionen ofta ytterligare variabler för att lagra mellanresultat, använda som loopvariabler och så vidare. Dessa ytterligare variabler kallas lokala variabler.

Exempel 8.1. Vi har två punkter (x_1, y_1) och (x_2, y_2) i planet (se figur 8.2).



Figur 8.2: Avståndet mellan punkterna (x_1, y_1) och (x_2, y_2) ges av Pythagoras sats.

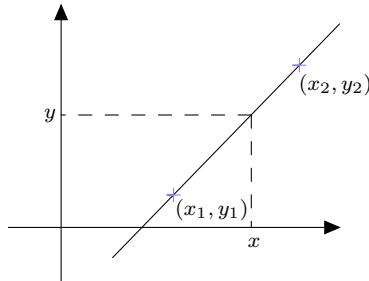
Avståndet mellan punkterna beräknas med hjälp av Pythagoras sats

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

Funktionen som givet de fyra koordinaterna x_1, y_1, x_2, y_2 returnerar avståndet mellan punkterna ges av

```
def distance(x1,y1,x2,y2):
    """
    tar koordinaterna för två punkter (x1,y1) och (x2,y2)
    och beräknar avståndet m.h.a. Pythagoras sats
    """
    d = np.sqrt((x1-x2)**2 + (y1-y2)**2)
    return d
```

Exempel 8.2. Vi har två punkter (x_1, y_1) , (x_2, y_2) enligt figur 8.3.



Figur 8.3: Linjär interpolation mellan punkterna (x_1, y_1) och (x_2, y_2) .

Vid interpolation bestämmer man ekvationen

$$y = kx + m$$

för den räta linjen genom punkterna. Denna ekvation används sedan för att beräkna y -värdet för ett givet x -värdet.

Riktningskoefficienten för linjen genom (x_1, y_1) och (x_2, y_2) ges av

$$k = \frac{y_2 - y_1}{x_2 - x_1}.$$

Insättning av x_1 ger

$$y_1 = kx_1 + m.$$

Värdet på m fås nu som

$$m = y_1 - kx_1.$$

Vi kan nu skriva en funktion som givet de fyra koordinaterna x_1, y_1, x_2, y_2 och ett x -värde returnerar motsvarande y -värde. Funktionen blir

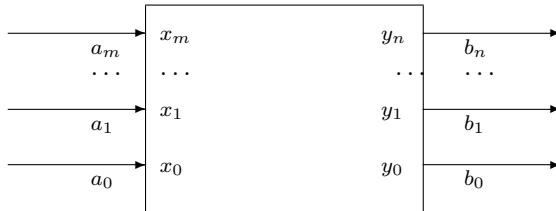
```
def interp(x1,y1,x2,y2,x):
    """
    linjär interpolation mellan två punkter (x1,y1) and (x2,y2).
    givet ett x beräknar vi motsvarande y
    """
    k = (y2-y1)/(x2-x1)    # riktningskoefficient
    m = y1 - k*x1          # m-värde
    y = k*x + m            # sätt in x i linjens ekvation
    return y
```

8.3 Anrop av funktioner

Funktioner placeras normalt högst upp i filen som innehåller ett program. Efter alla funktioner kommer själva programmet, se figur 8.1. I programmet anropas funktioner genom

```
b0,b1,...,bn = funktionsnamn(a0,a1,...,am)
```

Vid anropet tilldelas funktionens invariabler x_0, x_1, \dots, x_m värdena av de anropande variablene a_0, a_1, \dots, a_m . Tilldelningarna fungerar på samma sätt som vanliga tilldelningar.



Figur 8.4: Vid anropet tilldelas funktionens invariabler x_0, x_1, \dots, x_m värdena av de anropande variablene a_0, a_1, \dots, a_m . Efter det att satserna i funktionen har utförts tilldelas b_0, b_1, \dots, b_n värdena av y_0, y_1, \dots, y_n och exekveringen försätter i det anropande programmet.

Lokala variabler tilldelas normalt värden inne i funktionen och påverkar då inte variabler med samma namn i det anropande programmet. En lokal variabel som inte har blivit tilldelad ett värde inne i funktionen innan variabeln används, tilldelas värdet av motsvarande variabel i det anropande programmet. Användning av icke-definierade lokala variabler betraktas som dålig programmeringsstil, men kan vara användbart och motiverat i vissa sammanhang.

Vid ”återhoppet” från funktionen till det anropande programmet tilldelas variablene b_0, b_1, \dots, b_n värdena av funktionens utvariabler y_0, y_1, \dots, y_n varefter alla funktionens variabler tas bort.

Exempel 8.3. Vi har två punkter med koordinater $(1, 2)$ och $(3, -8)$. Avståndet mellan punkterna beräknas med följande program som kallar på funktionen `distance`

```

# distance_points
import numpy as np

def distance(x1,y1,x2,y2):
    """
        tar koordinaterna för två punkter (x1,y1) och (x2,y2)
        och beräknar avståndet m.h.a. Pythagoras sats
    """
    d = np.sqrt((x1-x2)**2 + (y1-y2)**2)
    return d

# här börjar det egentliga programmet
x1 = 1
y1 = 2
x2 = 3
y2 = -8

d = distance(x1,y1,x2,y2) # anrop av funktionen
print('Avståndet är ',round(d,3))
  
```

Då vi kör programmet får vi

Avståndet är 10.198

□

Exempel 8.4. Vi har följande omräkningstabell för hårdhetsskalor från CoroKey, Applikationsguide från Sandvik Coromat, 4:e utgåvan 1997.

OMRÄKNINGSTABELL FÖR HÄRDHETSSKALOR

Inom industrin används många olika system för mätning av materialhärdhet.
I tabellen nedan jämförs de tre vanligaste systemen.

CoroKey skärdaterekommendationer anges i Brinell (HB).

HB 180 för stål (CMC-kod 02.1)

HB 180 för rostfritt stål (CMC-kod 05.21)

HB 260 för gjutjärn (CMC-kod 08.2)

CMC = Coromant materialklassificering (Coromant Material Classification).

Se korsreferenslista för material på sid 4.

Dragbrottsgräns N/mm ²	Vickers	Brinell	Rockwell		Dragbrottsgräns N/mm ²	Vickers	Brinell	Rockwell
	HV	HB	HRC	HRB	HV	HB	HRC	
255	80	76,0	—	—	1030	320	304	32,2
270	85	80,7	—	41,0	1060	330	314	33,3
285	90	85,5	—	48,0	1095	340	323	34,4
305	95	90,2	—	52,0	1125	350	333	35,5
320	100	95,0	—	56,2	1155	360	342	36,6
350	110	105	—	62,3	1190	370	352	37,7
385	120	114	—	66,7	1220	380	361	38,8
415	130	124	—	71,2	1255	390	371	39,8
450	140	133	—	75,0	1290	400	380	40,8
480	150	143	—	78,7	1320	410	390	41,8
510	160	152	—	81,7	1350	420	399	42,7
545	170	162	—	85,0	1385	430	409	43,6
575	180	171	—	87,5	1420	440	410	44,5

Vi har en dragbrottsgräns på 290 N/mm² och ska bestämma motsvarande Brinell HB-värde. Från tabellen har att dragbrottsgränsen 285 N/mm² motsvarar ett Brinell HB-värde 85,5 och att dragbrottsgränsen 305 N/mm² motsvarar ett Brinell HB-värde 90,2. För att lösa uppgiften använder vi oss av linjär interpolation och skriver ett program som kallar på funktionen `interp`.

```
# hb_brinell.py

def interp(x1,y1,x2,y2,x):
    """
        linjär interpolation mellan två punkter (x1,y1) och (x2,y2).
        givet ett x beräknar vi motsvarande y
    """
    k = (y2-y1)/(x2-x1)    # riktningskoefficient
    m = y1 - k*x1          # m-värde
    y = k*x + m            # sätt in x i linjens ekvation
    return y

# här börjar det egentliga programmet
db1 = 285    # samhörande värden på drabrottsgräns
bhb1 = 85.5  # och Brinell HB-värden
db2 = 305
bhb2 = 90.2

# interpolera
db   = 290
bhb = interp(db1,bhb1,db2,bhb2,db)
print('Brinell HB-värde',round(bhb,1))
```

Då vi kör programmet får vi

Brinell HB-värde 86.7

vilket verkar mycket rimligt. □

8.4 Funktioner – odefinierade lokala variabler

En lokal variabel som inte har blivit tilldelad ett värde inne i funktionen innan variabeln används, tilldelas värdet av motsvarande variabel i det anropande programmet. Användning av odefinierade lokala variabler kan vara motiverat då dessa är fysikaliska konstanter eller andra konstanta kvantiteter. I övrigt avråder vi starkt från detta.

Exempel 8.5. Höjden h för en jordcirkulerande satellit med omloppstiden T ges av formeln (jfr övning 9 i kapitel 3)

$$h = \left(\frac{GMT^2}{4\pi^2} \right)^{1/3} - R,$$

där $G = 6.67 \times 10^{-11} \text{ m}^3\text{kg}^{-1}\text{s}^{-2}$ är gravitationskonstanten, $M = 5.97 \times 10^{34} \text{ kg}$ är jordens massa och $R = 6.371 \times 10^6 \text{ m}$ dess radie. Programmet **satellit.py** läser in T från skärmen och kallar på funktionen **altitude** för att beräkna h . Konstanterna G , M , R tilldelas i det anropande programmet och används sedan av funktionen.

```
# satellit.py
import numpy as np

def altitude(T):
    h = (G*M*T**2/(4*np.pi**2))**(1/3) - R
    return h

# här börjar själva programmet
# definiera konstanter
G = 6.67e-11           # gravitationskonstanten
M = 5.97e24              # jordens massa
R = 6.371e6               # jordens radie

T = float(input('Ge omloppstid '))
h = altitude(T)/1000      # h i km
print('Höjd i km', round(h))
```

Då vi kör programmet och matar in $T = 86\,400$, motsvarande en omloppstid på ett dygn, får vi

Höjd i km 35856

En satellit som rör sig i takt med jordens rotation sägs vara i en geostationär bana. □

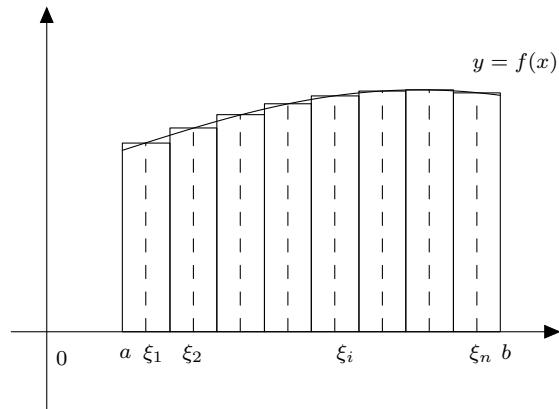
8.5 Funktionsnamn som invariabler

Invariablerna kan referera till tal, listor, vektorer etc. De kan också referera till andra funktioner. Detta kan användas för att skriva generella funktioner för att integrera eller beräkna nollställen, och då behöver funktionen som utgör integranden eller funktionen för vilken vi ska bestämma nollstället skickas med som invariabel.

Exempel 8.6. Låt $f(x)$ vara en funktion i intervallet $[a, b]$. Arean under grafen kan, enligt figur 8.5, approximeras av mittensumman

$$I = (f(\xi_1) + f(\xi_2) + \dots + f(\xi_n))\Delta x,$$

där $\Delta x = (b - a)/n$ är intervallbredden.



Figur 8.5: Arean under grafen kan approximeras av mittensumman.

Programmet **area.py** innehåller funktionerna **mittensumma**, som beräknar summan, och **f**, som definierar funktionen. Intervallgränserna a och b samt antalet intervall n läses in från skärmen och sedan kallas programmet på **mittensumma**

```
# area.py
import numpy as np

def mittensumma(f,a,b,n):
    """ area under grafen via mittensumma """
    dx = (b-a)/n
    xi = np.linspace(a+dx/2,b-dx/2,n)
    I = np.sum(f(xi))*dx
    return I

def f(x):
    y = x**2
    return y

# Här börjar själva programmet
a, b = input('Ge a och b, kommasseparerade ').split(',')
a, b = float(a), float(b)
n = int(input('Ge antalet punkter '))
I = mittensumma(f,a,b,n)          # anrop med funktionsnamn
print('Arean är: ',I)
```

Då vi kör programmet och matar in 0, 1 och 100 får vi

Arean är: 0.33332500000000004

Ovan har vi definierat $f(x)$ i en funktion inne i programmet. Alternativt, och lite mindre omständligt, kan vi införa $f(x)$ som en lambda-funktion

```
f = lambda x: x**2
```

Anropet av funktionen **mittensumma** ser likadant ut i båda fallen. □

Exempel 8.7. Programmet **skrivtabell.py** innehåller funktionen **tabell** som skriver ut en värdeatabell av en funktion definierad som en lambda-funktion.

```
# skrivtabell.py
import numpy as np

def tabell(f,x):
    print('x      f(x)')
    print('-----')
    for i in range(len(x)):
        print(round(x[i],2), ' ', round(f(x[i]),2))

# Här börjar själva programmet
f = lambda x: np.sqrt(x)
x = [1,2,3,4,5]
tabell(f,x)
```

Då vi kör programmet får vi följande utskrift

x	f(x)
1	1.0
2	1.41
3	1.73
4	2.0
5	2.24

□

Del II

Databehandling och modellering

Kapitel 9

Databehandling

I detta kapitel ska vi titta på lägesmått, medelvärde och median, och spridningsmått, standardavvikelse, av data. Vi ska också se hur data kan presenteras med hjälp av histogram och stolpdiagram. Vi går sedan vidare och undersöker vanliga fördelningar av experimentella data.

9.1 Summasymbolen

Här introducerar vi summasymbolen, Σ , vilken ger oss ett bekvämt sätt att skriva en summa.

Antag att vi har en följd av tal $x_1, x_2, x_3, x_4, \dots$ och att vi vill skriva summan av de sju första talen. Man skriver då

$$\sum_{k=1}^7 x_k.$$

Bokstaven k kallas *summationsindex*. Skrivsättets innebörd är att man i termen x_k skall byta ut summationsindex k mot talen 1 till 7 och sedan addera de erhållna termerna, dvs.

$$\sum_{k=1}^7 x_k = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7.$$

Vilken bokstav man använder som summationsindex är likgiltigt (bortsett att man inte får välja en bokstav som i sammanhanget har en annan betydelse). Så t.ex.

$$\sum_{k=1}^7 x_k = \sum_{m=1}^7 x_m = \sum_{p=1}^7 x_p$$

Exempel 9.1. I dessa exempel är termerna givna av formler.

a) $\sum_{k=1}^6 k = 1 + 2 + 3 + 4 + 5 + 6 = 21$

Här var termerna $x_k = k$

b) $\sum_{n=1}^5 n^2 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2 = 1 + 4 + 9 + 16 + 25 = 55$

Här var termerna $x_n = n^2$

□

Exempel 9.2. Skriv

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6}$$

med summatecknen.

Vi börjar med att skriva summan som

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6}.$$

Det är nu klart att summan kan skrivas som

$$\sum_{i=1}^6 \frac{1}{i}.$$

□

Normal beräknar man inte summor för hand utan använder Python. För att beräkna summan av tal givna som element i en vektor använder vi det inbyggda kommandot `np.sum`, se avsnitt 5.14. Summor av tal givna med hjälp av en formel beräknas enklast med hjälp av en for-loop.

Exempel 9.3. Teckna med hjälp av summatecknet summan av:

- (a) de 25 första positiva hela talen
- (b) de inverterade talen till de 100 första positiva hela talen
- (c) kvadraten på alla positiva heltalet från och med 5 till och med 50.

Beräkna sedan summorna med hjälp av Python.

Summorna skrivna med summatecken är

$$(a) \sum_{i=1}^{25} i \quad (b) \sum_{i=1}^{100} \frac{1}{i} \quad (c) \sum_{i=5}^{50} i^2$$

Följande kommandon beräknar summorna. Notera att i Python så tar vi det övre summationsindexet alltid som ett mer än indexet som ges i formeln för summan.

```
import numpy as np
print('Uppgift a')
# med for-loop
s = 0                      # initialisera summan till 0
for i in range(1,26):       # loopa över termer
    s = s + i                # addera term till summan
print('summa med for-loop:',s)
# med np.sum
x = np.arange(1,26)          # vektor med heltalet från 1 till 25
print('summa med np.sum :',np.sum(x))

print('Uppgift b')
# med for-loop
s = 0
for i in range(1,101):
    s = s + 1/i
print('summa med for-loop:',s)
# med np.sum
x = np.arange(1,101)         # vektor med heltalet från 1 till 100
```

```

print('summa med np.sum :',np.sum(1/x))

print('Uppgift c')
# med for-loop
s = 0
for i in range(5,51):
    s = s + i**2
print('summa med for-loop:',s)
# med np.sum
x = np.arange(5,51)           # vektor med heltal från 5 till 50
print('summa med np.sum :',np.sum(x**2))

```

Då vi kör programmet får vi

```

Uppgift a
summa med for-loop: 325
summa med np.sum : 325

```

```

Uppgift b
summa med for-loop: 5.187377517639621
summa med np.sum : 5.187377517639621

```

```

Uppgift c
summa med for-loop: 42895
summa med np.sum : 42895

```

Om man kan använda `np.sum` så är det att rekommendera. □

9.2 Medelvärde, frekvenstabell och histogram

Medelvärdet är ett mått på det genomsnittliga värdet av en samling tal. Om vi har talen $x_1, x_2, x_3, \dots, x_n$ beräknas medelvärdet genom

$$\bar{x} = \frac{x_1 + x_2 + \dots + x_n}{n} \quad \text{summan av alla talen delat med antal tal}$$

Om vi använder oss av summatecknet kan samma medelvärdet också skrivas som

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i.$$

Det finns olika beteckningar för medelvärdet och man ser ofta \bar{x} , μ , m eller M . Medelvärde beräknas i Python med hjälp av det inbyggda kommandot `np.mean`, se avsnitt 5.14.

Medelvärde, frekvenstabell och stolpdiagram

Exempel 9.4. Vi räknade antalet bokstäver i 25 på varandra följande ord i en svensk text. Resultatet finns i tabell 9.1.

Tabell 9.1: Antalet bokstäver i 25 på varandra följande ord i en svensk text.

6	3	6	2	5	3	5	6	6	10
4	7	4	2	3	1	12	7	2	3
3	4	10	8	2					

Vi kan sammanställa materialet i en frekvenstabell. I tabellen anger f frekvensen, dvs. hur många gånger ett ord med en viss ordlängd förekommer. f_x är produkten av x och f och anger det totala antalet bokstäver i alla ord med längden x .

Tabell 9.2: *Frekvenstabell*

ordlängd x	frekvens f	f_x
1	1	1
2	4	8
3	5	15
4	3	12
5	2	10
6	4	24
7	2	14
8	1	8
9	0	0
10	2	20
11	0	0
12	1	12
summa	25	124

Materialet kan illustreras i ett stolpdiagram. Detta får man genom att för varje ordlängd avsätta en mot frekvensen svarande sträcka. Följande Pythonkod beräknar och skriver ut medelvärdet och ritar ett stolpdiagram för att illustrera data.

```
import numpy as np
import matplotlib.pyplot as plt

data = np.array([6,3,6,2,5,3,5,6,6,10,4,7,4,2,3,1,12,7,2,3,3,4,10,8,2])
print('Medelvärde: ',np.mean(data))

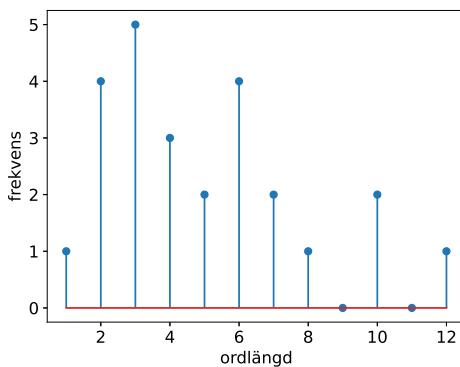
x = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
f = np.array([1,4,5,3,2,4,2,1,0,2,0,1])

fig, ax = plt.subplots()
ax.stem(x,f)                                     # stolpdiagram
ax.set_xlabel('ordlängd', fontsize = 14)
ax.set_ylabel('frekvens', fontsize = 14)
ax.tick_params(labelsize=14)
```

Då vi kör programmet får vi utskriften

Medelvärde: 4.96

Motsvarande stolpdiagram visas i figur 9.1. □



Figur 9.1: Stolpdiagram som visar frekvensen av ord med olika längd.

Medelvärde, frekvenstabell och histogram

Exempel 9.5. 40 flickor i gymnasieskolans årskurs 1 tillfrågades om sina vikter. Svaren finns i tabell 9.3.

Tabell 9.3: Vikt hos flickor i gymnasieskolans årskurs 1.

53	52	54	56	50	49	52	50	57	52
55	60	51	43	53	64	57	58	57	50
56	50	55	50	59	42	55	45	45	52
46	47	46	58	48	50	55	47	53	51

Vikterna varierar från 42 kg upp till 64 kg. För att få en överskådlig frekvenstabell är det lämpligt att indela vikterna i lika stora grupper eller *klasser*. Vi kan t.ex. låta den första klassen bestå av vikterna 42, 43, 44 (dvs. vikter mellan 41.5 och 44.5), den andra av vikterna 45, 46, 47 (dvs. vikter mellan 44.5 och 47.5) osv. Sedan räknar vi hur många vikter det finns i varje klass.

Tabell 9.4: Klassindelning och frekvens.

klass	frekvens
42–44	2
45–47	6
48–50	8
51–53	9
54–56	7
57–59	6
60–62	1
63–65	1
summa	40

Frekvenstabellen 9.4 kan illustrera med ett diagram. Det består av rektanglar, vilkas baser är sträckorna mellan klassgränserna 41.5, 44.5, 47.5 etc. upp till 65.5. Rektanglarnas höjder är lika med frekvenserna. Ett sådant diagram kallas ett *histogram*, se avsnitt 6.8. Följande Pythonkod beräknar och skriver ut medelvärdet och ritar ett histogram för att illustrera data. Programmet skriver också ut information från histogramkommandot i form av frekvenser och klassgränser.

```

import numpy as np
import matplotlib.pyplot as plt

klasser = np.arange(41.5,65.6,3) # klassgränser 41.5-44.5, 44.5-47.5 etc
data = np.array([53,52,54,56,50,49,52,50,57,52,55,60,51,43,53,64,57,58,57,50,
                 56,50,55,50,59,42,55,45,45,52,46,47,46,58,48,50,55,47,53,51])

print('Medelvärde ',np.mean(data))

fig, ax = plt.subplots()
frekvens, klasser, patch = ax.hist(data,bins=klasser,edgecolor='black')
ax.set_xlabel('vikt i kg',fontsize=14)
ax.set_ylabel('frekvens',fontsize=14)
ax.tick_params(labelsize=14)

print('klassgränser',klasser)
print('frekvens      ',frekvens)

```

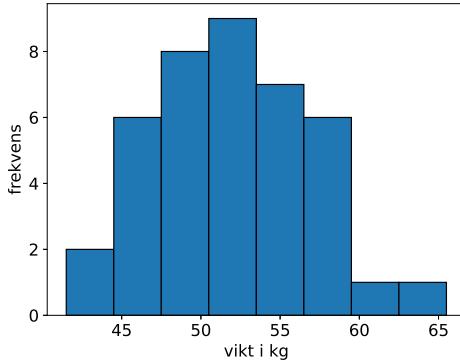
Då vi kör programmet svarar Python

```

Medelvärde 52.075
klassgränser [41.5 44.5 47.5 50.5 53.5 56.5 59.5 62.5 65.5]
frekvens     [2. 6. 8. 9. 7. 6. 1. 1.]

```

Frekvenserna är precis de vi har i tabellen ovan (det är ju vad vi förväntar oss). Motsvarande histogram visas i figur 9.2. \square



Figur 9.2: Histogram som visar frekvensen av vikter i olika klasser.

Exempel 9.6. Ofta låter man Python automatiskt välja klassgränser, och då man skall rita ett histogram anger man bara antalet klasser. I tabellen 9.5 har vi årsmedeltemperaturen i Stockholm från 1831 till 1860.

Tabell 9.5: Årsmedeltemperatur i Stockholm från 1831 till 1860.

5.4	5.6	5.8	6.8	5.6	5.0	4.8	3.9	5.3	5.3
5.6	6.4	5.8	4.1	5.0	6.5	5.5	5.7	5.1	5.4
6.0	6.1	5.6	6.5	4.9	4.7	6.7	7.0	6.5	4.9

Följande Pythonkod beräknar och skriver ut medelvärdet av årstemperaturerna och ritar ett histogram för att illustrera data. Programmet skriver också ut information från histgramkommandot i form av frekvenser och klassgränser.

```
import numpy as np
import matplotlib.pyplot as plt

data = np.array([5.4,5.6,5.8,6.8,5.6,5.0,4.8,3.9,5.3,5.3,
                 5.6,6.4,5.8,4.1,5.0,6.5,5.5,5.7,5.1,5.4,
                 6.0,6.1,5.6,6.5,4.9,4.7,6.7,7.0,6.5,4.9])

print('Medelvärde ',round(np.mean(data),1))    # avrunda till en decimal

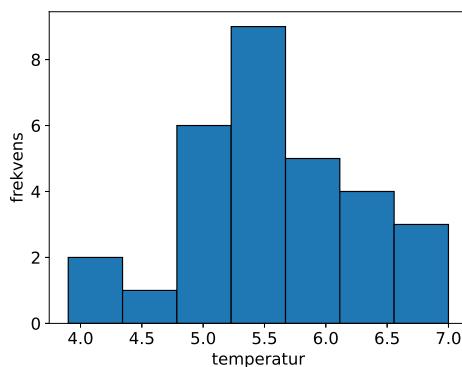
fig, ax = plt.subplots()
frekvens, klasser, patch = ax.hist(data,bins=7,edgecolor='black')
ax.set_xlabel('temperatur',fontsize=14)
ax.set_ylabel('frekvens',fontsize=14)
ax.tick_params(labelsize=14)

print('klassgränser',klasser)
print('frekvens   ',frekvens)
```

Då vi kör programmet får vi utskriften

```
Medelvärde 5.6
klassgränser [3.9          4.34285714 4.78571429 5.22857143 5.67142857 6.11428571
               6.55714286 7.          ]
frekvens      [2. 1. 6. 9. 5. 4. 3.]
```

Som en jämförelse var årsmedeltemperaturen i Stockholm för perioden 1991–2020 hela 7.9 grader, en ökning på 2.3 grader! Motsvarande histogram visas i figur 9.3. □



Figur 9.3: Histogram som visar frekvensen för årsmedeltemperaturen i Stockholm under perioden 1831–1860.

9.3 Standardavvikelse

Standardavvikelse eller standarddeviation är ett statistiskt mått på hur mycket de olika datavärdena avviker från medelvärdet. Om vi har datavärdena $x_1, x_2, x_3, \dots, x_n$ och låter medelvärde betecknas med \bar{x} så ges standardavvikelsen av

$$\sigma = \sqrt{\frac{1}{n} \left((x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \dots + (x_n - \bar{x})^2 \right)}.$$

Samma sak skrivet med summatecken blir

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2}.$$

Precis som för medelvärdet finns det flera beteckningar för standardavvikelsen och man ser både σ och s . Standardavvikelsen (eng. standard deviation) beräknas i Python med hjälp av det inbyggda kommandot `np.std`, se avsnitt 5.14.

Exempel 9.7. I tabellen 9.6 har vi längden (i centimeter) av tio artonåriga pojkar.

Tabell 9.6: *Längd för 10 artonåriga pojkar.*

179	167	183	179	177	178	182	187	176	172
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Medelvärdet av dessa längder är 178 cm.

Vi vill nu beräkna standardavvikelsen för att se hur pass spridda längderna är kring medelvärdet och räknar därför ut differenserna $179 - 178$, $167 - 178$, $183 - 178$ osv. och kvadrerar dessa differenser. Vi får (kontrollera själv)

1, 121, 25, 1, 1, 0, 16, 81, 4, 36

Standardavvikelsen fås nu genom att summa de kvadrerade avvikelserna, dela med antalet och dra roten ur

$$\sigma = \sqrt{\frac{1 + 121 + 25 + 1 + 1 + 0 + 16 + 81 + 4 + 36}{10}} \approx 5.3$$

För att beräkna medelvärde och standardavvikelse med hjälp av Python ger vi kommandona

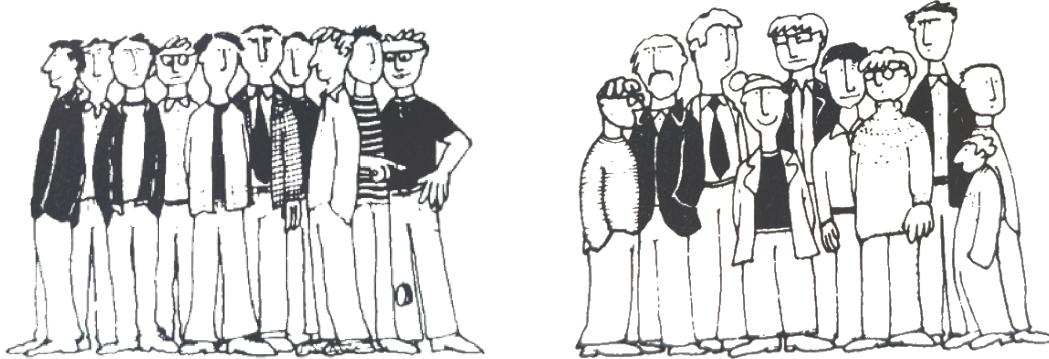
```
import numpy as np

l = np.array([179,167,183,179,177,178,182,187,176,172])
print('medelvärde      ',np.mean(l))
print('standardavvikelse',round(np.std(l),1))
```

Vi får följande utskrift

```
medelvärde      178.0
standardavvikelse 5.3
```

Sammanfattningsvis kan vi säga att om datavärdena varierar lite så har vi liten standardavvikelse. Om datavärdena varierar mycket har vi stor standardavvikelse. Detta illustreras i figur 9.4. \square



Figur 9.4: Längderna varierar lite, liten standardavvikelse (vänster). Längderna varierar mycket, stor standardavvikelse (höger).

Exempel 9.8. Vi går tillbaka till temperaturdata i exempel 6.10. I filen **T.txt**, vilken som vanligt kan laddas ner från Canvas, finns vektor T med uppmätta temperaturvärden under 23 år för en mätstation i Skåne. Följande program läser filen, beräknar medelvärdet och standardavvikelsen av ritar ett histogram med 20 klasser.

```
import numpy as np
import matplotlib.pyplot as plt

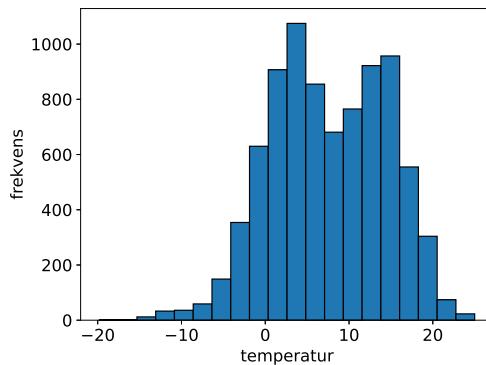
T = np.loadtxt('T.txt')
print('medelvärde      ', round(np.mean(T),1))
print('standardavvikelse', round(np.std(T),1))

fig, ax = plt.subplots()
ax.hist(T,bins=20,edgecolor='black')
ax.set_xlabel('temperatur',fontsize=14)
ax.set_ylabel('frekvens',fontsize=14)
ax.tick_params(labelsize=14)
```

Då vi utför kommandona svarar Python

```
medelvärde      7.6
standardavvikelse 6.9
```

Histogrammet visas i figur 9.5. □



Figur 9.5: Histogram som visar frekvensen av dagliga temperaturer under 23 år.

9.4 Median

För data där ett eller flera värden avviker kraftigt från de andra ger medelvärdet ibland en skev bild av vad som är 'normalt'. I dessa fall är det ofta bättre att använda medianen, dvs. det värde som hamnar precis i mitten då värdena sorterats i storleksordning. Om data innehåller ett jämnt antal värde fås medianen som medelvärdet av de två mittersta värdena. I Python beräknas medianen med hjälp av det inbyggda kommandot `np.median`, se avsnitt 5.14.

Exempel 9.9. För att få ett mått på löneläget i ett företag använder man ofta medianen. Anledningen är att där kan finnas ett fåtal individer med mycket hög lön i jämförelse med flertalet. Medelvärdet skulle ge intrycket av att lönen är högre för den 'normale' anställda än vad den faktiskt är. Medianen ger här ett bättre mått. I tabell 9.7 har vi lönen för personalen på ett litet företag.

Tabell 9.7: *Månadslöner i kronor för personal på ett litet företag. Den höga lönén, 255 000 kr, är den för direktören.*

31 000	33 500	35 300	34 800	255 000	39 100	32 800	30 500	34 300	38 700
--------	--------	--------	--------	---------	--------	--------	--------	--------	--------

Sorterar vi lönerna får vi

$$30\ 500, 31\ 000, 32\ 800, 33\ 500, 34\ 300, 34\ 800, 35\ 300, 38\ 700, 39\ 100, 255\ 000$$

Medianen fås som medelvärdet av de två mittersta värdena

$$\text{median} = \frac{34\ 300 + 34\ 800}{2} = 34\ 550.$$

Medelvärdet räknar vi ut till 56 500, vilket i detta fallet är ganska missvisande för att ange det 'normala' löneläget.

Median och medelvärde beräknas med följande program

```
import numpy as np

l = np.array([31000,33500,35300,34800,255000,39100,32800,30500,34300,38700])
print('medelvärde',round(np.mean(1)))
print('median   ',round(np.median(1)))
```

Då vi kör programmet får vi

```
medelvärde 56500
median      34550
```

□

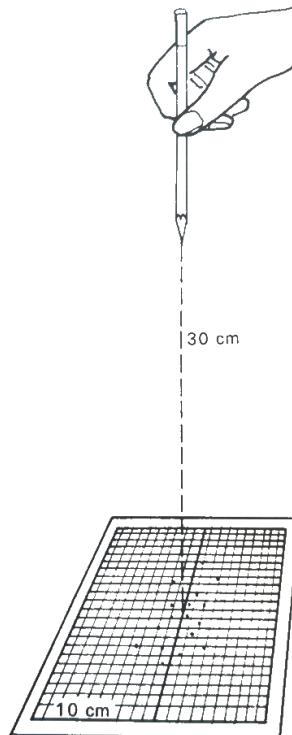
9.5 Slumpmässiga försök – normalfördelning

Många fenomen i naturen och samhället är sådana att observationerna följer ett klockliknande mönster sådant att flest värden ligger kring medelvärdet för att successivt avta ju längre bort man kommer från medelvärdet. I gränsen för oändligt många observationer får man att värdena fördelar sig efter en frekvensfunktion, den så kallade normalfördelningsfunktionen, given av

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2},$$

där μ är fördelningens medelvärde och σ standardavvikelsen. Om du inte har sett denna typ av funktioner, bli inte förfärad utan läs vidare i avsnitt 11.9.

Exempel 9.10. Vi lade ett millimeterrutat papper på ett bord. 10 cm från rutnätets vänstra kant har vi dragit en linje som var parallell med kanten. Sedan försökte vi pricka linjen med spetsen från en kulspetspenna. Vi släppte pennan från 30 centimeters höjd, se figur 9.6.

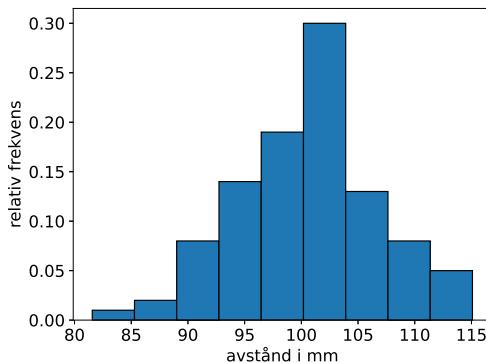


Figur 9.6: Vi släpper pennan från 30 centimeters höjd och försöker träffa linjen. Sedan mäter vi var vi hamnar

Vi gjorde det här försöket 100 gånger och fick 100 prickar på pappret. Sedan såg vi efter hur många millimeter från rutnätets kant prickarna låg. Data har sammanställts nedan

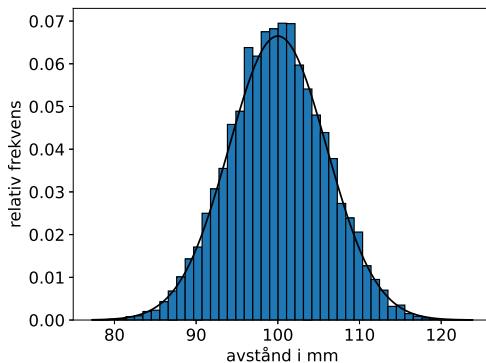
101	112	95	105	99	102	103	95	102	104	98	93.	103	98
98	95	104	96	104	101	90	95	100	105	103	114	100	101
104	93	95	99	101	97	94	105	87	103	100	99	104	99
107	103	99	100	98	89	103	105	103	96	104	109	90	90
96	99	95	97	108	104	102	102	99	110	86	108	103	112
103	108	101	104	96	102	105	103	110	103	106	98	82	91
113	115	95	104	95	102	97	101	102	89	100	106	111	92
93	108												

Vi åskådliggjorde sedan data i ett histogram med 9 klasser, där vi på y -axeln hade relativ frekvens (frekvens delat med antal observationer). Histogrammet visas i figur 9.7.



Figur 9.7: Histogram som visar den relativa frekvensen som funktion av avståndet i mm. Histogrammet baseras på resultat från 100 försök.

Vi gjorde nu om experimentet 10 000 gånger (ja, det var jättetråkigt). Histogrammet, nu med 45 klasser, visas i figur 9.8 tillsammans med en frekvensfunktion $f(x)$ med $\mu = 100$ och $\sigma = 6$. Data från försöket är till en mycket god approximation normalfördelad. För normalfördelade data faller drygt 68 % av observationerna inom en standardavvikelse från medelvärdet, drygt 95 % inom två standardavvikeler från medelvärdet och drygt 99.7 % är inom tre standardavvikeler från medelvärdet, se också avsnitt 11.9. \square



Figur 9.8: Histogram som visar den relativa frekvensen som funktion av avståndet i mm. Histogrammet baseras på resultat från 100 försök. Vi har även plottat frekvensfunktionen $f(x)$ med $\mu = 100$ och $\sigma = 6$.

9.6 Normalfördelade slumptal

I Python kan vi använda den inbyggda funktionen `np.random.rndn` för att generera normalfördelade slumptal med medelvärde 0 och standardavvikelse 1. För att få normalfördelningar med andra medelvärden och standardavvikeler skalar man och förskjuter sina värden.

Exempel 9.11.

- (a) Följande kommandon genererar 100 000 normalfördelade slumptal med medelvärde $\mu = 5$ och standardavvikelse $\sigma = 2$. Dessutom plottas ett histogram med 50 klasser.

```
import numpy as np
import matplotlib.pyplot as plt
```

```

x = 5 + 2*np.random.randn(100000)      # addition av 5 ger medelvärde 5
                                         # multiplikation med 2 ger standaravv. 2
fig, ax = plt.subplots()
ax.hist(x,bins=50,edgecolor='black')
ax.set_xlabel('värden',fontsize=14)
ax.set_ylabel('frekvens',fontsize=14)
ax.tick_params(labelsize=14)

```

Histogrammet återfinns till vänster i figur 9.9.

(b) Följande kommandon genererar 100 000 normalfördelade slumptal med medelvärde $\mu = -1$ och standardavvikelse $\sigma = 4$. Dessutom plottas ett histogram med 50 klasser.

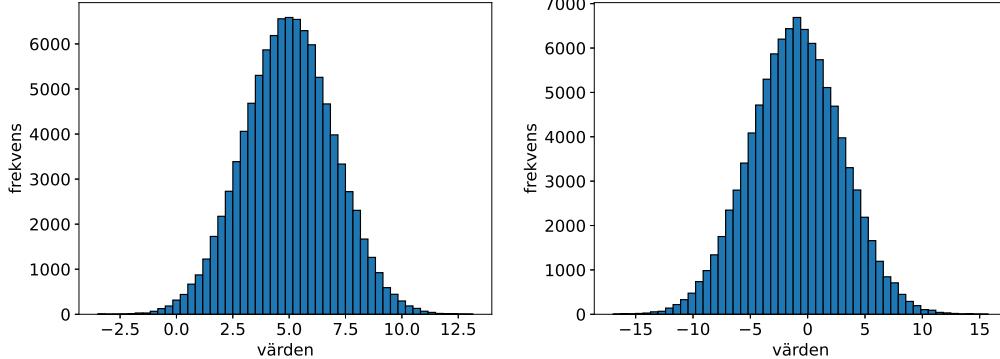
```

import numpy as np
import matplotlib.pyplot as plt

x = -1 + 4*np.random.randn(100000)      # addition av -1 ger medelvärde -1
                                         # multiplikation med 4 ger standaravv. 4
fig, ax = plt.subplots()
ax.hist(x,bins=50,edgecolor='black')
ax.set_xlabel('värden',fontsize=14)
ax.set_ylabel('frekvens',fontsize=14)
ax.tick_params(labelsize=14)

```

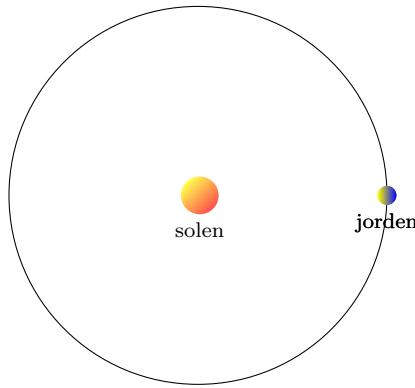
Histogrammet för denna fördelning visas till höger i figur 9.9. □



Figur 9.9: Normalfördelade slumptal med $\mu = 5$ och standardavvikelse $\sigma = 2$ (till vänster) samt med $\mu = -1$ och standardavvikelse $\sigma = 4$ (till höger).

9.7 Tillämpning: känslighetsanalys

Genererade slumptal kan användas för att undersöka hur fel propagerar (fortplantas) och därmed ge en uppskattning av felet hos en storhet som i sin tur beror på variabler som var och en är associerade med fel eller osäkerheter. Detta är en mycket vanlig teknik som används i en mängd sammanhang och som ibland går under benämningen känslighetsanalys.



Figur 9.10: Planeter rör sig i ellipsformade banor kring solen under inverkan av gravitationskraften.

I figur 9.10 har vi en planet som rör sig kring motsvarande sol. Kraften F som håller jorden i en bana kring solen beror av massorna för jorden och solen samt avståndet mellan dem och ges av Newtons gravitationslag

$$F = G \frac{M_1 M_2}{r^2}.$$

Vår uppgift är att undersöka hur felen i M_1 , M_2 och r propagerar (fortplantas) och resulterar i ett fel (eller osäkerhet) i F . Vi antar följande värden och fel i form av standardavvikelse för massorna och avståndet:

$$\begin{aligned} M_1 &= 60.00 \cdot 10^{24} \pm 0.05 \cdot 10^{24} \text{ kg} && \text{typisk planetmassa} \\ M_2 &= 20.0 \cdot 10^{30} \pm 0.03 \cdot 10^{30} \text{ kg} && \text{typisk solmassa} \\ r &= 1.50 \cdot 10^{11} \pm 0.05 \cdot 10^{11} \text{ m} && \text{typiskt avstånd} \\ G &= 6.67384 \cdot 10^{-11} \text{ m}^3 \text{kg}^{-1} \text{s}^{-2} && \text{gravitationskonstanten (exakt)} \end{aligned}$$

Programmet nedan genererar normalfördelade slumptal för massorna och avståndet och beräknar den resulterande kraften. Vi tar 100 000 slumptal för var och en av massorna och avståndet, vilka ger 100 000 olika F -värden, vilka plottas i ett histogram. Dessutom beräknas fördelningens medelvärde och standardavvikelse.

```
import numpy as np
import matplotlib.pyplot as plt

n      = 100000          # antal slumptärden
M1    = 6e24             # massa planet
sM1   = 0.05e24          # standardavvikelse planet
M2    = 2e30              # massa sol
sM2   = 0.03e30          # standardavvikelse sol
r     = 1.5e11            # avstånd
sr   = 0.05e11           # standardavvikelse avstånd
G    = 6.67384E-11       # gravitationskonstanten

M1_vekt = M1 + sM1*np.random.randn(n)  # vektor med n normalfördelade slumptal
M2_vekt = M2 + sM2*np.random.randn(n)
r_vekt  = r + sr*np.random.randn(n)
F = G*M1_vekt*M2_vekt/r_vekt**2

fig,ax = plt.subplots()
```

```
ax.hist(F,50,edgecolor='black')
ax.set_xlabel('F',fontsize=14)
ax.set_ylabel('frekvens',fontsize=14)
ax.tick_params(labelsize=14)
print('medelvärde      ',np.mean(F))
print('standardavvikelse ',np.std(F))
```

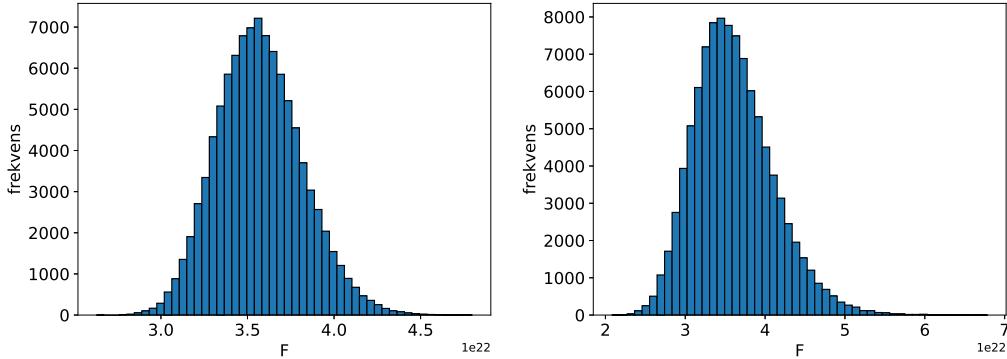
Då vi kör programmet svarar Python

```
medelvärde      3.5705045685291973e+22
standardavvikelse 2.4723040468476237e+21
```

och skapar histogrammet till vänster i figur 9.11. Från histogrammet ser vi att kraften F till en god approximation är normalfördelad. Om vi nu fördubblar osäkerheten i avståndet r och kör programmet med $1.50 \cdot 10^{11} \pm 0.1 \cdot 10^{11}$ m får vi följande utskrift

```
medelvärde      3.60821954985109e+22
standardavvikelse 4.979642197428042e+21
```

och skapar histogrammet till höger i figur 9.11. Kraften är nu inte längre normalfördelad utan lite skev med förhållandevis fler värden för höga värden på F . Läsaren uppmanas att experimentera lite med olika fel (standardavvikeler) för M_1 , M_2 och r . Detta är precis innebördens av känslighetsanalys: förändra och se vad som händer.



Figur 9.11: *Fördelningar av 100 000 F -värden under antagandet om normalfördelade massor och avstånd. I histogrammet till höger har standardavvikelsen för r fördubblats, vilket resulterar i en lite skev fördelning för F .*

9.8 Tillämpning: histogram från frekvenstabeller

Ofta har man inte tillgång till rådata, utan det vi kan ladda ner finns i termer av färdig klassindelning och frekvens. För att generera histogram från denna typ av data använder man sig i Python av kommandot `stairs`.

Officiell statistik för Sverige finns att ladda ner från SCB <https://www.scb.se/hitta-statistik/>. Från SCBs sida har vi laddat ner befolningsstatistik från 2023.

ålder,	kön	,	antal
0-4 år,	kvinnor	,	271312
5-9 år,	kvinnor	,	298653
10-14 år,	kvinnor	,	306171
15-19 år,	kvinnor	,	296877

20-24 år, kvinnor	,280864
25-29 år, kvinnor	,310619
30-34 år, kvinnor	,380273
35-39 år, kvinnor	,347937
40-44 år, kvinnor	,319679
45-49 år, kvinnor	,316405
50-54 år, kvinnor	,326231
55-59 år, kvinnor	,336975
60-64 år, kvinnor	,291833
65-69 år, kvinnor	,273918
70-74 år, kvinnor	,263781
75-79 år, kvinnor	,263325
80-84 år, kvinnor	,177878
85-89 år, kvinnor	,106066
90-94 år, kvinnor	,52260
95-99 år, kvinnor	,15857
100-104 år, kvinnor	,2274

För att använda kommandot `stairs` måste vi börja med att definiera klassgränserna. Klasserna omfattar 5 år och klassgränserna blir då $-0.5, 4.5, 9.5, \dots, 104.5$. Dessa anges i en vektor. För att beräkna medelvärdet kan vi låta varje klass representeras av klassmitten $2, 7, 12, \dots, 102$. Vi beräknar sedan summan av klassmitt gånger frekvens och delar med summan av frekvenserna

$$\text{medelvärde} = \frac{\text{summa av klassmitt gånger frekvens}}{\text{summan av frekvenserna}}$$

Programmet som ritar histogrammet och bestämmer medelåldern ges nedan.

```
import matplotlib.pyplot as plt
import numpy as np

klassgrans = np.arange(-0.5,104.6,5)
klassmitt = np.arange(2,102.1,5)
frekvens = np.array([271312,298653,306171,296877,280864,310619,380273,
                     347937,319679,316405,326231,336975,291833,273918,
                     263781,263325,177878,106066,52260,15857,2274])

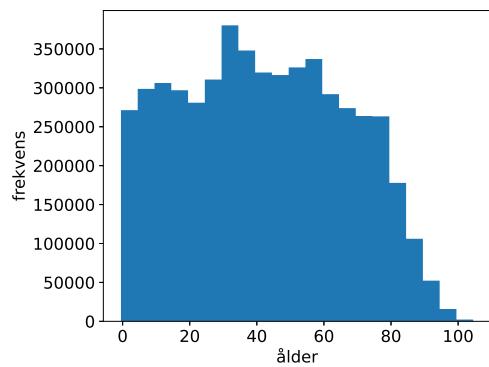
medel = np.sum(klassmitt*frekvens)/np.sum(frekvens)
print('Medelålder ',round(medel,1))

fig, ax = plt.subplots()
ax.stairs(frekvens, klassgrans, fill=True)
ax.set_xlabel('ålder', fontsize=14)
ax.set_ylabel('frekvens', fontsize=14)
ax.tick_params(labelsize=14)
```

Då vi kör programmet får vi utskriften

Medelålder 42.3

Histogrammet visas i figur 9.12.



Figur 9.12: *Folkmängd hos kvinnor efter ålder. Data för 2023.*

Kapitel 10

Funktionsklasser: polynom och rationella funktioner

Funktioner är ett av de viktigaste begreppen inom matematik och ligger till grund för mycket inom modellering och simulering. Vi börjar med några grundläggande begrepp för att sedan titta på de enkla, men viktiga, funktionsklasserna polynom och rationell funktioner.

10.1 Inledande exempel

Tabellen till höger bestämmer en mängd av talpar (x, y) nämligen

$$\{(0, 1), (1, 1), (2, 2), (3, 2), (4, 3), (5, 3)\}.$$

Denna mängd har följande egenskap:

Det finns inte två talpar som har samma första tal.

En sådan mängd av talpar kallas en *funktion*.

x	y
0	1
1	1
2	2
3	2
4	3
5	3

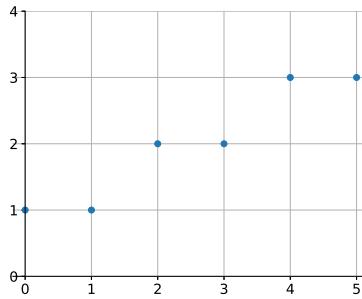
De första talen i talparen (dvs. talen i tabellens vänstra kolumn) bildar följande mängd:

$$D = \{0, 1, 2, 3, 4, 5\}.$$

Denna kallas funktionens *definitionsmängd*. De andra talen i talparen (dvs. talen i tabellens högra kolumn) bildar följande mängd:

$$V = \{1, 2, 3\}.$$

Den kallas funktionens *värdevärdemängd*. Mot varje tal x i definitionsmängden svarar ett tal y i värdevärdemängden. Talet y kallas *funktionsvärdet* för x . I tabellen ser vi t.ex. att funktionsvärdet för 2 är 2 och att funktionsvärdet för 4 är 3. Funktionens *graf* visas i figur 10.1.



Figur 10.1: En funktion bestående av ett antal talpar.

10.2 Linjära funktioner

Det är mycket vanligt att en funktion består av *oändligt många talpar*. En sådan funktion kan man förstås inte beskriva genom att räkna upp talpar. Man får istället ange en regel som talar om hur talparen ser ut.

Exempel 10.1. Här är en beskrivning av en funktion:

- 1) definitionsmängden består av alla reella tal
- 2) om det första talet i ett talpar är x , så skall det andra talet vara $2x - 1$.

Med hjälp av denna beskrivning kan vi räkna ut hur många talpar som helst. Om vi t.ex. sätter $x = -3$, så blir

$$2x - 1 = 2 \cdot (-3) - 1 = -6 - 1 = -7$$

och vi får talparet $(-3, -7)$. Detta och några andra av funktionens talpar finns i tabellen här bredvid.

x	$2x - 1$
-3	-7
-2	-5
-1	-3
0	-1
1	1
2	3
3	5

I koordinatsystemet i figur har vi markerat de punkter som motsvarar talparen. Som du ser ligger punkterna på en rät linje L . Funktionen består av oändligt många talpar. Funktionens graf består därför av oändligt många punkter. Man kan visa att *funktionens graf är hela linjen L* . Den regel som ger funktionens talpar kan skrivas på följande korta sätt

$$x \mapsto 2x - 1.$$

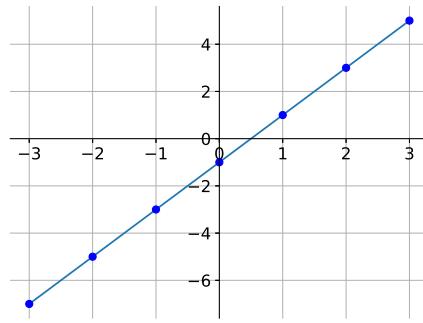
Pilen läses *avbildas på*. Skrivsättet används också som *beteckning* för funktionen. Funktionen kan också betecknas på följande sätt

$$y = 2x - 1$$

$$f(x) = 2x - 1$$

$$f : x \mapsto 2x - 1.$$

□



Figur 10.2: Grafen till funktionen $y = 2x - 1$ tillsammans med några markerade talpar.

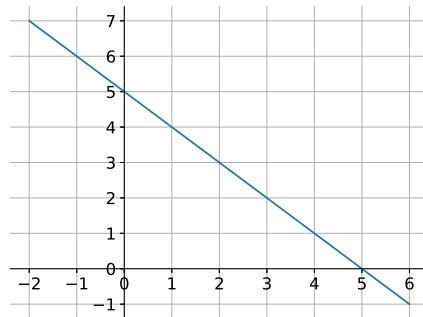
Exempel 10.2. Rita grafen till funktionen

$$x \mapsto 5 - x.$$

När vi liksom här inte anger någon definitionsmängd, så menar vi att definitionsmängden skall bestå av alla reella tal. Grafen ritas med hjälp av följande kommandon där vi väljer ett plotintervall som vi själva bedömer är intressant

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-2,6)
y = 5 - x
fig, ax = plt.subplots()
ax.plot(x,y)
ax.tick_params(labelsize=14)
ax.grid('on')
ax.set_xticks([-2,-1,0,1,2,3,4,5,6])
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_position('zero')
ax.spines['bottom'].set_position('zero')
```

Grafen visas i figur 10.3. □



Figur 10.3: Funktionen $y = 5 - x$.

En funktion vars graf är en rät linje kallas en *linjär funktion*. De två funktionerna i exemplen är alltså linjära. Man kan visa att funktionen

$$x \mapsto kx + m$$

är linjär, vilka talen k och m än är. Talet k kallas linjens *riktningskoefficient* och bestämmer linjens lutning.

10.3 Den räta linjens ekvation

Linjen i figuren går igenom punkten (x_1, y_1) och har riktningskoefficienten k . Vi ska bestämma en ekvation för linjen.

Eftersom riktningskoefficienten är k kan ekvationen skrivas

$$y = kx + m.$$

Ekvationen satisfieras (uppfylls) av talparet (x_1, y_1) :

$$y_1 = kx_1 + m.$$

Denna likhet subtraherar vi från ekvationen ovan. Då går m bort, och vi får följande ekvation för linjen:

$$y - y_1 = k(x - x_1).$$

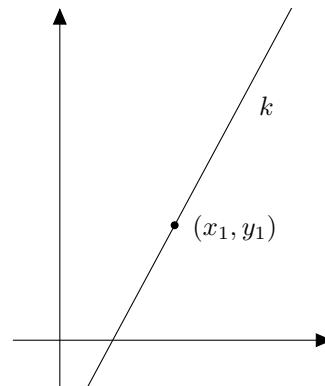
Exempel 10.3. Bestäm en ekvation för den räta linje, som går genom punkten $(-2, 5)$ och har riktningskoefficient -1 .

Formeln ovan med $x_1 = -2$, $y_1 = 5$ och $k = -1$ ger ekvationen

$$y - 5 = -1 \cdot (x - (-2)),$$

vilket är samma som

$$y = -x + 3.$$



□

Linjen i figuren till höger går igenom punkterna (x_1, y_1) och (x_2, y_2) . Vi ska bestämma linjens riktningskoefficient och en ekvation för linjen.

Vi kallar riktningskoefficienten k . Eftersom linjen går genom punkten (x_1, y_1) , så kan linjens ekvation skrivas:

$$y - y_1 = k(x - x_1)$$

Ekvationen satisfieras av talparet (x_2, y_2) :

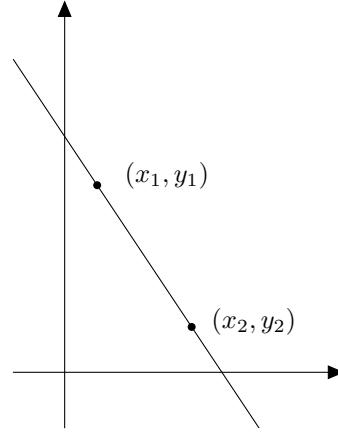
$$y_2 - y_1 = k(x_2 - x_1)$$

Vi antar att $x_2 \neq x_1$. Då är $x_2 - x_1 \neq 0$, division av båggen ledet med denna differens ger:

$$k = \frac{y_2 - y_1}{x_2 - x_1}$$

Denna formel ger linjens riktningskoefficient. Om vi sätter in uttrycket för k i första ekvationen ovan, så får vi följande ekvation för linjen genom de två punkterna:

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$$



Exempel 10.4.

- a) Vilken riktningskoefficient har den räta linjen genom punkterna $(3, -4)$ och $(-5, -2)$?

Vi sätter $x_2 = 3, y_2 = -4, x_1 = -5$ och $y_1 = -2$ i formeln för riktningskoefficienten:

$$k = \frac{y_2 - y_1}{x_2 - x_1} = \frac{-4 - (-2)}{3 - (-5)} = \frac{-4 + 2}{3 + 5} = \frac{-2}{8} = -\frac{1}{4}$$

- b) Bestäm en ekvation för den räta linje, som går genom punkterna $(5, 3)$ och $(-2, -1)$.

Sista ekvationen ovan ger:

$$y - 3 = \frac{3 - (-1)}{5 - (-2)}(x - 5)$$

$$y - 3 = \frac{4}{7}(x - 5)$$

$$y = \frac{4}{7}x + \frac{1}{7}$$

□

10.4 Proportionalitet

Om x och y är två variabler och det finns ett tal k sådant att

$$y = kx$$

så sägs y vara *proportionell* mot x . Talet k kallas *proportionalitetsfaktorn*.

Exempel 10.5. Antag att priset på en vara (t.ex. apelsiner) är 12 kr/kg. Om x kg kostar y kr, så är

$$y = 12x.$$

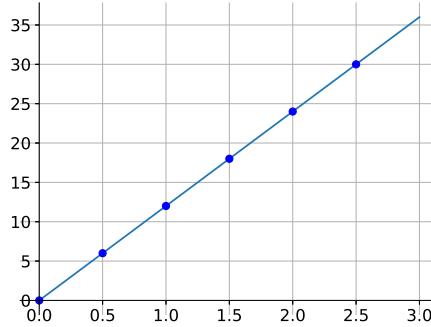
Priset är alltså proportionellt mot vikten. Proportionalitetsfaktorn betyder priset för 1 kg av varan. Ekvationen ovan bestämmer en oändlig mängd av talpar (några av dessa finns i tabellen nedan). Mängden av alla dessa talpar är en funktion. Med den beteckning vi tidigare använt kan funktionen skrivas så här:

$$x \mapsto 12x, \quad x \geq 0.$$

Olikheten visar att funktionens definitionsmängd består av talet noll och alla positiva tal. (Man kan ju inte köpa en negativ kvantitet av en vara.)

I fall som detta brukar man dock knappast använda skrivsättet $x \mapsto 12x$. Man nöjer sig med att skriva $y = 12x$, och man säger att y är en *funktion* av x , bestämd genom denna ekvation. Man kan också säga att priset är en funktion av vikten.

Funktionens graf är en från origo utgående stråle, vilken visas i figur 10.4 tillsammans med några beräknade talpar. \square



Figur 10.4: *Proportionalitet* $y = 12x$.

10.5 Polynom och polynomfunktioner

Ett *polynom* i en variabel x är ett uttryck av formen

$$a_0 + a_1x + a_2x^2 + \dots + a_nx^n \quad (a_n \neq 0).$$

Talen $a_0, a_1, a_2, \dots, a_n$ kallas polynomets *koefficienter*, och talet n kallas polynomets *grad*. Polynomets variabel kan naturligtvis betecknas med någon annan bokstav än x .

Exempel 10.6.

- a) Ett polynom av graden noll är av formen

$$a_0, \quad (a_0 \neq 0)$$

dvs. är helt enkelt ett tal, eller som man brukar säga, en konstant.

- b) Ett polynom av graden 1 (eller av första graden) är av formen

$$a_0 + a_1x \quad (a_1 \neq 0).$$

Exempel på sådana polynom är

$$2 - 3x, \quad 5s + 7, \quad x, \quad -7x, \quad \frac{2}{3} - \frac{3x}{4}, \quad 6 + 5t.$$

- c) Ett polynom av graden 2 (eller av andra graden) är av formen

$$a_0 + a_1x + a_2x^2 \quad (a_2 \neq 0).$$

Exempel på sådana polynom är

$$5 - 3x + 2x^2, \quad x^2 - x + 1, \quad x^2, \quad z^2 + \frac{1}{4}, \quad -150t^2.$$

- d) Ett polynom av graden 3 (eller av tredje graden) är av formen

$$a_0 + a_1x + a_2x^2 + a_3x^3 \quad (a_3 \neq 0).$$

- e) Polynomet $2x^4 + 3x^3 - 11$ är av fjärde graden. Dess värde för $x = -2$ är

$$2 \cdot (-2)^4 + 3 \cdot (-2)^3 - 11 = 2 \cdot 16 + 3 \cdot (-8) - 11 = -3.$$

□

Ett polynom i x betecknas ofta med symbolen $f(x)$. Funktionen

$$x \mapsto f(x)$$

med mängden av alla reella tal som definitionsmängd kallas en polynomfunktion. Om $f(x)$ är av graden 0 eller 1, så är funktionen av formen

$$x \mapsto a_0 \quad \text{resp.} \quad x \mapsto a_0 + a_1x.$$

En polynomfunktion av graden 0 eller 1 är alltså detsammans som en linjär funktion. I de följande kapitlen skall vi studera polynomfunktioner av andra graden och av högre grad.

10.6 Polynomfunktioner av andra graden

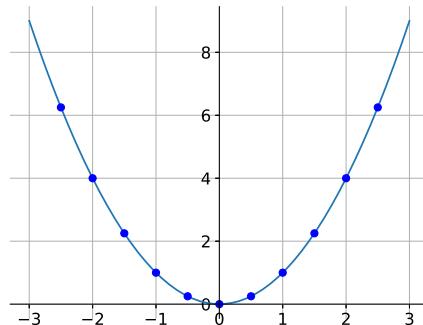
En polynomfunktion av andra graden är av formen

$$x \mapsto a_0 + a_1x + a_2x^2 \quad (a_2 \neq 0).$$

Den enklaste funktionen av detta slag är funktionen

$$x \mapsto x^2.$$

Funktionen, tillsammans med några beräknade talpar, visas i figur 10.5.



Figur 10.5: Funktionskurvan eller grafen kallas en parabel.

Punkterna ligger utefter en jämn kurva, och man kan visa att funktionens graf är denna kurva. Observera att kurvan är symmetrisk med avseende på y -axeln och att den inte har någon spets i origo, utan att den där är vackert rundad och tangerar x -axeln. Kurvan kallas en *parabel*. Den kan beskrivas som mängden av alla punkter, vilkas koordinater (x, y) satisfierar ekvationen $y = x^2$. Kurvan brukar därför kort kallas parabeln $y = x^2$. \square

Om sambandet mellan x och y är

$$y = kx^2,$$

där k är en konstant, så sägs y vara *proportionell mot kvadraten på x* , och k kallas proportionalfaktaorn.

Exempel 10.7. Vid fritt fall är fallsträckan s proportionell mot kvadraten på tiden t , mätt från rörelsens början. Sambandet mellan s och t är

$$s = k t^2.$$

En sten faller 78.5 m under de första 4 sekunderna av sin rörelse. Vi har då

$$78.5 = k \cdot 4^2$$

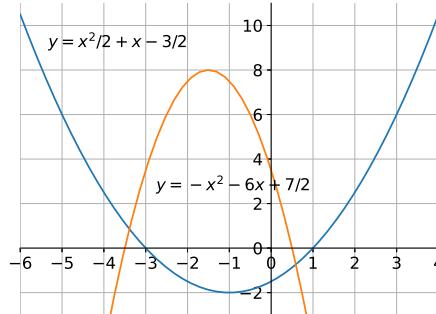
dvs.

$$k = \frac{78.5}{16} \approx 4.91.$$

En person släpper en sten från en bro. Stenen når vattenytan efter 2.4 s. Fallsträckan i meter är

$$s = 4.91 \cdot 2.4^2 \approx 28.82. \quad \square$$

Man kan visa, att grafen till en godtycklig polynomfunktion av andra graden kan erhållas ur parabeln $y = x^2$ genom en parallellförflyttning, eventuellt kombinerad med en spegelvälvning och en förstoring eller förminskning. Grafen är därför alltid en parabel. Se figuren 10.6 nedan.



Figur 10.6: Graferna till godtyckliga polynomfunktioner av andra graden är parabler.

Varje andragradspolynom $a_2x^2 + a_1x + a_0$ kan skrivas på formen

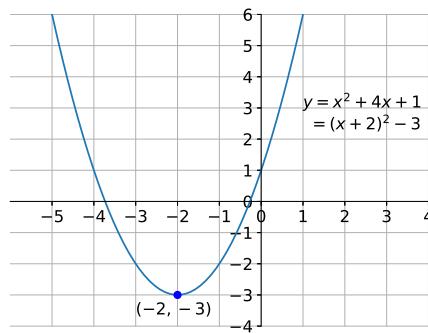
$$a_2(x - b)^2 + c.$$

Denna omskrivning kallas *kvadratkomplettering*.

Exempel 10.8. Polynomet $x^2 + 4x + 1$ kan skrivas

$$x^2 + 4x + 4 - 4 + 1 = (x + 2)^2 - 3.$$

Härav ser man att polynomets värde för $x = -2$ är -3 och att detta är det minsta värde polynomet kan ha. Ty termen $(x+2)^2$ i högra ledet ovan kan ju inte vara negativ. Funktionens graf har därför en *minimipunkt* (lägsta punkt) i $(-2, -3)$ se figur 10.7.



Figur 10.7: Vi kan bestämma minpunkten till parabeln genom att kvadratkomplettera.

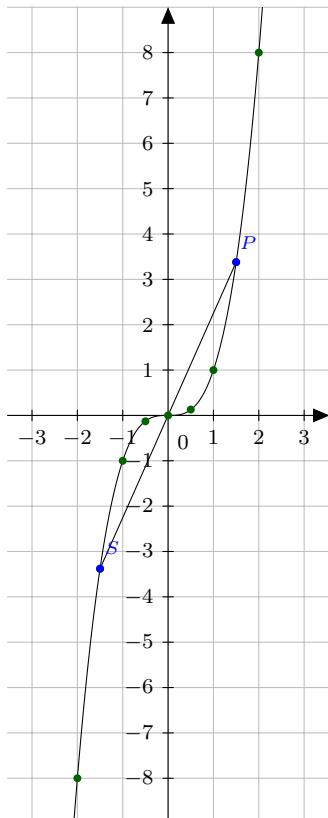
10.7 Polynomfunktioner av tredje graden

En polynomfunktion av tredje graden är av formen

$$x \mapsto a_0 + a_1x + a_2x^2 + a_3x^3 \quad (a_3 \neq 0).$$

Den enklaste funktionen av detta slag är

$$x \mapsto x^3.$$



I tabellen nedan har vi räknat ut några av funktionens talpar, och i figuren har motsvarande punkter markerats i ett koordinatsystem.

x	x^3
0	0
0.5	0.125
1	1
1.5	3.375
2	8
-0.5	-0.125
-1	-1
-1.5	-3.375
-2	-8

Funktionens graf är kurvan i figuren. Kurvan är symmetrisk med avseende på origo. Det betyder att för varje punkt P på kurvan gäller att punktens spegelbild S i origo också ligger på kurvan.

Observera att kurvan tangerar x -axeln i origo. Den skär alltså inte snett igenom x -axeln där.

Om sambandet mellan x och y är

$$y = kx^3.$$

där k är en konstant, så sägs y vara *proportionell mot kuben på* x , och k kallas proportionalitetsfaktorn.

Exempel 10.9. Vikten av en kula av ett visst material är proportionell mot kuben på radien (varför?). Om vikten är m och radien r , så är alltså

$$m = kr^3,$$

där k är en konstant. Två kulor av samma material har radierna 1.5 cm och 2.0 cm. Den mindre kulan väger 54 g. För att ta reda på vad den större kulan väger börjar vi med att beräkna proportionalitetsfaktorn

$$k = \frac{m}{r} = \frac{54}{1.5^3} = 16.$$

Den större kulan väger (i gram)

$$m = 16 \cdot 2.0^3 = 128.$$

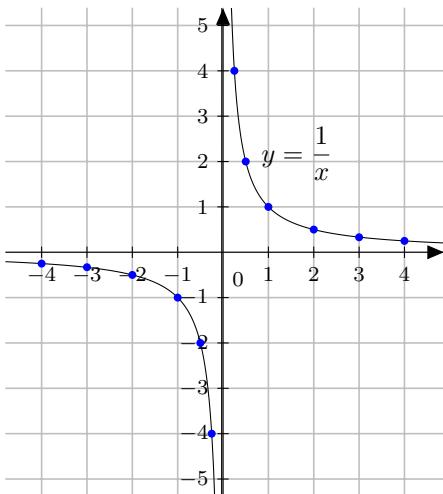
□

10.8 Rationella funktioner

Vi ritar grafen till funktionen

$$x \mapsto \frac{1}{x}.$$

I tabellen har vi räknat ut några av funktionens talpar, och motsvarande punkter har markerats i koordinatsystemet. Funktionens graf är den kurva som vi ritat genom punkterna.



x	$1/x$	y	$1/x$
1	1	-1	-1
2	1/2	-2	-1/2
3	1/3	-3	-1/3
4	1/4	-4	-1/4
1/2	2	-1/2	-2
1/4	4	-1/4	-4

Kurvan $y = \frac{1}{x}$ är symmetrisk med avseende på origo.

Om en punkt rör sig på kurvan så, att den avlägsnar sig från origo, så kommer punkten att alltmer nära sig antingen x -axeln eller y -axeln. Dessa linjer säges vara *asymptoter* till kurvan. Att y -axeln är en asymptot följer därav, att ju närmare noll man väljer x , desto större blir funktionsvärdet.

Så t.ex. ger $x = 0.001$ funktionsvärdet $\frac{1}{0.001} = 1\ 000$, och $x = -0.001$ ger funktionsvärdet $-1\ 000$.

Om y är proportionell mot $\frac{1}{x}$, dvs. $y = k \cdot \frac{1}{x}$ eller

$$y = \frac{k}{x}$$

så säges y vara *omvänt proportionell* mot x .

Exempel 10.10. Våglängden λ hos en radiovåg är omvänt proportionell mot vågens frekvens f . Om λ mäts i meter och f i Hertz (1 Hertz = 1 svängning per sekund), så blir proportionalitetsfaktorn $3 \cdot 10^8$ m/s (= ljusets fart). Sambandet mellan λ och f är

$$\lambda = 3 \cdot 10^8 \cdot \frac{1}{f}.$$

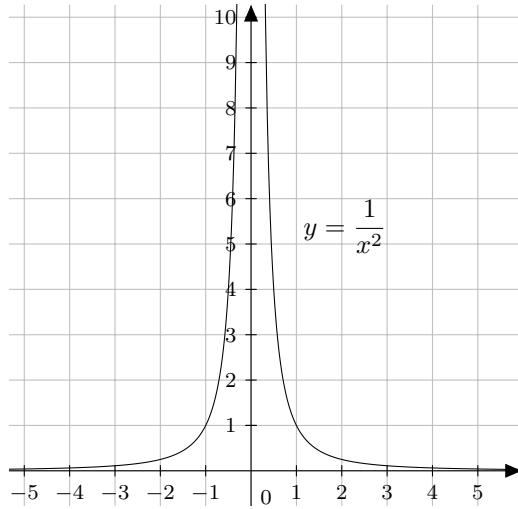
Våglängden då frekvensen är $5 \cdot 10^6$ Hz ges (i meter) av

$$\lambda = 3 \cdot 10^8 \cdot \frac{1}{5 \cdot 10^6} = 60.$$

□

Exempel 10.11. Vi ritar grafen till funktionen

$$x \mapsto \frac{1}{x^2}.$$



x	$1/x^2$
± 1	1
± 2	$1/4 = 0.25$
± 3	$1/9 \approx 0.11$
± 0.5	4
± 0.1	100
± 0.01	10 000

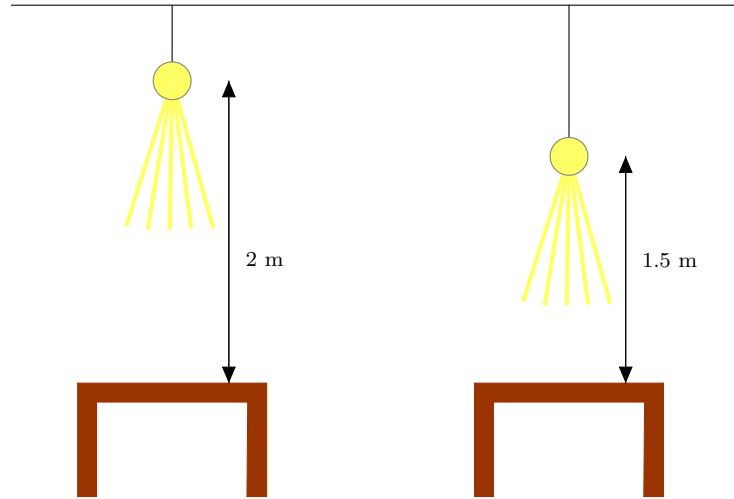
Grafen är symmetrisk med avseende på y -axeln, ty funktionsvärdena för x och $-x$ är lika. (För t.ex. $x = 3$ är funktionsvärdet $1/3^2 = 1/9$, och för $x = -3$ är funktionsvärdet $1/(-3)^2$, dvs. $1/9$). Grafen har x -axeln och y -axeln till asymptoter. □

Om y är proportionell mot $\frac{1}{x^2}$, dvs. $y = k \cdot \frac{1}{x^2}$ eller

$$y = \frac{k}{x^2}$$

så säges y vara *omvänt proportionell mot kvadraten på x* .

Exempel 10.12. En lampa hänger 2 m över ett bord. Hur många procent bättre blir belysningen på bordet rakt under lampan, om denna sänks 0.5 m? Belysningen är omvänt proportionell mot kvadraten på lampans höjd över bordet, se figur 10.8.



Figur 10.8: *Belysningen på bordet är omvänt proportionell mot lampans höjd x över bordet.*

Om belysningen betecknas B och lampans höjd x , så är $B = k/x^2$. För $x = 2$ resp. 1.5 är belysningen

$$B_1 = \frac{k}{2^2} \quad \text{resp.} \quad B_2 = \frac{k}{1.5^2}.$$

Kvoten B_2/B_1 blir nu

$$\frac{B_2}{B_1} = \frac{\frac{k}{1.5^2}}{\frac{k}{2^2}} = \frac{2^2}{1.5^2} \approx 1.77.$$

Belysningen blir alltså 77 % starkare om vi sänker lampan 0.5 m. \square

Funktionerna i exemplen ovan är exempel på *rationella funktioner*. En sådan funktion är av formen

$$x \mapsto \frac{f(x)}{g(x)},$$

där $f(x)$ och $g(x)$ är polynom. Funktionen är definierad för alla de reella tal x för vilka $g(x) \neq 0$. (För ett x -värde som gör $g(x) = 0$ är kvoten $f(x)/g(x)$ inte definierad.)

Om $g(x) = 1$, så blir funktionen

$$x \mapsto f(x)$$

dvs. en polynomfunktion. Polynomfunktionen är alltså speciella fall av rationella funktioner, och de kallas därför även *hela rationella funktioner*.

Varje rationell funktion är av formen

$$x \mapsto \frac{ax + b}{cx + d}$$

har en graf, som liknar kurvan $y = \frac{1}{x}$, fast den kan vara spegelvänt och ha annan storlek och annat läge.

Kapitel 11

Funktionsklasser: potens- och exponentialfunktioner

Potenser är ett viktigt begrepp som behövs för att beskriva exponentialfunktioner. Vi inför potenser och diskuteras deras egenskaper för att därefter införa potens- och exponentialfunktioner. Dessa används för att beskriva avtagande och växande kvantiteter i olika tillämpningar.

11.1 Potenser

En potens skrivs som

$$a^b,$$

där a kallas för *bas* och b för *exponent*.

11.2 Potenser med heltalsexponent

Vi har följande potensregler:

a) $4^3 \cdot 4^5 = (4 \cdot 4 \cdot 4) \cdot (4 \cdot 4 \cdot 4 \cdot 4 \cdot 4) = 4 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \cdot 4 = 4^8$

Observera att summan av faktorernas exponenter är lika med produktens exponent ($3+5=8$).

b) $5^2 \cdot 5^4 = 5^{2+4} = 5^6$

c) $10 \cdot 10^6 = 10^1 \cdot 10^6 = 10^{1+6} = 10^7$

Av exemplet ovan framgår följande regel:

En produkt av två potenser med samma bas kan skrivas som *en* potens med samma bas. Produktens exponent är summan av faktorernas exponenter. Om vi kallar basen a och exponenterna x och y , så kan regeln skrivas:

$$a^x \cdot a^y = a^{x+y}.$$

Exempel 11.1. Vad ska man mena med 2^0 ?

Om vi vill att multiplikationsregeln ovan ska gälla även för denna potens, så ska t.ex. följande likhet gälla:

$$2^3 \cdot 2^0 = 2^{3+0}$$

$$2^3 \cdot 2^0 = 2^3.$$

Härav ser vi att 2^0 måste vara lika med 1:

$$2^0 = 1.$$

□

På samma sätt som i exemplet kan vi visa att a^0 bör vara lika med 1 för varje bas a (som inte är noll). Man har därför infört följande *definition*: För varje $a \neq 0$ är

$$a^0 = 1.$$

Exempel 11.2. Vad ska man mena med 2^{-3} ?

Om vi vill att multiplikationsregeln ska gälla även för denna potens, så måste följande likhet gälla:

$$2^3 \cdot 2^{-3} = 2^{3+(-3)} = 2^0$$

$$2^3 \cdot 2^{-3} = 1.$$

Division av bågge lednen med 2^3 ger:

$$2^{-3} = \frac{1}{2^3} = \frac{1}{8}.$$

□

Genom samma resonemang som i detta exempel leds vi till följande *definition*:

Om $a \neq 0$ och n är ett positivt heltal, så gäller:

$$a^{-n} = \frac{1}{a^n}.$$

Exempel 11.3. Några potenser med negativa exponenter:

$$\text{a) } 4^{-1} = \frac{1}{4^1} = \frac{1}{4} = 0.25$$

$$\text{b) } 3^{-2} = \frac{1}{3^2} = \frac{1}{9} \approx 0.111$$

$$\text{c) } 10^{-6} = \frac{1}{10^6} = \frac{1}{1\,000\,000} = 0.000\,001$$

$$\text{d) } 0.5^{-3} = \frac{1}{0.5^3} = \frac{1}{0.125} = 8.$$

□

11.3 Potenser med rationella exponenter

Vad ska man mena med $3^{0.5}$?

Om vi vill att multiplikationsregeln för potenser fortfarande ska gälla, så måste

$$3^{0.5} \cdot 3^{0.5} = 3^{0.5+0.5} = 3^1 = 3$$

$$(3^{0.5})^2 = 3$$

$3^{0.5}$ är alltså ett tal vars kvadrat är 3. Det finns ett enda positivt tal som har denna egenskap, nämligen kvadratroten ur 3. Alltså bör gälla:

$$3^{0.5} = \sqrt{3} \approx 1.732.$$

Exemplet motiverar följande *definition*. För varje positivt tal a gäller:

$$a^{0.5} = a^{\frac{1}{2}} = \sqrt{a}.$$

Vi kan gå vidare och fråga vilket värde har potensen $8^{\frac{1}{3}}$?

Om multiplikationsregeln för potenser ska gälla, så måste

$$\left(8^{\frac{1}{3}}\right)^3 = 8^{\frac{1}{3}} \cdot 8^{\frac{1}{3}} \cdot 8^{\frac{1}{3}} = 8^{\frac{1}{3} + \frac{1}{3} + \frac{1}{3}} = 8^1 = 8$$

dvs. potensens kub är 8. Potensen är alltså 2

$$8^{\frac{1}{3}} = 2.$$

□

Antag att a är ett positivt tal och q ett positivt heltal. På samma sätt som ovan finner vi att

$$\left(a^{\frac{1}{q}}\right)^q = a,$$

dvs. att $a^{\frac{1}{q}}$ satisficerar ekvationen $x^q = a$. Då x växer från noll, så växer x^q och antar varje positivt värde exakt en gång. Ekvationen $x^q = a$ har alltså precis en positiv rot. Vi gör därför följande definition:

Potensen $a^{\frac{1}{q}}$ är den positiva roten till ekvationen $x^q = a$. Den positiva roten till ekvationen $x^q = a$ betecknas också $\sqrt[q]{a}$, vilket utläses 'q-te roten ur a '. Man kan alltså skriva:

$$a^{\frac{1}{q}} = \sqrt[q]{a}.$$

I fallet $q = 2$ skriver man som bekant inte ut tvåan. $\sqrt[3]{a}$ kallas också kubikroten ur a .

Exempel 11.4. Några förenklingar:

- a) $16^{\frac{1}{2}} = \sqrt{16} = 4$
- b) $8^{\frac{1}{3}} = \sqrt[3]{8} = 2$ (ty $2^3 = 8$)
- c) $7^{\frac{1}{3}} = \sqrt[3]{7} \approx 1.913$ (fås på dator)
- d) $625^{\frac{1}{4}} = \sqrt[4]{625} = 5$ (ty $5^4 = 625$)

□

Vilket värde har potensen $8^{\frac{2}{3}}$?

$$8^{\frac{2}{3}} = 8^{\frac{1}{3} + \frac{1}{3}} = 8^{\frac{1}{3}} \cdot 8^{\frac{1}{3}} = \left(8^{\frac{1}{3}}\right)^2 = 2^2 = 4.$$

På samma sätt får vi:

$$81^{\frac{3}{4}} = 81^{\frac{1}{4}} \cdot 81^{\frac{1}{4}} \cdot 81^{\frac{1}{4}} = \left(81^{\frac{1}{4}}\right)^3 = 3^3 = 27$$

$$9^{2.5} = 9^{\frac{5}{2}} = \left(9^{\frac{1}{2}}\right)^5 = 3^5 = 243.$$

□

Med ledning av räkningarna ovan gör vi följande *definition*. Om a är ett positivt tal, p ett helt tal och q ett positivt heltal så är

$$a^{\frac{p}{q}} = \left(a^{\frac{1}{q}}\right)^p.$$

Därmed är potensen a^x definierad för alla rationella exponenter x .

Följande räknelagar kan härledas ur denna definition:

$$(a^q)^{\frac{p}{q}} = a^p \quad a^{-\frac{p}{q}} = \frac{1}{a^{\frac{p}{q}}}.$$

11.4 Potenser med reella exponenter

Vi har definierat potensen a^x för godtycklig rationell exponent x . Definitionerna har valts så att räkneregeln

$$a^x \cdot a^y = a^{x+y}$$

ska gälla. Man kan definiera a^x för godtycklig positiv bas a och godtycklig reell exponent x så att räknelagen ovan blir giltig. Det visar sig att också följande räkneregler gäller:

$$\frac{a^x}{a^y} = a^{x-y}$$

$$(a^x)^y = a^{xy}$$

Exempel 11.5.

a) $\frac{2^7}{2^3} = 2^{7-3} = 2^4 = 16$

b) $(10^2)^3 = 10^{2 \cdot 3} = 10^6 = 1\,000\,000$

c) $\frac{10^2}{10^5} = 10^{2-5} = 10^{-3} = 0.001$

d) $\frac{10^{-2}}{10^5} = 10^{-2-5} = 10^{-7} = 0.000\,000\,1$

e) $\frac{10^2}{10^{-5}} = 10^{2-(-5)} = 10^7 = 10\,000\,000$

f) $\frac{10^{-2}}{10^{-5}} = 10^{-2-(-5)} = 10^3 = 1000.$ □

11.5 Beräkning av potenser på dator

Potenser beräknas enkelt med hjälp av Python.

Exempel 11.6. För att beräkna $2.8^{1.2}$ ger vi kommandot

```
2.8**1.2
```

Python svarar

```
3.4402471014328118
```

Talet $10^{-0.7}$ beräknas genom

```
10**(-0.7)
```

vilket ger

```
0.19952623149688797
```

□

11.6 Potensfunktioner

En potensfunktion ges av

$$y = x^a, \quad x > 0.$$

Funktionen är konstant ($=1$) om $a = 0$. Vidare är den strängt växande om $a > 0$ och strängt avtagande om $a < 0$.

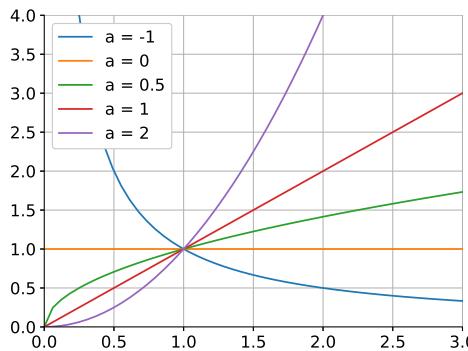
Exempel 11.7. Följande kommandon plottar potensfunktionen $y = x^a$ i intervallet $[0, 3]$ för några värden på a . Eftersom $y = x^a$ växer mot oändligheten för negativa a då x går mot noll behöver vi använda kommandot `axis` för att begränsa utsträckningen av axlarna.

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
x = np.linspace(0,3)
ax.plot(x,x**(-1),label='a = -1')
ax.plot(x,x**0,label='a = 0')
ax.plot(x,x**0.5,label='a = 0.5')
ax.plot(x,x**1,label='a = 1')
ax.plot(x,x**2,label='a = 2')
ax.axis([0,3,0,4])           # begränsar axlarna
ax.tick_params(labelsize=14)
ax.grid('on')
ax.legend(fontsize=14)

# dessa kommandon ger axlar genom origo
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_position('zero')
ax.spines['bottom'].set_position('zero')
```

Plotten visas i figur 11.1. □



Figur 11.1: Potensfunktioner $y = x^a$ för några olika värden på a . För $a < 0$ är funktionen strängt avtagande medan den är strängt växande för $a > 0$.

Exempel 11.8. Pendeln har genom tiderna använts både praktiskt och experimentellt. Den används i klockor och den användes tidigt för att undersöka kroppars fallrörelse. En pendel består av en masspunkt upphängd i ett snöre, se figur 11.2.



Figur 11.2: Pendel upphängd i en fast punkt.

För små vinklar kan man visa att svängningstiden T i sekunder ges av uttrycket

$$T = 2\pi \sqrt{\frac{l}{g}},$$

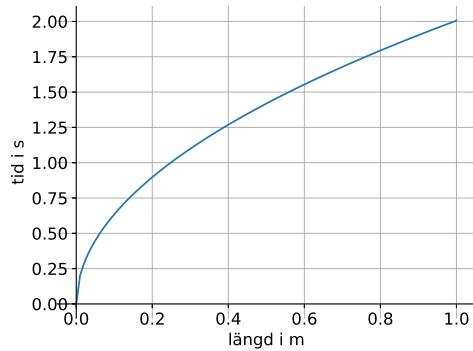
där l är pendelns längd i meter och $g = 9.81 \text{ m/s}^2$ tyngdaccelerationen. För att plotta T som funktion av längden l ger vi kommandona

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
g = 9.81                      # tyngdaccelerationen
l = np.linspace(0,1,100)         # längd l mellan 0 och 1
T = 2*np.pi*np.sqrt(l/g)        # svängningstid T
ax.plot(l,T)
ax.tick_params(labelsize=14)
ax.grid('on')
ax.set_xlabel('längd i m', fontsize=14)
ax.set_ylabel('tid i s', fontsize=14)

# dessa kommandon ger axlar genom origo
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_position('zero')
ax.spines['bottom'].set_position('zero')
```

Vi får plotten i figur 11.3. Från plotten kan vi grafiskt avläsa att en pendellängd $l \approx 0.25 \text{ m}$ ger en svängningstid $T = 1 \text{ s}$. \square



Figur 11.3: Svängningstiden T i s för pendeln beror av pendellängden l i m enligt $T = 2\pi\sqrt{\frac{l}{g}}$.

11.7 Exponentialfunktioner

En exponentialfunktion ges av

$$y = a^x, \quad a > 0,$$

där x kan anta alla reella tal. Om $a > 1$ är exponentialfunktionen strängt växande. Om $0 < a < 1$ är funktionen strängt avtagande.

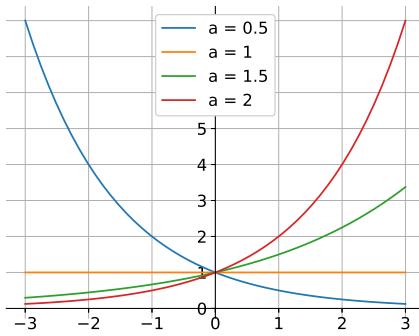
Exempel 11.9. Följande kommandon plottar exponentialfunktionen $y = a^x$ i intervallet $[-3, 3]$ för några värden på a . Vi ser att alla exponentialfunktioner går genom punkten $(0, 1)$.

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
x = np.linspace(-3,3)
ax.plot(x,0.5**x,label='a = 0.5')
ax.plot(x,1**x,label='a = 1')
ax.plot(x,1.5**x,label='a = 1.5')
ax.plot(x,2**x,label='a = 2')
ax.tick_params(labelsize=14)
ax.grid('on')
ax.legend(fontsize=14)

# dessa kommandon ger axlar genom origo
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_position('zero')
ax.spines['bottom'].set_position('zero')
```

Plotten visas i figur 11.4. □



Figur 11.4: Exponentialfunktioner $y = a^x$ för några olika värden på a . För $0 < a < 1$ är funktionen strängt avtagande medan den är strängt växande för $a > 1$.

Exempel 11.10. Antag att ett kapital K (kr) växer med 5% ränta. Efter 1 år har kapitalet vuxit till

$$K \cdot 1.05,$$

dvs. det har multiplicerats med *räntefaktorn* (eller förändringsfaktorn) 1.05. Under andra året räknas ränta på kapitalet $K \cdot 1.05$ (ränta på ränta). Efter 2 år har därför kapitalet vuxit till

$$K \cdot 1.05 \cdot 1.05 = K \cdot 1.05^2.$$

Det är lätt att se att kapitalet efter t år har vuxit till $K \cdot 1.05^t$. Kapitalets storlek vid olika tidpunkter ges alltså av funktionsvärdena till den växande exponentialfunktionen

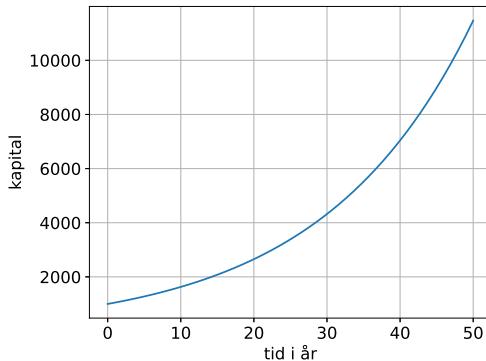
$$x \mapsto K \cdot 1.05^t.$$

Man säger därför att kapitalet *växer exponentiellt*. Följande Python kod ritar funktionens graf för $K = 1000$ och $0 \leq t \leq 50$.

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
t = np.linspace(0,50)
K = 1000
ax.plot(t,K*1.05**t) # exponentiellt växande funktion
ax.tick_params(labelsize=14)
ax.grid('on')
ax.set_xlabel('tid i år', fontsize=14)
ax.set_ylabel('kapital', fontsize=14)
```

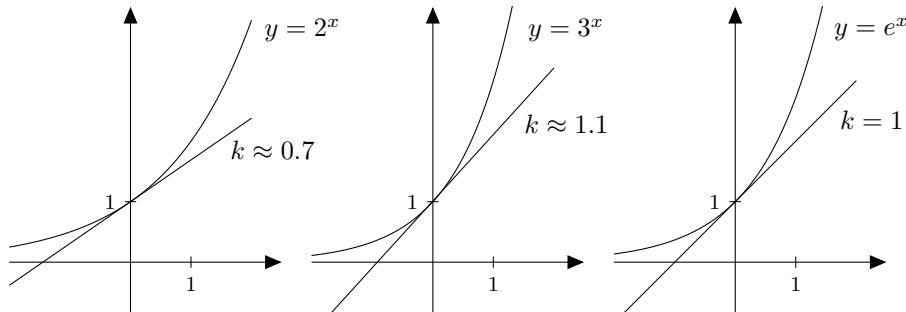
Plotten visas i figur 11.5. Från plotten kan vi grafiskt avläsa att det tar cirka 14 år för att kapitalet att fördubblas från 1000 kr till 2000 kr. \square



Figur 11.5: Funktionen $x \mapsto K \cdot 1.05^t$ beskriver hur ett kapital växer exponentiellt med tiden t .

11.8 Funktionen $y = e^x$

Figurerna nedan visar kurvorna $y = 2^x$ och $y = 3^x$. Till vardera kurvan har tangenten i punkten $(0, 1)$ dragits. Riktningskoefficienten för tangenten i figuren till vänster är ungefär $1/1.4 \approx 0.7$ och i mitten ungefär $1/0.9 \approx 1.1$. Det verkar troligt att det finns ett tal, vilket vi skall kalla e , mellan 2 och 3 sådant att kurvan $y = e^x$ i punkten $(0, 1)$ har en tangent med riktningskoefficienten 1, se figuren till höger.



Med dator har man beräknat e med hundratusentals decimaler. Med de 25 första decimalerna utskrivna är

$$e = 2.71828\ 18284\ 59045\ 23536\ 02874\dots$$

Talet e är av stor betydelse för matematiken och dess tillämpningar. Detsamma gäller exponentialfunktionen $x \mapsto e^x$. I Python ges e^x av kommandot `np.exp(x)`.

I många tillämpningar använder vi exponentialfunktioner av typen e^{kx} , där k är en konstant. Om $k > 0$ är exponentialfunktionen strängt växande och om $k < 0$ är exponentialfunktionen strängt avtagande.

För en exponentiellt växande eller avtagande funktion kan vi direkt få fram förändringsfaktorn. Vi har till exempel (jmf. räkneregel i avsnitt ??)

$$e^{0.1x} = (\underbrace{e^{0.1}}_{1.105})^x = 1.105^x,$$

vilket motsvarar en förändringsfaktor 1.105 (exponentiellt växande). På samma sätt är

$$e^{-0.1x} = (\underbrace{e^{-0.1}}_{0.905})^x = 0.905^x$$

vilket motsvarar en förändringsfaktor 0.905 (exponentiellt avtagande).

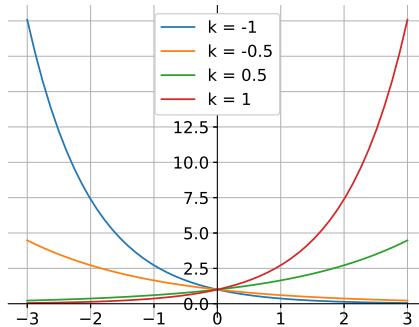
Exempel 11.11. Följande kommandon plottar exponentialfunktionen $y = e^{kx}$ i intervallet $[-3, 3]$ för några värden på k . Vi ser att alla exponentialfunktioner går genom punkten $(0, 1)$.

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
x = np.linspace(-3,3)
ax.plot(x,np.exp(-x),label='k = -1')
ax.plot(x,np.exp(-0.5*x),label='k = -0.5')
ax.plot(x,np.exp(0.5*x),label='k = 0.5')
ax.plot(x,np.exp(x),label='k = 1')
ax.tick_params(labelsize=14)
ax.grid('on')
ax.legend(fontsize=14)

# dessa kommandon ger axlar genom origo
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_position('zero')
ax.spines['bottom'].set_position('zero')
```

Plotten visas i figur 11.6. □



Figur 11.6: Exponentialfunktioner $y = e^{kx}$ för några olika värden på k . För negativa k är funktionen strängt avtagande medan den är strängt växande för positiva k .

11.9 Normalfördelning

Exponentialfunktionen i avsnitt 11.8 kommer till användning för att beskriva normalfördelningar. Variabler som förekommer i naturen, t.ex. medellängd, är i många fall normalfördelade och antar ofta värden som ligger nära medelvärdet och mycket sällan värden som har en stor avvikelse. Normalfördelningen beskrivs av en så kallad täthetsfunktion (eller frekvensfunktion)

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2},$$

där μ är fördelningens medelvärde och σ standardavvikelsen.

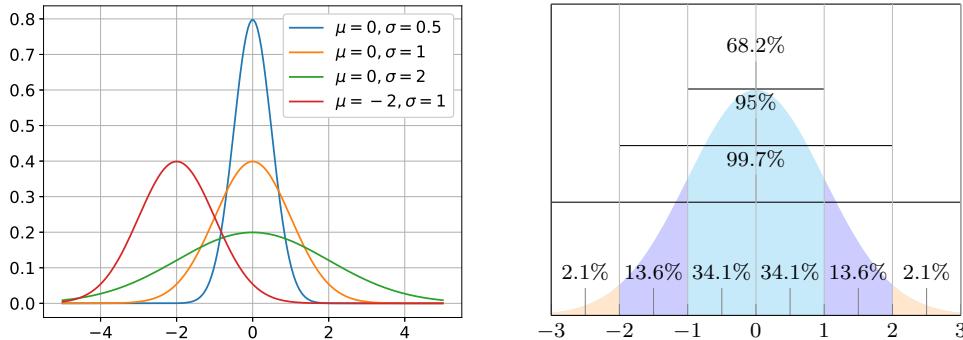
Exempel 11.12. Följande program plottar normalfördelningens täthetsfunktion i intervallet $[-5, 5]$ för lite olika värden på μ och σ .

```
import numpy as np
import matplotlib.pyplot as plt

# inför täthetsfunktionen som en lambda-funktion
f = lambda x, mu, sigma : 1/(sigma*np.sqrt(2*np.pi)) * \
    np.exp( -(x - mu)**2/(2*sigma**2) )

fig, ax = plt.subplots()
x = np.linspace(-5, 5, 200)
ax.plot(x, f(x, 0, 0.5), label=r'$\mu = 0, \sigma = 0.5$')
ax.plot(x, f(x, 0, 1), label=r'$\mu = 0, \sigma = 1$')
ax.plot(x, f(x, 0, 2), label=r'$\mu = 0, \sigma = 2$')
ax.plot(x, f(x, -2, 1), label=r'$\mu = -2, \sigma = 1$')
ax.grid('on')
ax.tick_params(labelsize=14)
ax.legend(fontsize=14)
```

Plotten visas till vänster figur 11.7. Vi ser att medelvärdet μ anger var fördelningen är centrerad. Standardavvikelsen σ anger hur bred fördelningen är, ju större σ desto bredare fördelning. För en variabel som är normalfördelad faller drygt 68 % av observationerna inom en standardavvikelse från medelvärdet, drygt 95 % inom två standardavvikeler från medelvärdet och drygt 99.7 % är inom tre standardavvikeler från medelvärdet. Detta är illustrerat till höger figur 11.7 \square



Figur 11.7: Vänster: normalfördelningens täthetsfunktion plottad för några kombinationer av medelvärdet μ och standardavvikelsen σ . Höger: för en variabel som är normalfördelad faller drygt 68 % av observationerna inom en standardavvikelse från medelvärdet, drygt 95 % inom två standardavvikeler från medelvärdet och drygt 99.7 % är inom tre standardavvikeler från medelvärdet.

11.10 Tillämpning: befolkningstillväxt

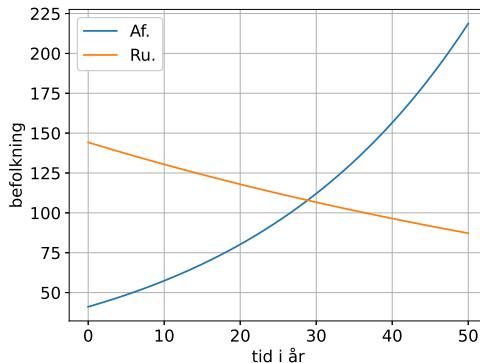
Under 2000 till 2022 har jordens folkmängd ökat med i medeltal 1.2% om året från 6.144 miljarder 2000 till 7.951 miljarder 2022. Detta betyder att folkmängden ökar exponentiellt. I vissa länder är ökningen mycket större: i Afghanistan har befolkningen ökat med i medeltal 3.4% om året från 19.5 miljoner 2000 till 41.1 miljoner 2022 medan i andra länder som Ryssland har befolkningen minskat med i medeltal -0.1% om året från 146.6 miljoner 2000 till 144.2 miljoner 2022. Källa: <https://wdi.worldbank.org/tables>. Följande Pythonprogram plottar befolkningsutvecklingen

i Afghanistan och Ryssland 50 år framåt under antagande om konstant befolkningsökning/minskning.

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
t = np.linspace(0,50)
ax.plot(t,41.1*1.034**t,label='Af.') # Afganistan faktor 1.034
ax.plot(t,144.2*0.99**t,label='Ru.') # Ryssland faktor 0.99
ax.tick_params(labelsize=14)
ax.grid('on')
ax.set_xlabel('tid i år',fontsize=14)
ax.set_ylabel('befolknings',fontsize=14)
ax.legend(fontsize=14)
```

Plotten visas i figur 11.8. Vi ser att befolkningen i Afghanistan kommer att överstiga befolkningen i Ryssland om lite mindre än 30 år. \square



Figur 11.8: Befolkningsutveckling i Afghanistan och Ryssland.

11.11 Tillämpning: radioaktivt nedfall

Vid kärnkraftskatastrofen i Tjernobyl 1986 blev Sverige påverkat då regnmoln band radioaktiva partiklar som drev med vinden och orsakade radioaktivt nedfall av främst ^{137}Cs (cesium) och ^{90}Sr (strontium). Mängden cesium och strontium, $N(t)$, avklingar exponentiellt enligt formeln

$$N(t) = N_0 e^{-\lambda t},$$

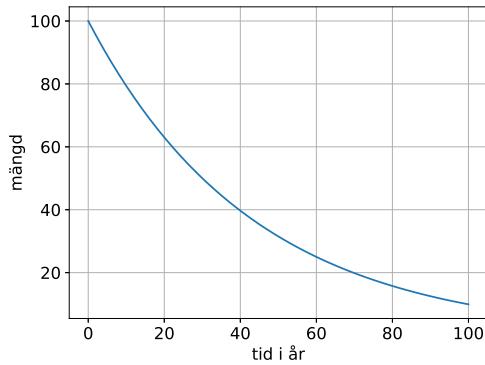
där N_0 är mängden av ämnet vid tiden $t = 0$, dvs. då vi börjar mäta, och λ är den så kallade sönderfallskonstanteren. För ^{137}Cs är $\lambda^{\text{Cs}} = 0.0231 \text{ år}^{-1}$ och för ^{90}Sr är $\lambda^{\text{Sr}} = 0.0240 \text{ år}^{-1}$. I programmet nedan tar vi mängden $N_0 = 100$ och plottar $N = N_0 e^{-\lambda t}$ för cesium då $0 \leq t \leq 100$.

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
t = np.linspace(0,100)
lambda_Cs = 0.0231
N0 = 100
```

```
ax.plot(t,N0*np.exp(-lambda_Cs*t))
ax.tick_params(labelsize=14)
ax.grid('on')
ax.set_xlabel('tid i år',fontsize=14)
ax.set_ylabel('mängd',fontsize=14)
```

Plotten visas i figur 11.9. Från plotten kan vi utläsa att mängden cesium halveras på 30 år. Man kan direkt beräkna halveringstiden från exponentialfunktionen med hjälp av den så kallade logaritmfunktionen. Detta ligger dock utanför denna kurs. \square



Figur 11.9: Den radioaktiva isotopen ^{137}Cs sönderfaller exponentiellt med en halveringstid på 30 år.

Kapitel 12

Funktionsklasser: periodiska funktioner

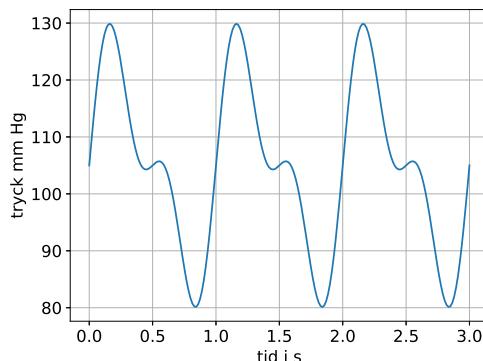
Periodiska fenomen är mycket vanligt förekommande i naturen. Det kan vara hjärtaktivitet, dagsljusvariationer, klimatvariationer, ändringar i djurpopulationer eller också mekaniska svängningar och vågfenomen som ljus och ljud. Vid modellering kan periodtiden ge värdefull information om vilken typ av process det är frågan om. Ofta, men inte alltid, kan man säga att periodtiden ökar med systemets storlek.

12.1 Periodiska funktioner

För att beskriva periodiska fenomen matematiskt använder man sig av *periodiska funktioner*. En funktion $f(t)$ kallas periodisk om funktionsvärdena upprepar sig med en viss period T_{period} , dvs om

$$f(t + T_{\text{period}}) = f(t) \quad \text{för alla } t.$$

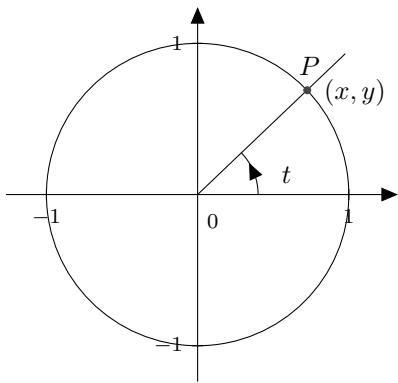
Exempel 12.1. I figur 12.1 har vi plottat blodtrycket i mm Hg hos en ung person som funktion av tiden t i sekunder. Blodtrycket beskrivs av en periodisk funktion $f(t)$ med $T_{\text{period}} \approx 1$. \square



Figur 12.1: Blodtrycket beskrivs av en periodisk funktion. Det maximala blodtrycket, vid den så kallade systoliska toppen, skall för en ung mänsklig ligga kring 130 mm Hg. Den lite lägre toppen kallas den distoliska toppen.

12.2 Cosinus- och sinusfunktioner

Periodiska funktioner uttrycks ofta i termer av cosinus- och sinusfunktioner. Dessa definieras utifrån *enhetscirkeln*.



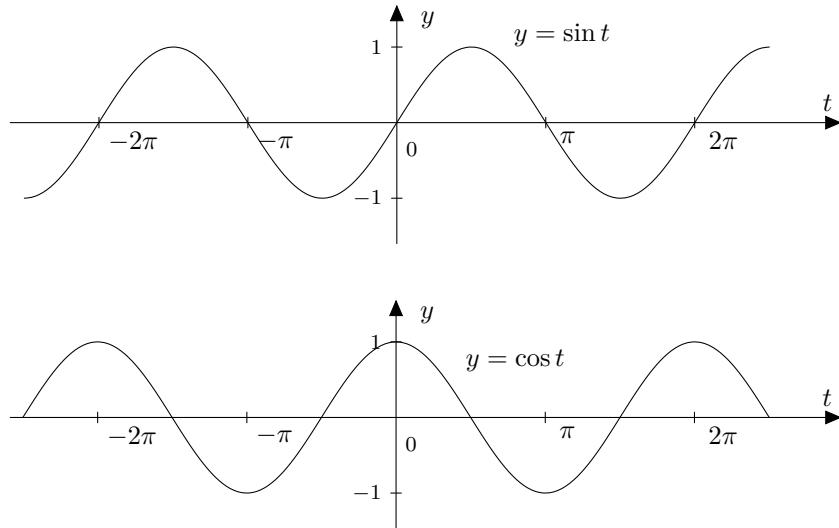
Cirkelbågens längd används som mått på vinkeln. Detta vinkelmått kallas *radianer*. Om cirkelbågen har längden t så är vinkeln t radianer:

grader	0	360	180	90
radianer	0	2π	π	$\frac{\pi}{2}$

$\cos t$ är x -koordinaten för punkten motsvarande vinkeln t i radianer.

$\sin t$ är y -koordinaten för punkten motsvarande vinkeln t i radianer.

Cosinus- och sinusfunktionerna får genom att ta x - och y -koordinaten för punkten på enhetscirkeln som funktion av vinkeln t i radianer. Funktionerna visas i figuren nedan



Funktionerna är periodiska med period 2π och

$$\cos(t + n \cdot 2\pi) = \cos(t), \quad n = \text{heltal}$$

$$\sin(t + n \cdot 2\pi) = \sin(t), \quad n = \text{heltal}.$$

I Python använder vi kommandona `np.sin(t)` och `np.cos(t)` för att beräkna cosinus- och sinusfunktionerna. Kommandona förutsätter att t ges i radianer.

12.3 Allmänna cosinus- och sinusfunktioner

Vi betraktar nu den mera allmänna funktionen

$$y(t) = y_0 + A \sin(\omega t + \delta).$$

Här är y_0 nollnivå, dvs. den nivå kring vilken svängningen ligger centrerad. Högt y_0 'hissar' upp hela kurvan medan lågt y_0 'hissar' ner hela kurvan. Den positiva parametern A är amplituden och anger storleken (omfånget) på svängningen. Högt värde på A , kraftig svängning. Den positiva parametern ω kallas vinkelfrekvensen och talar om hur snabb svängningen är. Högt värde på ω ger snabb svängning. Vinkelfrekvensen är relaterad till periodtiden T_{period} enligt

$$\omega = \frac{2\pi}{T_{\text{period}}} \quad \text{hög vinkelfrekvens } \omega \text{ ger kort periodtid } T_{\text{period}}.$$

Slutligen är δ den så kallade fasförskjutningen, som förskjuter hela kurvan i sidled. Positivt δ förskjuter kurvan till vänster och negativt δ förskjuter kurvan till höger.

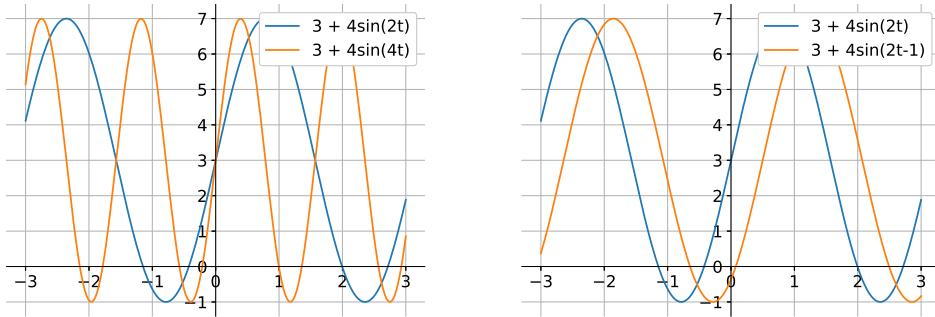
Exempel 12.1. Följande kommandon plottar $y(t) = 3 + 4 \sin(2t)$ och $y(t) = 3 + 4 \sin(4t)$ och $y(t) = 3 + 4 \sin(2t)$ och $y(t) = 3 + 4 \sin(2t - 1)$ över intervallet $[-3, 3]$.

```
import numpy as np
import matplotlib.pyplot as plt

# Plottar till vänster
fig, ax = plt.subplots()
t = np.linspace(-3,3,200)
ax.plot(t,3 + 4*np.sin(2*t),label = '3 + 4sin(2t)')
ax.plot(t,3 + 4*np.sin(4*t),label = '3 + 4sin(4t)')
ax.tick_params(labelsize=14)
ax.grid('on')
ax.legend(loc='upper right',fontsize=14)
# dessa kommandon ger axlar genom origo
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_position('zero')
ax.spines['bottom'].set_position('zero')

# plottar till höger
fig, ax = plt.subplots()
t = np.linspace(-3,3,200)
ax.plot(t,3 + 4*np.sin(2*t), label = '3 + 4sin(2t)')
ax.plot(t,3 + 4*np.sin(2*t-1),label = '3 + 4sin(2t-1)')
ax.tick_params(labelsize=14)
ax.grid('on')
ax.legend(loc='upper right',fontsize=14)
# dessa kommandon ger axlar genom origo
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_position('zero')
ax.spines['bottom'].set_position('zero')
```

De två första plotarna visas till vänster i figur 12.2 och de två sista till höger i samma figur. \square



Figur 12.2: Vänster: två funktioner med olika vinkelfrekvens. Höger två funktioner med olika fasförskjutning. Negativ fas ger förskjutning till höger.

12.4 Addition av cosinus- och sinusfunktioner

Cosinus- och sinusfunktioner med samma vinkelfrekvens ω kan adderas och ger en ny sinusfunktion med samma vinkelfrekvens men med en fasförskjutning skild från noll. Vi har

$$A \sin(\omega t) + B \cos(\omega t) = C \sin(\omega t + \delta),$$

där $C = \sqrt{A^2 + B^2}$. Vinkelfrekvensen δ kan fås genom att lösa en ekvation, men vi går inte in på detta. Ibland är det fördelaktigt (enkelt) att arbeta med summan av två funktioner och ibland är det enklare att arbeta med en funktion, men med en fasförskjutning skild från noll.

Exempel 12.2. Följande Pythonkommandon plottar funktionen

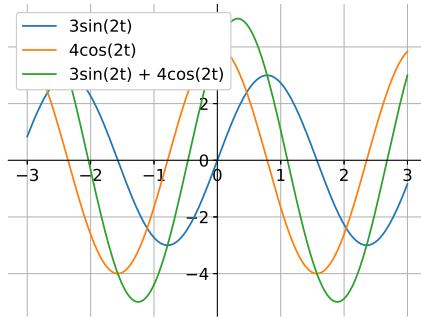
$$y(t) = 3 \sin(2\omega t) + 4 \cos(2\omega t)$$

i intervallet $[-3, 3]$.

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
t = np.linspace(-3,3,200)
ax.plot(t,3*np.sin(2*t),label = '3sin(2t)')
ax.plot(t,4*np.cos(2*t),label = '4cos(2t)')
ax.plot(t,3*np.sin(2*t)+4*np.cos(2*t),label = '3sin(2t) + 4cos(2t)')
ax.tick_params(labelsize=14)
ax.grid('on')
ax.legend(loc='upper left',fontsize=14)
# dessa kommandon ger axlar genom origo
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_position('zero')
ax.spines['bottom'].set_position('zero')
```

Plotten visas i figur 12.3. Vi ser att summan har en amplitud på $\sqrt{3^2 + 4^2} = 5$ och ligger förskjuten relativt de två grundfunktionerna $3 \sin(2\omega t)$ och $4 \cos(2\omega t)$. \square



Figur 12.3: Summan av en sinus- och en cosinusfunktion är en ny sinusfunktion med fasförskjutning skild från noll.

12.5 Tillämpning: medeltemperatur

Medeltemperaturen i Lund under perioden 1961–1990 visas i tabell 12.1.

Tabell 12.1: Medeltemperatur i Lund under perioden 1961–1990.

	Jan	Feb	Mar	Apr	Maj	Jun
	-0.6	-0.5	1.9	6.0	11.4	15.4
	Jul	Aug	Sep	Okt	Nov	Dec
	16.8	16.5	13.0	9.1	4.5	1.1

Följande Python program plottar de givna medeltemperaturerna, T , som funktion av tiden t (i månader) och använder ringar för datapunkterna. Vidare plottas en funktion på formen

$$T(t) = T_0 + A \sin(\omega t + \delta)$$

som beskriver data. Det är rimligt att ta T_0 som medelvärdet av temperaturerna under hela året. Amplituden kan vi få som det största temperaturvärdet minus det minsta och sedan dela med två. Eftersom perioden är $T_{\text{period}} = 12$ månader blir vinkelfrekvensen

$$\omega = \frac{2\pi}{T_{\text{period}}} = \frac{\pi}{6}.$$

För att bestämma fasförskjutningen har vi testat oss fram innan vi fick ett bra värde.

```
import numpy as np
import matplotlib.pyplot as plt

tdata = np.arange(1,13)
Tdata = np.array([-0.6,-0.5,1.9,6.0,11.4,15.4,16.8,16.5,13.0,9.1,4.5,1.1])

fig, ax = plt.subplots()
ax.plot(tdata,Tdata,'o')
t = np.linspace(1,12,100)
T0 = np.mean(Tdata)           # medelvärde av data
A = (16.8 - (-0.6))/2       # högsta minus lägsta delat med två
```

```

omega = np.pi/6           # en full svängning på 12 månader
delta = -2.2               # fått fram via testning
T = T0 + A*np.sin(omega*t + delta)
ax.plot(t,T)
ax.tick_params(labelsize=14)
ax.set_xlabel('tid i månader', fontsize=14)
ax.set_ylabel('medeltemperatur T', fontsize=14)
print('T0 = ', round(T0,2))
print('A = ', round(A,2))

```

Då vi kör programmet får vi

```

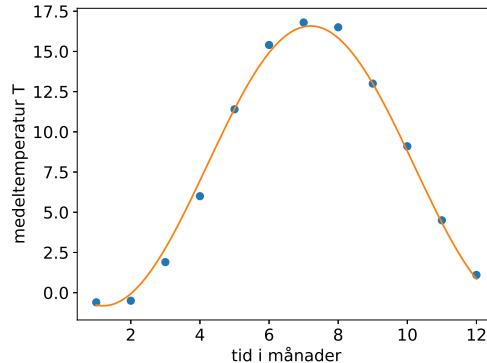
T0 =  7.88
A =  8.7

```

vilket ger att medeltemperaturen skulle kunna beskrivas av den allmänna sinusfunktionen

$$T(t) = 7.88 + 8.7 \sin\left(\frac{\pi}{6}t - 2.2\right).$$

Plotten visas i figur 12.4 och vi har en god överensstämmelse mellan funktion och uppmätta medeltemperaturer. \square



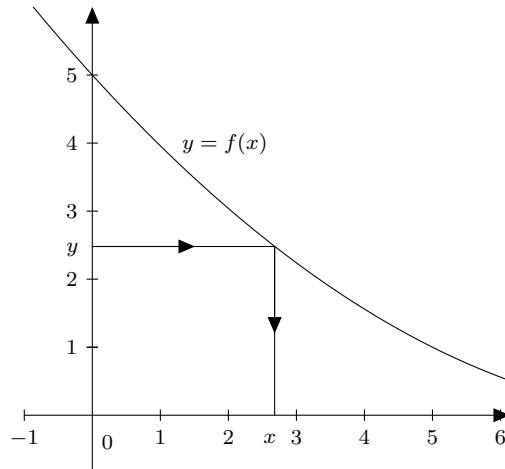
Figur 12.4: Medeltemperatur i Lund under perioden 1961–1990 tillsammans med en allmän sinusfunktion.

Kapitel 13

Ekvationer

13.1 Bestämning av x somhör till givet funktionsvärdet.

I kapitel 10 och 11 har vi diskuterat funktionsklasser som användas för att beskriva olika fenomen från befolkningsutveckling till pendelns svängningstid. En funktion $y = f(x)$ är en regel som givet x talar om vilket y vi får. I många fall vill vi göra det omvänta: givet y vill vi ta reda på det eller de x som ger detta värde. Situationen illustreras i figur 13.1.



Figur 13.1: Givet ett y vill vi bestämma det x som ger detta värde.

I matematiska termer är bestämningen av x detsamma som att lösa ekvationen

$$y = f(x) \quad y \text{ är givet och vi ska bestämma } x$$

för ett givet y . För enklare funktioner kan vi lösa ekvationerna för hand medan för mera komplicerade funktioner $f(x)$ måste vi lösa ekvationen med hjälp av inbyggda funktioner i Python.

I Python måste vi subtrahera y från båda sidor och baka in värdet i funktionen. Ekvationen vi ska lösa blir då

$$f(x) = 0.$$

Ett tal \bar{x} sådant att $f(\bar{x}) = 0$ kallas en rot till ekvationen eller ett nollställe till funktionen f . Fastän rot och nollställe egentligen har olika betydelse, betraktas de ofta som utbytbara.

13.2 Inbyggda funktioner för ekationslösning

Numerisk bestämning av rötter är en iterativ process. Med utgångspunkt i ett interval som omsluter roten eller en punkt nära roten beräknas en följd av tal som närmar sig den sökta roten. Processen avslutas då den konstruerade följen av tal, enligt något kriterium, kommer tillräckligt nära den sökta roten. I Python finns funktionerna som beräknar rötter i modulen `optimize`, vilken hör till `scipy`-paketet. Modulen importeras genom

```
import scipy.optimize as opt
```

Det finns flera användbara funktionerna för att bestämma rötter. Den enklaste är `root`, där man bara behöver ange funktionen $f(x)$ och ett startvärde nära roten. För att få fram startvärde är det bäst att först plotta funktionen.

Exempel 13.1. I en kemisk reaktion ges ämneskoncentrationen som funktion av tiden av

$$f(t) = 10e^{-3t} + 2e^{-5t}, \quad t \geq 0.$$

Koncentrationen minskar från ett maximalt värde 12. Vi vill bestämma tiden då koncentrationen har gått ner till 6. Detta är samma som att lösa ekvationen

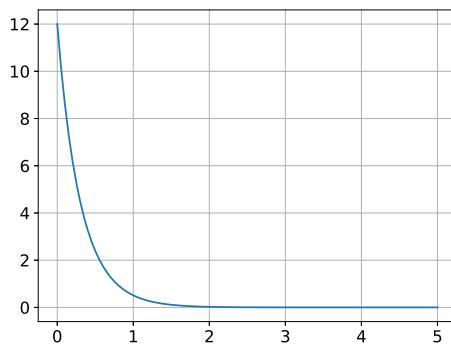
$$f(t) - 6 = 10e^{-3t} + 2e^{-5t} - 6 = 0.$$

Vi startar med att plotta $f(t)$ för att grovt lokalisera roten.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as opt

fig, ax = plt.subplots()
t = np.linspace(0, 5, 100)
f = lambda t : 10*np.exp(-3*t) + 2*np.exp(-5*t)
ax.plot(t,f(t))
ax.grid()
ax.tick_params(labelsize=14)
```

Detta ger plotten i figur 13.2. Från plotten ser vi att ämneskoncentrationen har gått ner till 6 vid $t \approx 0.2$.



Figur 13.2: Ämneskoncentrationen som funktion av tiden av. Vi vill bestämma tiden för vilken ämneskoncentrationen gått ner till hälften, dvs. 6.

För att bestämma ett mer noggrant värde, kallar vi på `root` med en ny funktion, där vi har subtraherat 6 från $f(t)$.

```
g = lambda t : f(t) - 6 # ny funk. g(t) där vi har subtraherat 6 från f(t)
r = opt.root(g,0.2)      # lös ekvationen g(t) = 0 med startvärde t = 0.2
                         # spara information om lösendet i variabeln r
print(r.message)         # r.message informerar om rot har hittats
print(r.x)                # r.x anger värdet på roten
```

Python svarar med att skriva ut

```
The solution converged.
[0.21132725]
```

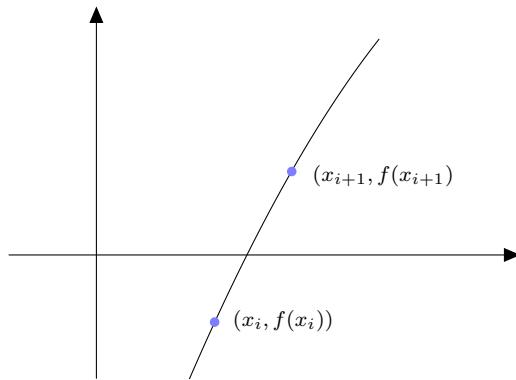
Den sista raden ger att roten är 0.21132725. \square

13.3 Egen funktion för ekvationslösning

Vi vill bestämma alla rötter till ekvationen

$$f(x) = 0$$

i ett interval $[a, b]$, där $f(x)$ är en kontinuerlig funktion. För den sakens skull låter vi $x_0, x_1, x_2, \dots, x_n$ vara punkter jämnt fördelade i (typiskt bestämda med hjälp av kommandot `np.linspace`). Vi looper igenom punkterna och varje gång $f(x_i)$ och $f(x_{i+1})$ har olika tecken har vi en rot någonstans mellan x_i och x_{i+1} , se figur 13.3. Vi approximerar denna rot med medelvärdet $(x_i + x_{i+1})/2$. Denna procedur ger alla rötterna i intervallet med en noggrannhet som bestäms av hur tätt punkterna ligger. Många punkter som ligger tätt ger hög noggrannhet på roten, medan få punkter som ligger på stora avstånd från varandra ger låg noggrannhet.



Figur 13.3: Om $f(x_i)$ och $f(x_{i+1})$ har olika tecken så har $f(x)$ en rot mellan x_i och x_{i+1} .

Funktionen `roots` nedan tar en vektor x med punkter i intervallet $[a, b]$ och en vektor f med motsvarande funktionsvärde som invariabler och returnerar en lista med rötter till ekvationen $f(x) = 0$.

```
def roots(x,f):
    """
        x vektor med x-värden i intervallet [a,b]
        f vektor med motsvarande funktionsvärden
        r är en lista med rötter till ekvationsn f(x) = 0
    """
    r = [] # börja med tom lista
    for i in range(len(x)-1):
        if f[i]*f[i+1] <= 0: # f[i], f[i+1] olika tecken om
            print('Rot',(x[i+1] + x[i])/2) # produkten negativ
            r.append((x[i+1] + x[i])/2) # om rot, addera till listan
    return r
```

Notera hur vi började med en tom lista `r = []` och sedan adderade element till listan med hjälp av metoden `append` när vi hittade nya rötter.

Exempel 13.2. Vi har funktionen

$$f(x) = 2x^3 - 8x$$

och ska bestämma alla rötter till ekvationen

$$f(x) = 2$$

i intervallet $[-2, 2]$. Det är detsamma som att bestämma rötterna till

$$f(x) - 2 = 2x^3 - 8x - 2 = 0.$$

Följande kommando plottar funktionen och kallar på funktionen `roots` som bestämmer rötter.

```
import numpy as np
import matplotlib.pyplot as plt

def roots(x,f):
    """
        x vektor med x-värden i intervallet [a,b]
        f vektor med motsvarande funktionsvärden
        r är en lista med rötter till ekvationsn f(x) = 0
    """
    r = [] # tom lista
    for i in range(len(x)-1):
        if f[i]*f[i+1] <= 0: # f[i], f[i+1] olika tecken om
            print('Rot',(x[i+1] + x[i])/2) # produkten negativ
            r.append((x[i+1] + x[i])/2) # om rot, addera till listan
    return r

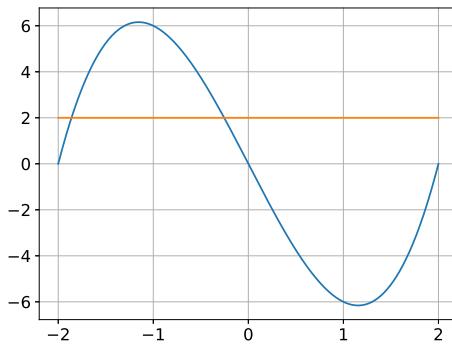
x = np.linspace(-2,2,1000) # 1000 punkter i intervall [-2,2]
f = 2*x**3 - 8*x # f vektor med funktionsvärden
fig, ax = plt.subplots()
ax.plot(x,f)
ax.plot(x,2 + 0*x) # plotta y = 2
ax.grid('on')
ax.tick_params(labelsize=14)

# kalla på roots med vektorn f - 2
r = roots(x,f - 2)
```

Då vi kör programmet får vi utskriften

```
Rot -1.8618618618618619
Rot -0.25225225225225223
```

Motsvarande plott visas i figur 13.4. Rötterna är inte lika noggranna som om vi hade använt den inbyggda funktionen `opt.root`, men i gengåld får vi alla rötter i intervallet utan att behöva köra den inbyggda funktionen upprepade gånger med olika startvärdet. \square



Figur 13.4: Funktionen $f(x) = 2x^3 - 8x$ i intervallet $[-2, 2]$. Den egenutvecklade bestämmer alla rötter till ekvationen $f(x) = 2$ i det angivna intervallet.

13.4 Tillämpning: jordens befolkning exponentiell modell

Jordens har för nuvarande en befolkning på cirka 8.098 miljarder invånare, se https://www.worldometers.info/world-population/#google_vignette. Vidare växer befolkningen med 0.9 % per år. Detta ger att befolkningen, $y(t)$, beskrivas av

$$f(t) = 8.099 \cdot 1.009^t,$$

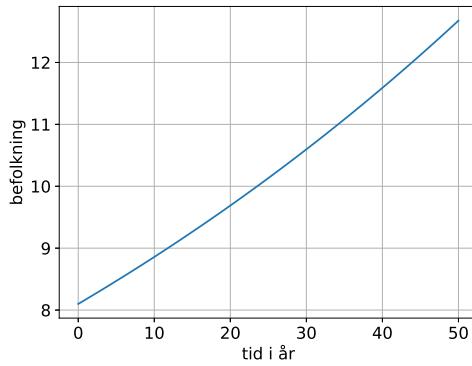
där vi räknar $t = 0$ från dagens datum. Bestäm hur många år det tar innan befolkningen går över 10 miljarder.

Vi börjar med att plotta befolkningsutvecklingen.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as opt

fig, ax = plt.subplots()
t = np.linspace(0, 50, 100)
f = lambda t : 8.099*1.009**t
ax.plot(t,f(t))
ax.grid()
ax.tick_params(labelsize=14)
```

Detta ger plotten i figur 13.2. Vi ser att befolkning är kommer upp i 10 miljarder efter ungefär 23 år.



Figur 13.5: Jordens befolkning som funktion av t . Vi vill bestämma tiden när befolkningen har kommit upp till 10 miljarder.

För att bestämma ett mer noggrant värde, kallar vi på `root` med en ny funktion $g(t)$, där vi har subtraherat 10 från $f(t)$.

```

g = lambda t : f(t) - 10    # ny funk. g(t) där vi har subtraherat 10 från f(t)
r = opt.root(g,23)           # lös ekvationen g(t) = 0 med startvärde t = 23
                             # r ger en massa information om lösandet
print(r.message)            # r.message informerar om rot har hittats
print(r.x)                  # r.x anger värdet på roten

```

Utskriften blir

```

The solution converged.
[23.53243101]

```

Funktionen hittade en rot. Från den sista raden kan vi utläsa att roten är 23.53243101.

Kapitel 14

Minstakvadratanpassningar

I det här kapitlet ska vi se hur vi kan hantera experimentell data. Ibland har vi, till exempel genom någon bakomliggande teori, tillgång till en modelfunktion som beskriver data. Vår uppgift är då att bestämma modellparametrarna så att modelfunktionen ansluter så bra som möjligt till data. Detta görs ofta via så kallade minstakvadratanpassningar (regression), och Python har flera inbyggda funktioner för detta ändamål.

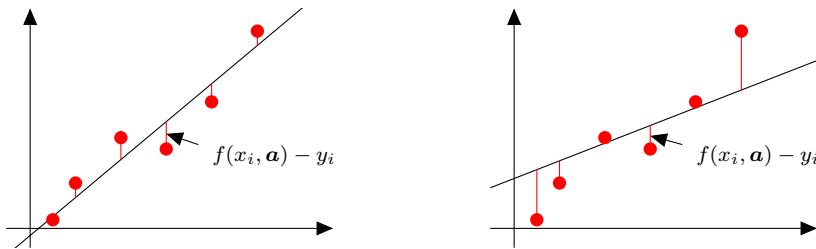
14.1 Minstakvadratanpassningar

Vid modellering är det viktigt att kunna anpassa modelfunktioner till experimentell data behäftad med osäkerhet. I det allmänna fallet har vi en modelfunktion $y = f(x, \mathbf{a})$, som beror av parameterarna $\mathbf{a} = (a_0, a_1, \dots, a_{m-1})$. Vi vill bestämma parametrarna så att modelfunktionen, i något mening, ansluter så väl som möjligt till ett antal datapunkter (x_i, y_i) , $i = 0, 1, \dots, n - 1$. Normalt sett är antalet datapunkter mycket större än antalet parametrar. I minstakvadratmetoden beräknar man summan av de kvadrerade avstånden i y -led mellan modelfunktionen $y = f(x, \mathbf{a})$ och datapunkterna

$$\chi^2 = \sum_{i=0}^{n-1} \underbrace{(f(x_i, \mathbf{a}) - y_i)^2}_{\begin{array}{l} \text{kvadrerat avstånd i } y\text{-led} \\ \text{mellan modelfunktionen} \\ \text{i } x_i \text{ och datapunkten } y_i \end{array}},$$

och bestämmer a_0, a_1, \dots, a_{m-1} så att felsumman χ^2 minimeras.

Metoden illustreras i figur 14.1, där vi vill anpassa en linjär modell (räta linje) $f(x, a_0, a_1) = a_0 + a_1 x$ till ett antal datapunkter. Avstånden i y -led, $f(x_i, a_0, a_1) - y_i$, mellan modelfunktionen och data är markerade med vertikala linjer. Varje avstånd kvadreras och sedan summeras alla kvadrerade avstånd för att ge χ^2 .



Figur 14.1: Avstånd i y -led, $f(x_i, a_0, a_1) - y_i$, mellan modellfunktionen och datapunkterna visas som vertikala streck. Felfunktionen χ^2 ges som summan av alla kvadrerade avstånd. De parametrar, a_0 och a_1 , som minimerar χ^2 ger den bästa anpassningen. Modellfunktionen till vänster har ett lägre χ^2 än modellfunktionen till höger och ger en bättre anpassning.

De parametrar, a_0 och a_1 , som minimerar χ^2 ger den bästa anpassningen. Parametrarna a_0 och a_1 för modellfunktionen till vänster i figuren ger ett lägre värde på χ^2 , och därmed en bättre anpassning, än parametrarna a_0 och a_1 för modellfunktionen till höger.

Vi har ofta någon teori, fastän begränsad och ofullständig, som förklrar data. Denna teori bestämmer vilken modellfunktion vi ska anpassa. Om det inte finns någon underliggande teori, kan målet vara att ta fram en lämplig modellfunktion. I detta fall kan vi behöva testa och utvärdera flera olika modeller som förklrar och beskriver data.

14.2 Polynom som modellfunktioner

Ofta använder man polynom som modellfunktioner, och i Python kan vi använda kommandot `polyfit`. För att få tillgång till kommandot börjar vi med att importera relevant modul från `numpy` på följande sätt

```
import numpy.polynomial.polynomial as pol
```

Kommandot beskrivs nedan.

`pol.polyfit(x,y,m)` ger polynomet av grad m som ger bästa anpassning till $(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})$ i minstkvadratmening.

Exempel 14.1. Vi har följande datapunkter

x_i	0.0	1.0	2.0	3.0	4.0
y_i	0.01	0.91	2.02	3.12	4.15

(a) Koefficienterna för polynomet av grad 1 som ger den bästa anpassningen ges av

```
import numpy.polynomial.polynomial as pol
import numpy as np
xdata = np.array([0,1,2,3,4])
ydata = np.array([0.01,0.91,2.02,3.12,4.15])
p1 = pol.polyfit(xdata,ydata,1)
print(p1)
```

och Python svarar

[-0.056 1.049]

Första elementet ger konstanten och andra elementet ger koefficienten framför x . Polynomet har alltså formen

$$y = -0.056 + 1.049 x.$$

Kommandona nedan plottar datapunkterna och modellfunktionen i intervallet $[-1, 5]$. Vi har använt optionen $ms = 15$, där ms står för ”marker size”, för att öka storleken på markören för datapunkterna

```
import numpy as np
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.plot(xdata,ydata,'+',ms=15) # plotta data
x = np.linspace(-1,5)          # många x-värden, tät gridd, för modellfunk.
y = p1[0] + p1[1]*x           # plotta modellfunktionen
ax.set_xlabel('x',fontsize=14)
ax.set_ylabel('y',fontsize=14)
ax.tick_params(labelsize=14)
```

Plotten visas till vänster i figur 14.2.

(b) Koefficienterna för polynomet av grad 3 som ger den bästa anpassningen fås genom

```
p3 = pol.polyfit(xdata,ydata,3)
print(p3)
```

och Python svarar

```
[ 0.00771429  0.77690476  0.15785714 -0.02333333]
```

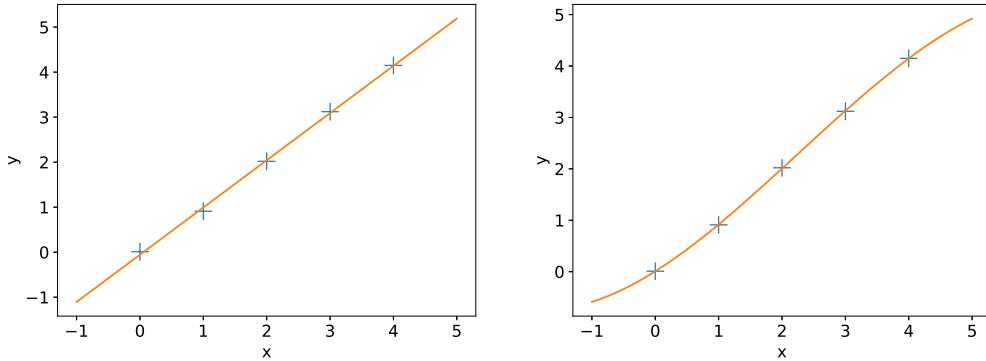
Polynomet har alltså formen

$$y = 0.0077 + 0.7769 x + 0.1579 x^2 - 0.0233 x^3.$$

Datapunkterna och modellfunktionen plottas genom

```
import numpy as np
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.plot(xdata,ydata,'+',ms=15) # plotta data
x = np.linspace(-1,5)          # många x-värden, tät gridd, för modellfunk.
y = p3[0] + p3[1]*x + p3[2]*x**2 + p3[3]*x**3
ax.plot(x,y)                  # plotta modellfunktionen
ax.set_xlabel('x',fontsize=14)
ax.set_ylabel('y',fontsize=14)
ax.tick_params(labelsize=14)
```

Vi får plotten till höger i figur 14.2. Man ska vara försiktig med anpassningar till polynom av höga gradtal. \square



Figur 14.2: Anpassade polynom av grad 1 och 3.

14.3 Allmänna modelfunktioner

För allmänna modelfunktioner använder vi kommandot `leastsq` för att göra minstakvadratanpassningen. För att få tillgång till kommandot måste vi importera modulen `optimize` från `scipy`. Detta görs genom

```
import scipy.optimize as opt
```

Användningen av funktionen `leastsq` beskrivs nedan.

<code>opt.leastsq(res,a0,(x,y))</code>	ger parametrarna till en allmän modelfunktion. <code>res</code> är funktionen som definierar residualen, dvs. skillnaden mellan funktionsvärdena och <code>y</code> -data, <code>a0</code> är en vektor med startvärdet för parametrarna. <code>x</code> och <code>y</code> är vektorer med datavärden.
--	---

Exempel 14.2. Vi har ett radioaktivt prov. Aktiviteten A som funktion av t (i minuter), ges nedan.

t_i	0.0	1.25	2.5	3.75	5.0	6.25	7.5
A_i	513	349	196	124	83	72	43
t_i	8.75	10.0	11.25	12.5	13.75	15.0	
A_i	38	23	22	12	5	10	

Aktiviteterna visas till vänster i figur 14.3. Givet vår bakgrund inom fysik tar vi en modelfunktion

$$A(t) = a_0 e^{-a_1 t},$$

där a_0 är aktiviteten vid $t = 0$ och a_1 är sönderfallskonstanten. Från plotten till vänster i figur 14.3 kan vi se att $a_0 = 500$ och $a_1 = 0.3$ är rimliga startvärden för parametrarna.

För att göra anpassningen definierar vi två funktioner, f och res , som ger modelfunktionen och residualen (skillnaden mellan funktionsvärdena och `y`-data). Efter funktionsdefinitionerna skriver vi in data och kallar på `leastsq`, som gör anpassningen. Givet parametrarna från `leastsq` plottar vi den anpassade modelfunktionen tillsammans med data. Programmet som gör anpassningen ges nedan

```

import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as opt

def f(a,t):                      # modellfunktion
    return a[0]*np.exp(-a[1]*t)

def res(a,tdata,Adata):           # residual
    return Adata - f(a,tdata)

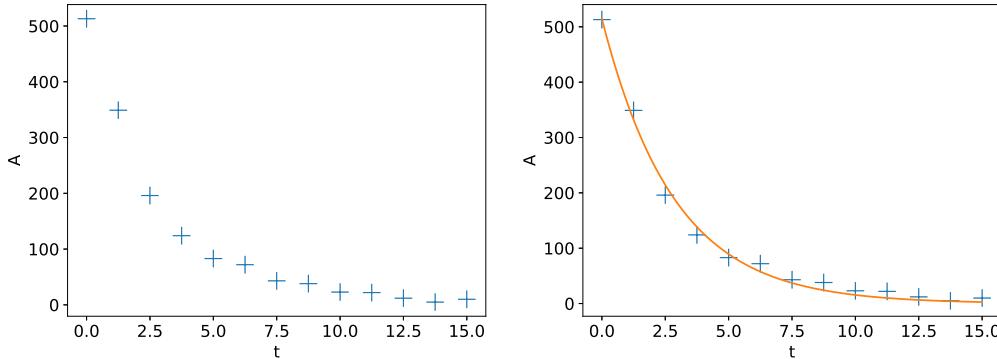
tdata = np.array([0,1.25,2.5,3.75,5,6.25,7.5,8.75,10,\n
                  11.25,12.5,13.75,15.])
Adata = np.array([513.0,349,196,124,83,72,43,38,23,22,\n
                  12.5,10.])
a0 = [500,0.3]                   # startgissning
a,q = opt.leastsq(res,a0,(tdata,Adata))
print('Parametrar:',a)
fig, ax = plt.subplots()
ax.plot(tdata,Adata,'+',ms=15)   # plotta data
t = np.linspace(0,20,200)        # många t-värden, tät gridd, för modellfunk.
A = f(a,t)                      # plotta modellfunktionen
ax.set_xlabel('t',fontsize=14)
ax.set_ylabel('A',fontsize=14)
ax.tick_params(labelsize=14)

```

Programmet skriver ut

Parametrar: [5.14583940e+02 3.50326928e-01]

vilket ger $f(t) = 514.6e^{-0.350t}$. Plotten visas till höger i figur 14.3. \square



Figur 14.3: Modellfunktion anpassad till aktivitetsdata.

Exempel 14.3. Vi ska nu gå tillbaka till data i avsnitt 1.4. I tabell 14.1 har vi data från ett experiment där ingenjörsstudenter vid Malmö universitet släppte en bordtennisboll från olika höjder och mätte motsvarande studshöjd i cm

Tabell 14.1: Studshöjd i cm som funktion av släpphöjd i cm.

släpp	0	40	80	120	160	200	250	300	400	500	600	700
studs	0	36	65	85	100	113	135	141	160	165	175	185

Under antagandet att studshöjden är proportionell mot släpphöjden för låga släpphöjder, men måste plana av mot ett konstant värde tog vi en modelfunktion av formen

$$h(x) = \frac{a_0 x}{1 + a_1 x}.$$

Hur ska vi få fram startvärde för a_0 och a_1 ? Vi kan testa oss fram och plotta modelfunktionen för ett antal kombinationer av a_0 och a_1 . Det går bra men är tidsödande. Vi kan istället försöka resonera oss fram. För små värden på släpphöjden har vi nästan ett linjärt samband, dvs.

$$h(x) \approx a_0 x$$

Från data kan vi grovt skatta a_0 som 0.8. För stora släpphöjder är

$$h(x) \approx \frac{a_0 x}{a_1 x} = \frac{a_0}{a_1}, \quad \text{studshöjd konstant för stora släpphöjder}$$

Vi skatta maximal studshöjd till 200 cm. Detta ger

$$200 = \frac{a_0}{a_1} \Leftrightarrow a_1 = \frac{a_0}{200} = 0.004.$$

Vi har alltså startvärde $a_0 = 0.8$ och $a_1 = 0.004$. Följande Pythonprogram

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as opt

xdata = np.array([0,40,80,120,160,200,250,300,400,500,600,700])
hdata = np.array([0,36,65,85,100,113,135,141,160,165,175,185])

def f(a,x):                      # modelfunktion
    return a[0]*x/(1 + a[1]*x)
def res(a,xdata,hdata):           # residual
    return hdata - f(a,xdata)

a0 = [0.8,0.004]                  # startgissning
a,q = opt.leastsq(res,a0,(xdata,hdata))
print('Parametrar:',a)
fig, ax = plt.subplots()
ax.plot(xdata,hdata,'+',ms=15)   # plotta data
x = np.linspace(0,1000,200)       # många x-värden, tät gridd för modelfunk.
h = f(a,x)                       # plotta modelfunktionen
ax.set_xlabel('x i cm',fontsize=14)
ax.set_ylabel('h i cm',fontsize=14)
ax.tick_params(labelsize=14)
```

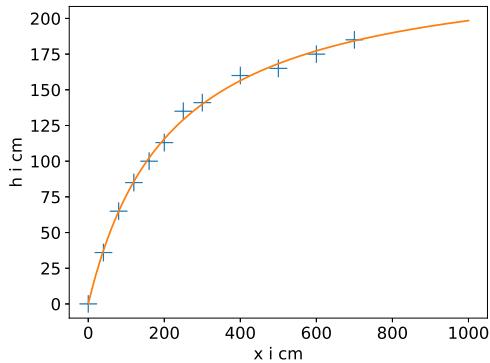
Programmet skriver ut

Parametrar: [1.10748208 0.00458031]

vilket ger

$$h(x) = \frac{1.107x}{1 + 0.00458x}.$$

Plotten visas i figur 14.4. Maximal studshöjd ges av $h_{\max} = a_0/a_1 = 242$ cm. □



Figur 14.4: Modellfunktion anpassad till aktivitetsdata.

14.4 Tillämpning: jordens befolkning logistisk modell

Jordens befolkning ökar ständigt, och tillförlitliga befolkningsuppgifter finns redovisade sedan 1951. Nedan visas ett utdrag från en befolkningstabellen från <https://www.worldometers.info/world-population/world-population-by-year/>.

World Population by Year

Year	World Population	Yearly Change	Net Change
2023	8,045,311,447	0.88 %	70,206,291
2022	7,975,105,156	0.83 %	65,810,005
2021	7,909,295,151	0.87 %	68,342,271
2020	7,840,952,880	0.98 %	76,001,848
2019	7,764,951,032	1.06 %	81,161,204
2018	7,683,789,828	1.10 %	83,967,424
2017	7,599,822,404	1.15 %	86,348,166
2016	7,513,474,238	1.17 %	86,876,701
2015	7,426,597,537	1.19 %	87,584,118

All befolkningsdata i tabellen, från 1951 fram till och med 2023 (73 år), finns samlade i textfilen **population.txt**, vilken kan laddas ner från Canvas. Givet dessa data är vår uppgift att göra en modell för hur befolkningen utvecklar sig fram till 2100. Det är enklast att välja tiden så att $t = 0$ motsvarar det år då vi börjar studera befolkningsutvecklingen, dvs. $t = 0$ motsvarar år 1951. År 2100 motsvaras då av $t = 149$.

Jordens resurser räcker inte för att föda en hur stor befolkning som helst, utan vi måste tänka oss att det finns en övre gräns för befolkningen. Denna övre gräns, K , kallas bärarkapaciteten, se också avsnitt 16.5.2. Vi söker en modell som startar med en given befolkning N_0 för $t = 0$, växer exponentiellt enligt $N_0 e^{rt}$ för små t , för att plana ut mot bärarkapaciteten K för stora värden på tiden t . En sådan modell ges av den logistiska funktionen

$$N(t) = \frac{K N_0 e^{rt}}{N_0 e^{rt} + (K - N_0)}.$$

För t nära noll är $e^{rt} \approx e^0 = 1$. Nämnden i den logistiska funktionen är då $N_0 e^{rt} + (K - N_0) \approx K$. Detta i sin tur ger att

$$N(t) \approx N_0 e^{rt}$$

för små värden på t . För stora värden på t är

$$N(t) \approx \frac{KN_0e^{rt}}{N_0e^{rt}} = K.$$

Allt verkar stämma. Vi låter nu $K = a_0$, $N_0 = a_1$ och $r = a_2$ och anpassar modellfunktionen till data med hjälp av minstakvadratmetoden. Detta göra med Pythonprogrammet nedan. Programmet plottar dessutom den anpassade modellfunktionen tillsammans med data. Som startvärde för parametrarna tar vi $K = 12$ (gissad bärarkapacitet), $N_0 = 2.54$, vilket är jordens befolkning år 1951 ($t = 0$) enligt data. Vidare tar vi $r = 0.02$, vilket ungefärlig motsvarar en ursprunglig förändringsfaktor 1.02 (se avsnitt 11.8).

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as opt

def f(a,t):                      # modellfunktion
    num   = a[0]*a[1]*np.exp(a[2]*t)
    denom = a[1]*np.exp(a[2]*t) + (a[0] - a[1])
    return num/denom

def res(a,tdata,Ndata):           # residual
    return Ndata - f(a,tdata)

Ndata = np.loadtxt("population.txt") # läs data från fil
tdata = np.arange(73)              # t = 0, år 1951, t = 72, år 2023

fig, ax = plt.subplots()          # plotta befolkningsdata 1951 till 2023
ax.plot(tdata,Ndata,'+')

a0 = [12,2.54,0.02]              # startgissning
a,q = opt.leastsq(res,a0,(tdata,Ndata))
print('Parametrar:',a)

t = np.linspace(0,149)            # tid fram till 2100, dvs t = 149
N = f(a,t)
ax.plot(t,N)                     # plotta modellfunktionen
ax.set_xlabel('t i år från 1951',fontsize=14)
ax.set_ylabel('befolknings miljarder',fontsize=14)
ax.tick_params(labelsize=14)
```

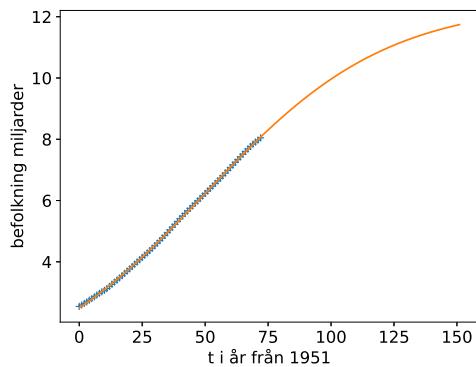
Då vi kör programmet får vi

Parametrar: [12.45928027 2.494143 0.02769494]

Modellfunktionen är alltså

$$N(t) = \frac{12.459 \cdot 2.494 e^{0.0277t}}{2.494 e^{0.0277t} + (12.459 - 2.494)}.$$

Fram till år 2100 kommer befolkningen enligt vår modell att växa från dagens 8.045 miljarder till cirka 11.8 miljarder. Enligt vår modell kommer jordens befolkning plana ut mot 12.4 miljarder mänskiskor. Dessa värden ligger väl i linje med mera avancerade modeller, där man försöker att skatta en mängd parametrar som på sikt kan tänkas påverka befolkningsutvecklingen. Befolkningsdata tillsammans med anpassad modellfunktion visas i figur 14.5.

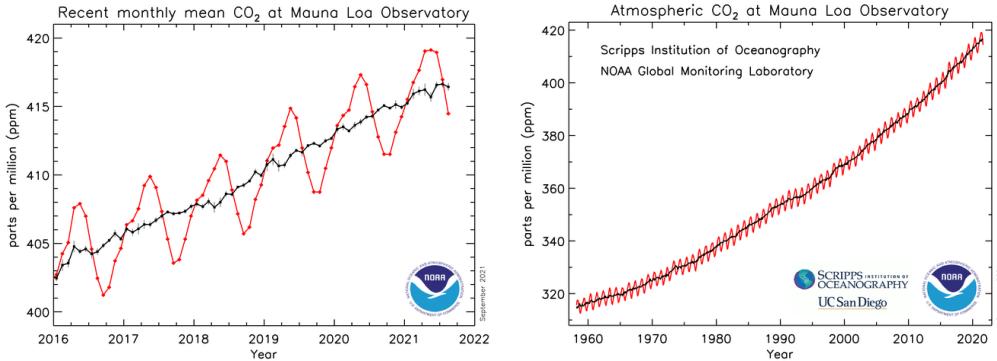


Figur 14.5: Jordens befolkning kan modelleras med en logistisk funktion.

14.5 Tillämpning: CO₂ i atmosfären

Jorden andas! Under våren och sommaren absorberar vegetationen på norra halvklotet CO₂ för att med hjälp av fotosyntesen bygga upp biomassa (löv, ved etc.). Absorptionen gör att CO₂-koncentrationen i atmosfären minskar. Under hösten och vintern faller löven av och multnar, CO₂ frigörs och koncentrationen i atmosfären ökar. Samma processer sker på södra halvklotet men förskjutna med ett halvår. Påverkan på CO₂-koncentrationen i atmosfären är dock mindre eftersom landmassan på södra halvklotet är betydligt mindre än landmassan på norra halvklotet. Jordens andning kan följas på satellitbilder av landmassan under ett år. 'Andningen' kan följas på videon nedan.

Utöver andningen påverkas CO₂-koncentrationen också av de mänskliga utsläppen, främst från förbränning av olja och kol, så att vi får en kraftigt uppåtgående trend. CO₂-koncentrationen mäts regelbundet vid Mauna Loa-observatoriet (se <https://gml.noaa.gov/ccgg/trends/>), och i figur 14.6 ser vi koncentrationen de senaste 5 åren och även koncentrationen sedan 1960. Vi ser ingen avmattning, utan koncentrationen stiger alarmerande fort.



Figur 14.6: CO_2 -koncentration i atmosfären. Förutom de naturliga årliga variationerna ser vi en nästan konstant ökning av koncentrationen beroende på människans utsläpp. Källa: <https://gml.noaa.gov/ccgg/trends/>.

I filen **co2full2010.txt**, vilken laddas ner från Canvas, har vi data över koncentrationer upptagna varje månad från januari 2010 till december 2020, dvs. 11 år av data. Filen har 8 kolonner och kolonnen med index 2 ger tiden (i år) för mätningen och kolonnen med index 3 ger koncentrationen i ppm. Vi ska anpassa en modellfunktion

$$f(t, \mathbf{a}) = a_0 + a_1 t + a_2 t^2 + a_3 \sin(kt) + a_4 \cos(kt) + a_5 \sin(2kt) + a_6 \cos(2kt),$$

där de tre första termerna beskriver ökningen av koncentrationen genom mänsklig påverkan och där de fyra sista termerna beskriver de naturliga årliga variationerna. k är vinkelfrekvensen för de årliga variationerna. Vi har en huvudsvängning (2π radianer) per år och då är $k = 2\pi$. Tar vi även hänsyn till svängningarna orsakade av vegetationen på södra halvklotet behöver vi också termer med vinkelfrekvensen $2k = 4\pi$. Programmet **co2.py** läser datafilen **co2full2010.txt**, anpassar modellfunktionen till data och plottar data tillsammans med den anpassade modellfunktionen. Parametrarna a_0 och a_1 har grovt skattats från data. Övriga parametrar har helt sonika satts till noll, anpassningen är inte känslig för dessa och konvergerar vilka värden vi än tar.

```

import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as opt

k = 2*np.pi
def f(a,t):                      # modellfunktion
    return a[0] + a[1]*t + a[2]*t**2 + \
           a[3]*np.sin(k*t) + a[4]*np.cos(k*t) + \
           a[5]*np.sin(2*k*t) + a[6]*np.cos(2*k*t)

def res(a,tdata,co2data):          # residual
    return co2data - f(a,tdata)

co2 = np.loadtxt('co2full2010.txt',usecols=(2,3))
tdata = co2[:,0]                  # kolonn 1 är tiden
co2data = co2[:,1]                # kolonn 2 är co2

a0 = [390,2.5,0,0,0,0,0]          # startgissning
a,q = opt.leastsq(res,a0,(tdata,co2data))
print('Parametrar:',a)
fig, ax = plt.subplots()
ax.plot(tdata,co2data,'+')      # plotta data

```

```
t = np.linspace(2010,2022,1000) # tät grid för modellfunktionen
y = f(a,t)
ax.plot(t,y) # plottar modellfunktionen
ax.set_xlabel('t', fontsize=14)
ax.set_ylabel('co2 in ppm', fontsize=14)
ax.tick_params(labelsize=14)
```

Då vi kör programmet får vi

```
Parametrar: [ 7.94417092e+04 -8.08718576e+01  2.06676522e-02  2.89159823e+00
 -8.55434022e-01 -5.83770219e-01  6.92137635e-01]
```

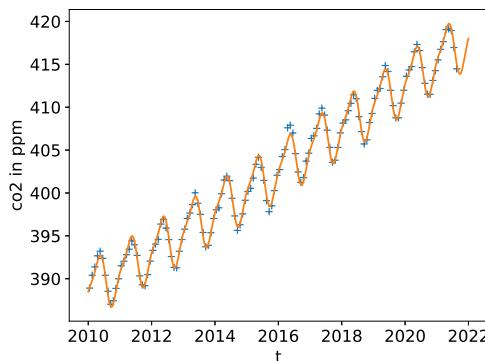
Vi kan använda vår anpassade modellfunktion för att förutsäga CO₂-halten 2050 (vi bortser från de årliga svängningarna). Kommandona är

```
t = 2050
co2_2050 = a[0] + a[1]*t + a[2]*t**2
print('CO2 år 2050: ',round(co2_2050,1),'ppm')
```

Python svarar

```
CO2 år 2050: 510.2 ppm
```

FN organet 'Framework Convention on Climate Change' (UNFCCC) har uppskattat att en CO₂ koncentration på 450 ppm ger oss en 50 % chans att hålla oss inom en temperaturhöjning på 2 grader. Om inte drastiska åtgärder kommer på plats, missar vi detta målet grovt. Läs mer om CO₂ utsläppen och dess konsekvenser på <https://www.theworldcounts.com/challenges/global-warming/CO2-concentration>. CO₂ koncentration tillsammans med anpassad modellfunktion visas i figur 14.7.



Figur 14.7: CO₂-koncentration i atmosfären tillsammans med anpassad modellfunktion.

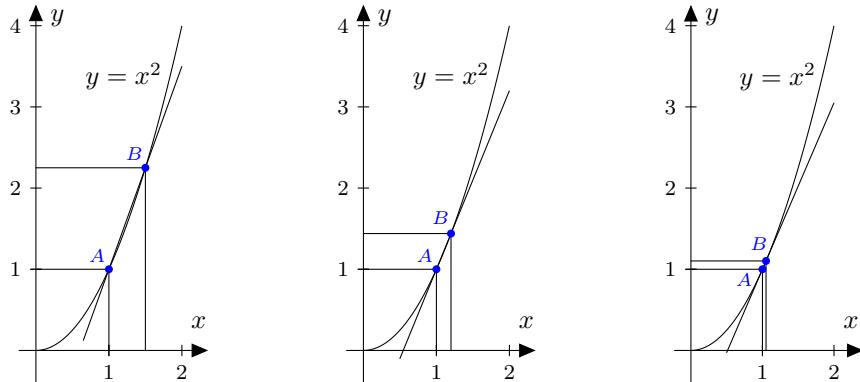
Kapitel 15

Förändring

Funktioner ligger till grund för modellering. Om en funktion beskriver ett system som funktion av tiden vill vi ofta kunna ange hur fort systemet ändras vid olika tider t . Detta görs genom att beräkna funktionens derivata. Derivata ligger också till grund för så kallade dynamiska modeller vilka vi studerar i nästa kapitel.

15.1 Tangenten till en kurva

De tre figurerna nedan visar alla samma stycke av kurvan $y = x^2$. I var och en av figurerna har vi markerat två punkter på kurvan, nämligen punkten $A = (1, 1)$ och dessutom en annan punkt B .



I den första figuren har B koordinaterna $(1.5, 2.25)$. Linjen AB går genom punkterna $(1.5, 2.25)$ och $(1, 1)$ och har alltså riknings koefficienten

$$\frac{2.25 - 1}{1.5 - 1} = \frac{1.25}{0.5} = 2.5.$$

I den andra figuren har B flyttats närmare A . Punkten B har nu koordinaterna $(1.2, 1.44)$, och linjen AB har riktningskoefficienten

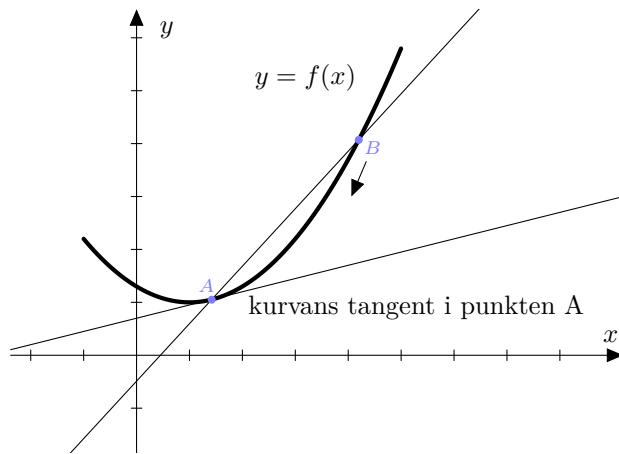
$$\frac{1.44 - 1}{1.2 - 1} = \frac{0.44}{0.2} = 2.2.$$

I den tredje figuren har B kommit ännu närmare A . Punkten B har koordinaterna $(1.05, 1.1025)$, och AB har riktningskoefficienten

$$\frac{1.1025 - 1}{1.05 - 1} = \frac{0.1025}{0.05} = 2.05.$$

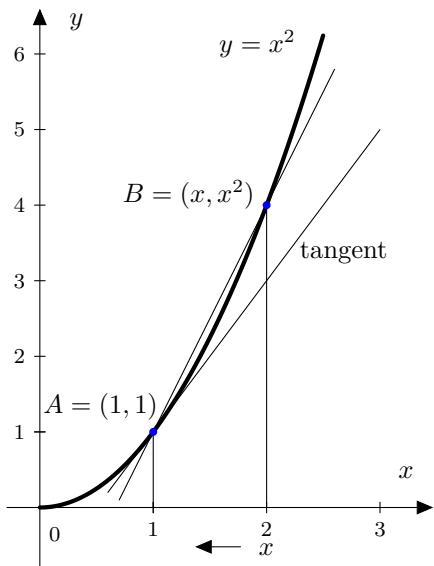
Om B kommer mycket nära A , så ”snuddar” linjen AB vid kurvan i punkten A , och linjens riktningskoefficient är praktiskt taget 2. Antag t. ex. att B har x -koordinaten 1.001. Punktens y -koordinat är då $1.001^2 = 1.002001$, och AB har riktningskoefficienten

$$\frac{1.002001 - 1}{1.001 - 1} = \frac{0.002001}{0.001} = 2.001 \approx 2.$$



På kurvan ovan har två punkter A och B markerats, och linjen AB har dragits. Tänk dig att punkten A ligger stilla men att punkten B glider längs kurvan in mot A . Då kommer linjen AB att alltmera närlägga sig den tangerande linjen i punkt A i figuren. Denna linje kallas kurvans *tangent* i punkten A .

Vi ska beräkna riktningskoefficienten för tangenten till kurvan $y = x^2$ i punkten $A = (1, 1)$.



Vi gör som i förra exemplet, men vi betecknar första koordinaten för punkten B med x . I varje punkt på kurvan är den andra koordinaten för punkten B lika med kvadraten på den första koordinaten ($y = x^2$). Punkten B har alltså koordinaterna (x, x^2) . Linjen AB har riktningskoefficienten

$$\frac{x^2 - 1}{x - 1} = \frac{(x + 1)(x - 1)}{x - 1} = x + 1.$$

Nu låter vi punkten B glida längs kurvan in mot A . Det betyder att vi låter $x \rightarrow 1$, dvs. låter x ”gå mot” 1. Då måste förstås riktningskoefficienten $x + 1 \rightarrow 2$.

Slutsats: Tangenten i punkten $A = (1, 1)$ har riktningskoefficienten 2.

15.2 Derivator

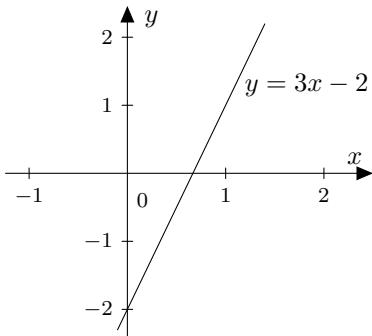
Man betecknar ofta en funktion med en bokstav, t.ex. f . Funktionsvärdet i en punkt x betecknas då $f(x)$. Ekvationen

$$y = f(x)$$

motsvaras i ett koordinatsystem av en punktmängd. Denna punktmängd kallas funktionens *graf* eller *funktionskurvan*.

I de flesta fall kan man dra en tangent i en godtyckligt vald punkt $(x, f(x))$ på funktionskurvan. Riktningskoefficienten för denna tangent brukar betecknas $f'(x)$, "f prim x ". Den funktion f' som har funktionsvärdena $f'(x)$ kallas *derivatan* till funktionen f .

Vi låter f vara den funktion vars funktionsvärdet är $f(x) = 3x - 2$.



Funktionens graf är $y = 3x - 2$, dvs. en linje med riktningskoefficienten 3 (se figuren till vänster). I detta fall är det inget problem att dra tangenter till grafen. Varje tangent sammanfaller förstås med grafen. Tangentens riktningskoefficient är alltså alltid 3, dvs.

$$f'(x) = 3 \quad \text{för alla } x$$

Eller annorlunda uttryckt: Funktionens derivata har överallt värdet 3.

Samma resonemang som i exemplet ger följande *deriveringsregel*:

$$f(x) = kx + b \quad (k \text{ och } b \text{ är konstanter}) \quad \text{derivata} \quad f'(x) = k.$$

Regeln uttrycker det faktum att linjen $y = kx + b$ har riktningskoefficienten k . Lägg särskilt märke till följande specialfall (grafen är ett horisontellt streck):

$$f(x) = b \quad (\text{en konstant}) \quad \text{derivata} \quad f'(x) = 0.$$

I tidigare exempel visade vi att tangenten till kurvan $y = x^2$ i punkten (a, a^2) har riktningskoefficienten $2a$. Tangenten i punkten (x, x^2) har alltså riktningskoefficienten $2x$. Detta resultat ger följande deriveringsregel:

$$f(x) = x^2 \quad \text{derivata} \quad f'(x) = 2x.$$

Eller i ord: derivatan av x^2 är $2x$. (Egentligen borde man säga " derivatan av funktionen $y = x^2$ är funktionen $y = 2x$ ". Men detta blir obekvämt långt.)

Exempel 15.1. Beräkna $f'(x)$ då

a) $f(x) = 6x^2$

I högra ledet står 6 gånger x^2 . Man kan visa att derivatan är 6 gånger derivatan av x^2 , dvs. att

$$f'(x) = 6 \cdot 2x = 12x.$$

b) $f(x) = 4x^2 - 3x + 2$

Högra ledet är summan av termerna $4x^2$, $-3x$ och 2.

Man kan visa att derivatan är summan av termernas derivator, dvs. att

$$f'(x) = 4 \cdot 2x - 3 + 0 = 8x - 3.$$

□

Funktionsvärdet $f(x)$ betecknas ofta y . Derivatans funktionsvärde $f'(x)$ betecknas då y' . Om t.ex.

$$y = 6x^2$$

så är $y' = 6 \cdot 2x = 12x$.

□

15.3 Viktiga deriveringsregler

Vi börjar med potenser. För varje exponent p gäller följande deriveringsregel:

$$y = x^p \quad \text{derivata} \quad y' = px^{p-1}$$

Då vi övergår till exponentialfunktionen. För varje k gäller följande:

$$y = e^{kx} \quad \text{derivata} \quad y' = ke^{kx}$$

Bevisen för reglerna är svårt och måste utelämnas. Några specialfall av reglerna ska vi dock bevisa lite längre fram.

Exempel 15.2. Beräkna y' då

$$y = x^4 - 2x^3 + 3x^2 - 4x + 5.$$

Som nämnts tidigare kan man visa att derivatan av en summa är lika med summan av termernas derivator. Alltså gäller:

$$y' = 4x^3 - 2 \cdot 3x^2 + 3 \cdot 2x - 4 + 0$$

$$y' = 4x^3 - 6x^2 + 6x - 4.$$

□

Exempel 15.3. Beräkna y' då

$$y = 4e^{3x}.$$

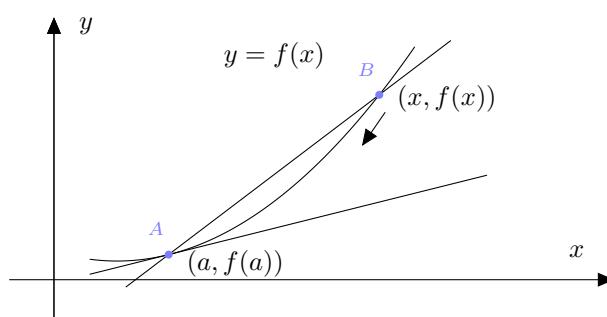
Användning av regeln ovan ger

$$y' = 4 \cdot 3e^{3x} = 12e^{3x}.$$

□

15.4 Om derivatans definition

Figuren föreställer grafen till en funktion f . I punkten $A = (a, f(a))$ har vi dragit tangenten till grafen. Tangentens riktningskoefficient är $f'(a)$, dvs. ett funktionsvärde till funktionens derivata f' .



För att beräkna $f'(a)$ så väljer vi en godtycklig punkt $B = (x, f(x))$ på grafen och drar linjen AB . Linjens riktningskoefficient är lika med kvoten

$$\frac{f(x) - f(a)}{x - a}.$$

Riktningskoefficienten $f'(a)$ för tangenten är det värde som denna kvot närmrar sig då $x \rightarrow a$. Detta brukar skrivas så här:

$$f'(a) = \lim_{x \rightarrow a} \frac{f(x) - f(a)}{x - a}.$$

Tecknet $\lim_{x \rightarrow a}$ läses ”gränsvärdet då x går mot a ” eller ”limes då x går mot a ”. (Limes är latin och betyder gräns).

Derivatan av en funktion f definieras vanligen genom likheten ovan. I många fall då derivatan tillämpas är det nämligen av föga intresse att veta att den är riktningskoefficient för en tangent till funktionens graf. Detta gäller om nästan alla naturvetenskapliga och tekniska tillämpningar av derivatan.

Exempel 15.4. Beräkna $f'(a)$ då $f(x) = x^4$.

Om $f(x) = x^4$, så är $f(a) = a^4$, och

$$f(x) - f(a) = x^4 - a^4 = (x^2 + a^2)(x^2 - a^2).$$

Här har vi använt konjugatregeln. Det går att använda regeln en gång till:

$$f(x) - f(a) = (x^2 + a^2)(x + a)(x - a).$$

Nu dividerar vi bågge leden med $x - a$

$$\frac{f(x) - f(a)}{x - a} = \frac{(x^2 + a^2)(x + a)(x - a)}{x - a} = (x^2 + a^2)(x + a).$$

Så använder vi derivatans definition

$$\begin{aligned} f'(a) &= \lim_{x \rightarrow a} \frac{f(x) - f(a)}{x - a} = \lim_{x \rightarrow a} (x^2 + a^2)(x + a) = \\ &= (a^2 + a^2)(a + a) = 2a^2 \cdot 2a = 4a^3. \end{aligned}$$

Vi finner alltså att $f'(a) = 4a^3$. Om vi i denna likhet byter ut a mot x , så får vi följande derivningsregel (som vi hoppas du känner igen):

$$f(x) = x^4 \quad , \quad f'(x) = 4x^3$$

□

15.5 Om derivatans beteckning

Vi har hittills betecknat en funktion med bokstaven f och variabeln i funktionsuttrycket med x . Man kan naturligtvis använda andra bokstäver. Exempelvis bestämmer ekvationen

$$g(t) = 4t^2 - 5t$$

en funktion g , och derivering ger att

$$g'(t) = 8t - 5.$$

Flera gånger har vi betecknat funktionsvärdet $f(x)$ med y , dvs. vi har skrivit $y = f(x)$. Derivatans värde har vi då betecknat y' . Man använder också beteckningen $\frac{dy}{dx}$ (uttalas "dydx"). Om t.ex.

$$y = 2x^3 + 6x \quad \text{så är} \quad \frac{dy}{dx} = 6x^2 + 6$$

Naturligtvis kan man också ha andra bokstäver. Om t. ex.

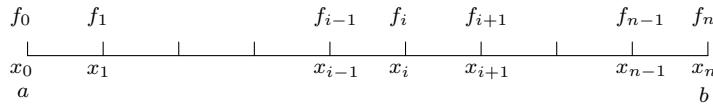
$$s = 3t - t^2 \quad \text{så är} \quad s' = 3 - 2t \quad \text{eller} \quad \frac{ds}{dt} = 3 - 2t.$$

15.6 Gridd och finita differensapproximationer

Med en gridd på intervallet $[a, b]$ menar vi ett antal punkter x_0, x_1, \dots, x_n sådana att

$$a = x_0 < x_1 < x_2 < \dots < x_{n-1} < x_n = b.$$

Vi ska här titta på hur vi kan approximera derivatorna av en funktion $f(x)$ genom finita differenser baserade endast på funktionsvärden på gridden. För enkelhetens skull antar vi att gridden är likformig så att $x_{i+1} - x_i = h$, där $h = (b - a)/n$. I figur 15.1 visas en funktion given på en likformig gridd med griddpunktsavståndet h . Då det är skönt att slippa skriva så mycket har vi inför beteckningen f_i för $f(x_i)$.



Figur 15.1: Funktionen $f(x)$ i intervallet $[a, b]$ given på en likformig gridd x_0, x_1, \dots, x_n med griddavståndet $h = (b - a)/n$.

Med användning av definitionen i avsnitt 15.4 har vi att derivatan i punkten x_i kan approximeras med den så kallade framåtdifferensen

$$f'(x_i) \approx \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} = \frac{f(x_{i+1}) - f(x_i)}{h}$$

Derivatan kan också approximeras med den så kallade bakåtdifferensen

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} = \frac{f(x_i) - f(x_{i-1})}{h}$$

Man kan visa att en ännu bättre approximation ges av centraldifferensen

$$f'(x_i) \approx \frac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}} = \frac{f(x_{i+1}) - f(x_{i-1})}{2h},$$

vilken är ett medeldärde av framåt- och bakåtdifferenserna. Vi har nu följande metod för att plotta grafen till $f'(x)$ i Python: generera en likformig gridd x_0, x_1, \dots, x_n , beräkna $f'(x_0)$ med hjälp av framåtdifferensen och $f'(x_n)$ med hjälp av bakåtdifferensen, för inre punkter x_i beräkna $f(x_i)$ med hjälp av centraldifferensen, plotta de beräknade derivatorna på gridden.

Exempel 15.5. Vi har en funktion

$$f(x) = \frac{x^2}{1 + x^2}.$$

Följande program definierar en funktion med namn `fdiff`, som beräknar derivatan med hjälp av finita differenser. Funktionen tar en vektor f med funktionsvärden beräknade på en gridd x och tillhörande griddavstånd h som inparametrar och returnerar en vektor med derivatan beräknad på gridden. Vi plottar funktionen $f(x)$ och kallar sedan på funktionen för derivatan för att beräkna och plotta $f'(x)$ i intervallet $[0, 5]$. Dessutom beräknas största värdet för derivatan och i vilken punkt x derivatan är som störst (funktionen växer snabbast). För att få en bra approximation på derivatan tar vi 2000 gridpunkter.

```

import numpy as np
import matplotlib.pyplot as plt

# funktion för att beräkna derivatan med hjälp av finita differenser
def fdiff(f,h):
    """
        beräkna derivatan f'(x) m.h.a finita differenser
        f vektor med funktionsvärden på en gridd
        h griddavstånd
        df vektor med derivator på en gridd
    """
    n = len(f)                                # antalet griddpunkter
    df = np.zeros(n)                           # vektor med derivatan
    for i in range(n):                        # loopa över griddpunkterna
        if i == 0:                            # första punkten
            df[i] = (f[i+1] - f[i])/h          # framåtdifferens
        elif i == n - 1:                      # sista punkten
            df[i] = (f[i] - f[i-1])/h          # bakåtdifferens
        else:                                # inre punkter
            df[i] = (f[i+1] - f[i-1])/(2*h)   # centraldifferens
    return df

x = np.linspace(0,5,2000)                   # tät gridd
h = x[1] - x[0]                            # griddavstånd
f = x**2/(1+x**2)                          # vektor med funktionsvärden

fig, ax = plt.subplots()                    # plotta funktionen
ax.plot(x,f,label="f(x)")                 # fdiff(f,h) vektor med dervatan
ax.plot(x,fdiff(f,h),label="f'(x)")
ax.tick_params(labelsize=14)
ax.grid('on')
ax.legend(fontsize=14)

# dessa kommandon ger axlar genom origo
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_position('zero')
ax.spines['bottom'].set_position('zero')

# bestäm maxvärdet för derivatan och i vilken punkt
print("Maxvärdet för f'(x) ",round(np.max(fdif(f,h)),2))
print("Maxvärdet för f'(x) antas i x = ",\
      round(x[np.argmax(fdif(f,h))],2))

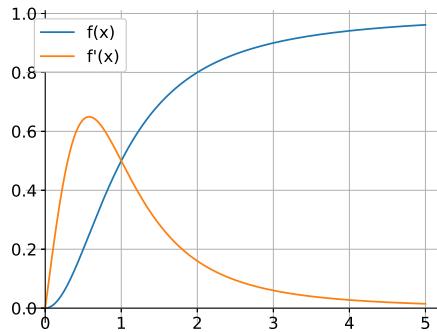
```

Då vi kör programmet svarar Python

Maxvärdet för f'(x) 0.65

Maxvärde för $f'(x)$ antas i $x = 0.58$

Plotten med funktionen och derivatan visas i figur 15.2. \square



Figur 15.2: Funktionen $f(x) = x^2/(1+x^2)$ tillsammans med derivatan $f'(x)$.

15.7 Tillämpning: befolkningstillväxt

I avsnitt 14.4 anpassade vi en logistisk modellfunktion till befolkningsdata för år 1951 ($t = 0$) till år 2023 och fick fram att

$$N(t) = \frac{12.459 \cdot 2.494 e^{0.0277t}}{2.494 e^{0.0277t} + (12.459 - 2.494)}$$

gav en bra beskrivning. Befolkningsdata tillsammans med anpassad modellfunktion visas till vänster i figur 15.3. Vi ska nu använda modellen för att besvara frågan när förändringen per år (derivatan) är som störst och hur stor den är. För att lösa uppgiften beräknar vi derivatan och använder den för att lösa uppgiften. Följande Pythonprogram plottar derivata $N'(t)$ i intervallet $[0, 149]$. Dessutom beräknas största värdet för derivatan och i vilken tid t derivatan är som störst (befolkningen växer snabbast). Även den maximala årliga tillväxten beräknas och redovisas. För att få en bra approximation på derivatan tar vi 2000 griddpunkter.

```
import numpy as np
import matplotlib.pyplot as plt

# funktion för att beräkna derivatan med hjälp av finita differenser
def fdiff(f,h):
    """
    beräknar derivatan f'(x) m.h.a finita differenser
    f vektor med funktionsvärden på en gridd
    h griddavstånd
    df vektor med derivator på en gridd
    """
    n = len(f)                                # antalet griddpunkter
    df = np.zeros(n)                            # vektor med derivatan
    for i in range(n):                         # loopa över griddpunkterna
        if i == 0:                             # första punkten
            df[i] = (f[i+1] - f[i])/h          # framåtdifferens
        elif i == n - 1:                        # sista punkten
            df[i] = (f[i] - f[i-1])/h          # bakåtdifferens
        else:                                 # inomgräns
            df[i] = (f[i+1] - f[i-1])/(2*h)
```

```

        else:                                # inre punkter
            df[i] = (f[i+1] - f[i-1])/(2*h)  # centraldifferens
        return df

t = np.linspace(0,149,2000)                  # tät gridd
h = t[1] - t[0]                            # griddasvtånd
a = 12.459*2.494*np.exp(0.0277*t)         # täljare
b = 2.494*np.exp(0.0277*t) + (12.459-2.494) # nämnare
N = a/b                                    # vektor med funktionsvärde

fig, ax = plt.subplots()
ax.plot(t,fdiff(N,h),label="N'(t)")          # fdiff(N,h) vektor med dervatan
ax.tick_params(labelsize=14)
ax.set_xlabel('t i år från 1951',fontsize=14)
ax.set_ylabel('befolkningsförändring per år i miljarder',fontsize=14)
ax.grid('on')
ax.legend(fontsize=14)

# bestäm maxvärde för derivatan och i vilken punkt
dmax = np.max(fdiff(N,h))                  # maximalt värde på derivatan
imax = np.argmax(fdiff(N,h))                # index för maximalt värde
tmax = t[imax]                             # motsvarande tid

print("Maximal befolkningsförändring per år ",round(dmax,5),"miljarder")
print("Maximal befolkningsförändring för t = ",round(tmax))

# bestäm maximal årlig tillväxt i procent (relativ förändring)
print("Maximal årligt tillväxt ",round(100*dmax/N[imax],3),"procent")

```

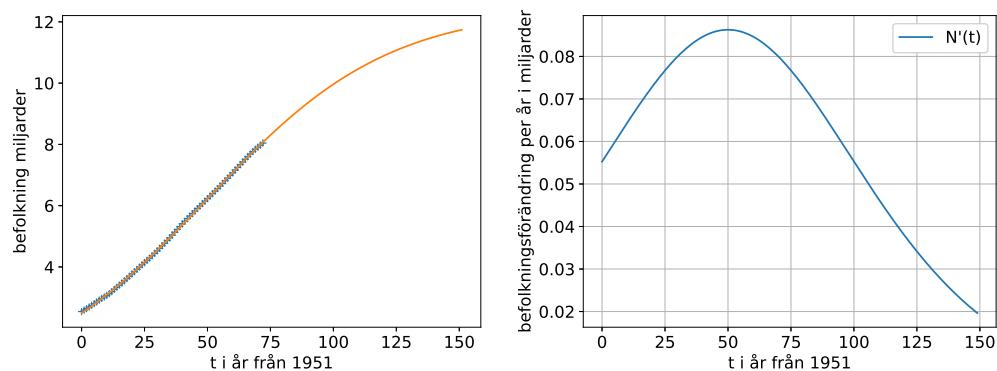
Då vi kör programmet får vi utskriften

```

Maximal befolkningsförändring per år  0.08628 miljarder
Maximal befolkningsförändring för t =  50
Maximal årligt tillväxt  1.385 procent

```

Den maximal befolkningsförändring per år är 0.08628 miljarder och antas för $t = 50$, vilket motsvarar år 2001. Den maximala årliga tillväxten (relativa förändringen) är 1.385 %. Plotten med derivatan visas i till höger i figur 15.3.



Figur 15.3: Jordens befolkning kan modelleras med en logistisk funktion. Till höger visas derivatan, från vilken vi kan utläsa befolkningsförändringen per år. Befolkningsförändringen per år är som störst för $t = 50$, vilket motsvara år 2001.

Kapitel 16

Dynamiska modeller

I det här kapitlet skall vi titta på dynamiska modeller, dvs. modeller som beskriver hur system utvecklar sig med tiden. System tas i vid bemärkelse och kan vara allt från en kopp med kaffe som svalnar, där vi tittar på temperaturen, till två konkurrerande djurarter, där vi tittar på hur populationen av de två djurarterna utvecklas med tiden.

16.1 Differentialekvationer

Vi vill konstruera en modeller för hur en systemvariabel y beror av tiden t . Ofta kan våra modellhypoteser (antaganden om systemet) uttryckas som en relation mellan systemvariabelns förändringshastighet dy/dt och systemvariabeln y själv

$$\underbrace{\frac{dy}{dt}}_{\text{förändring per tid}} = \underbrace{f(t, y)}_{\text{uttryck med } y \text{ och } t}.$$

En sådan relation kallas en differentialekvation. Givet ett värde y_0 på systemvariabeln vid tiden t_0 (när vi börjar studera systemet), kan vi enklare fall lösa ekvationen analytiskt och få fram den exakta lösningen $y(t)$. I mera komplicerade fall är det enda vi kan göra att beräkna approximationer till den (okända) exakta lösningen $y(t)$ i ett antal griddpunkter på ett intervall $[t_0, t_{\text{end}}]$. En differentialekvation tillsammans med ett värde y_0 , ett så kallat begynnelsevärde, i en punkt t_0 kallas ibland för ett begynnelsevärdesproblem (eng. initial value problem).

Exempel 16.1. Radioaktivt sönderfall. Låt $N(t)$ vara antalet atomer i ett radioaktivt prov som funktion av tiden t . Noggranna experiment visar att förändringen per tidsenhet är proportionellt mot antalet atomer i provet, dvs.

$$\frac{dN}{dt} = -\lambda N,$$

där λ är en positiv konstant (sönderfallskonstanten) som är karakteristisk för ämnet. Att förändringen per tid är proportionell mot antalet atomer N uttrycker bara det enkla faktum att om vi har 10 sönderfall per tid då antalet atomer N är 1000 så har vi 100 sönderfall per tid då antalet N är 10 000 (detta är proportionalitet, se avsnitt 10.4). \square

Exempel 16.2. Vi tänker oss att vi har en kropp med temperatur T som befinner sig i ett omgivande medium med konstant temperatur $T_{\text{omg.}}$. Det kan till exempel vara en kopp varmt kaffe som står på ett bord. Det är rimligt (stöds av experiment) att anta att ändringen av temperaturen per tidsenhet (avsvalningen) är proportionell mot skillnaden mellan kroppens temperatur och omgivningens temperatur, $T - T_{\text{omg.}}$. Detta stämmer med erfarenheten: då kaffet är varmt

och skillnaden mellan kaffets temperatur och omgivningens temperatur är stor går avsvalningen fort. Då kaffet har nästan samma temperatur som omgivningen går avsvalningen långsamt. Differentialekvationen blir

$$\frac{dT}{dt} = -k(T - T_{\text{omg}}),$$

där k är en positiv konstant som bestämmer hur fort värmeutbytet är mellan kroppen och omgivningen. Ju större värde på k , desto snabbare avsvalningsprocess. \square

16.2 Differentialekvationer – stegmetoder

Differentialekvationer av typen

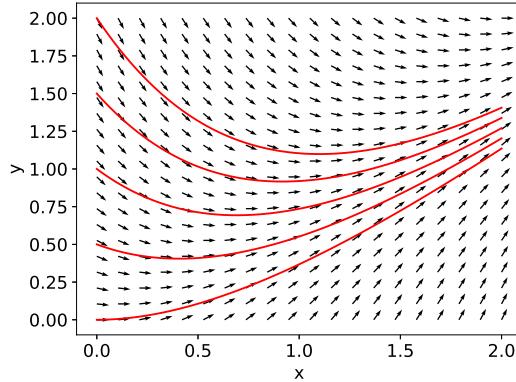
$$\frac{dy}{dt} = f(t, y)$$

har en enkel geometrisk tolkning. Derivatan dy/dt ger riktningskoefficienten för lösningen genom punkten (t, y) . Genom likheten $dy/dt = f(t, y)$ associerar därmed funktionen $f(t, y)$ en riktning till varje punkt i ty -planet. Genom att markera riktningen med en liten pil får man det så kallade riktningsfältet. Genom att studera riktningsfältet är det ofta möjligt att få en kvalitativ uppfattning om lösningarna till differentialekvationen.

Exempel 16.3. Vi har differentialekvationen

$$\frac{dy}{dt} = t - y.$$

Riktningsfältet visas i figur 16.1, där vi även plottat några lösningar. Givet ett värde y_0 på systemvariabeln vid tiden t_0 kan vi fram värdena på $y(t)$ genom att följa riktningsfältet. \square



Figur 16.1: Riktningsfält till $dy/dt = t - y$ tillsammans med några lösningar.

Eulers metod

Riktningsfältet kan tas som utgångspunkt för numeriska metoder. I Eulers metod söker man en lösning till ekvationen

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0$$

i ett antal ekvidistanta punkter t_0, t_1, \dots, t_n i intervallet $[t_0, t_{\text{end}}]$. Avståndet mellan punkterna, den så kallade steglängden, ges av $h = (t_{\text{end}} - t_0)/n$. Som approximation till lösningskurvan nära

(t_0, y_0) använder man dess tangent. Ett approximativt värde y_1 på lösningskurvan i en punkt $t_1 = t_0 + h$ ges då av

$$y_1 = y(t_0) + \frac{dy}{dt}(t_0)h = y_0 + f(t_0, y_0)h.$$

Detta närmevärde kan i sin tur utnyttjas för att konstruera ett närmevärde y_2 i punkten $t_2 = t_0 + 2h$ osv., vilket ger formeln

$$y_{k+1} = y_k + f(t_k, y_k)h.$$

Känner vi bara begynnelsevärdet y_0 får vi approximationer till den exakta lösningen $y(t)$ från formeln ovan.

16.3 Lösning av differentialekvationer i Python

Python implementerar flera metoder, mer eller mindre relaterade till Eulers metod, för att beräknar man approximationer till den exakta lösningen $y(t)$ i ett antal diskreta punkter i ett intervall $[t_0, , t_{\text{end}}]$. Funktionerna som löser ordinära differentialekvationer är samlade i modulen `scipy.integrate` som importeras genom

```
import scipy.integrate as intgr
```

Den mest användbara funktionen är `solve_ivp`, som har flera optioner. Här står ivp står för initial value problem.

```
s = intgr.solve_ivp(f, [t0,tend],y0,opt)
```

beräknar lösningen till en differentialekvation. Ekvationens högerled ges i f . Lösningsintervallet ges i $[t_0, t_{\text{end}}]$. Begynnelsevärdena är samlade i en vektor y_0 . Lösningen beräknas i ett antal diskreta punkter. Optioner finns för att sätta integrationsmetod, noggrannhet, punkter där vi vill att lösningen ska beräknas och villkor för att avbryta integrationen innan vi har nått högerintervallgräns $, t_{\text{end}}$.

Exempel 16.4.

(a) För att lösa begynnelsevärdesproblemet

$$\frac{dN}{dt} = -\lambda N, \quad N(0) = 1$$

i intervallet $[0, 3]$ med $\lambda = 1$, definierar vi högersidan i en funktion f . Observera att vi måste ha både t och N (i denna ordning) som invariabler även om t inte används. Notera också att även om vi bara har ett begynnelsevärdet så måste detta ges som en vektor med ett element och inte bara som ett tal. Vi kan nu kalla på lösaren

```
import scipy.integrate as intgr
f = lambda t, N: -N # obs, t måste med
sol = integr.solve_ivp(f,[0,3],[1]) # [1] och inte bara 1.
print(sol)
```

All information om lösningen är lagrade i instansen `sol`

```
message: 'The solver ... the end of the integration interval.'
nfev: 32
njev: 0
```

```

nlu: 0
sol: None
status: 0
success: True
t: array([0., 0.10001999, 1.03186487, 1.90765136,
         2.78720278, 3.          ])
t_events: None
y: array([[1., 0.90481933, 0.35660435, 0.14860781,
         0.06169755, 0.04987137]])
y_events: None

```

Griddpunkterna i vilka vi får lösningen ges i `sol.t`. Motsvarande lösning ges i `sol.y`. Notera att `sol.t` är en vektor medan `sol.y` är en matris med en rad.

Om vi vill ha lösningen i bestämda punkter, t.ex. $(0, 1, 2, 3)$ använder vi optionen `t_eval` på följande sätt

```

import scipy.integrate as intgr
f = lambda t, N: -N           # obs, t måste med
sol = intgr.solve_ivp(f,[0,3],[1],t_eval=[0,1,2,3])
print(sol)

```

Instansen `sol` blir nu

```

message: 'The solver ... the end of the integration interval.'
nfev: 32
njev: 0
nlu: 0
sol: None
status: 0
success: True
t: array([0, 1, 2, 3])
t_events: None
y: array([[1., 0.36814005, 0.13548843, 0.04987137]])
y_events: None

```

För att plotta lösningen måste denna beräknas på en tät gridd i lösningsintervallet och kommandona blir

```

import matplotlib.pyplot as plt
import numpy as np
import scipy.integrate as intgr
f = lambda t, N:
sol = intgr.solve_ivp(f,[0,3],[1],\
                      t_eval=np.linspace(0,3,100))      # tät gridd för plottning
fig, ax = plt.subplots()
ax.plot(sol.t,sol.y[0])           # sol.y är en matris, lösningen
                                   # ges i första raden, sol.y[0]
ax.set_xlabel('t',fontsize=14)
ax.set_ylabel('N(t)',fontsize=14)
ax.tick_params(labelsize=14)

```

Plotten visas till vänster i figur 16.2.

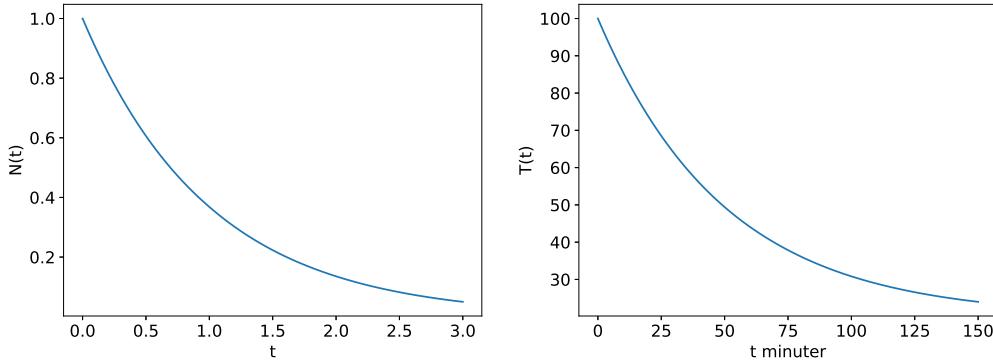
(b) För att lösa ekvationen

$$\frac{dT}{dt} = -k(T - T_{\text{omg}}), \quad T(0) = 100$$

i intervallet $[0, 150]$ (minuter) med $k = 0.02$ och $T_{\text{omg.}} = 20$ och plotta lösningen ger vi kommandona

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.integrate as intgr
f = lambda t, T: -0.02*(T - 20)
sol = intgr.solve_ivp(f, [0,150], [100], \
                      t_eval=np.linspace(0,150,100))      # tät gridd för plottning
fig, ax = plt.subplots()
ax.plot(sol.t,sol.y[0])                  # sol.y är en matris, lösningen
                                         # ges i första raden, sol.y[0]
ax.set_xlabel('t minuter', fontsize=14)
ax.set_ylabel('T(t)', fontsize=14)
ax.tick_params(labelsize=14)
```

Plotten visas till höger i figur 16.2. Temperaturen beter sig som vi förväntar oss. Den minskar fort i början då skillnaden till omgivningen är stor och långsamt i slutet då skillnaden är liten. \square



Figur 16.2: Numeriska lösningar till differentialekvationer.

16.4 Anpassning av modellparametrar

Detta är ett utmanande avsnitt! Dynamiska modeller beror ofta av parametrar. Till exempel beror modellen för radioaktivt sönderfall

$$\frac{dN}{dt} = -\lambda N$$

av sönderfallskonstanten λ , medan modellen för avsvalning

$$\frac{dT}{dt} = -k(T - T_{\text{omg}})$$

beror av k , som bestämmer hastigheten på värmeutbytet är mellan kroppen och omgivningen.

I kapitel 14 har vi bestämt parametrar i modelfunktioner genom minstakvadratanpassningar till data. Samma metod kan användas för att bestämma parametrar i dynamiska modeller, där modelfunktionen nu fås som en lösning till differentialekvationen. Metoden att bestämma parametrar i dynamiska modeller beskrivs enklast med ett exempel, se även avsnitt 16.5.3.

Exempel 16.5. Temperaturen $T(t)$ i en kaffemugg som svalnar av ges som lösning av differentialekvationen

$$\frac{dT}{dt} = -k(T - T_{\text{omg}}).$$

Vid ett amatörmässigt experiment, utfört av PJ, mättes temperaturen av morgonkaffet. Mätningarna finns redovisade i tabell 16.1.

Tabell 16.1: Uppmätt temperatur T på kaffet vid några olika tider t .

t (min)	0	10	20	40	63	90	126
T (°C)	68	56.5	49	40	33.5	29	26

Följande program bestämmer modellparametrarna och plottar motsvarande modelfunktion (lösning till differentialekvationen) som funktion av tiden tillsammans med uppmätt temperaturer. I minstakvadratanpassningen låter vi $a_0 = k$ och $a_1 = T_{\text{omg}}$. Notera hur vi bestämmer lösningen till differentialekvationen vid tiderna som är angivna i tabellen ($t_{\text{eval}} = \text{tdata}$). När vi har bestämt modellparametrarna och ska plotta modelfunktionen använder vi en tät gridd ($t_{\text{eval}} = t$, där $t = \text{np.linspace}(0, 150, 200)$). Notera även att vi inte använder $T(0) = 68$ från tabellen, utan tar $T(0)$ som en parameter a_2 , vilken bestäms vid minstakvadratanpassningen. Vi behöver startvärden för minstakvadratanpassningen. Från tabellen med data får vi att $a_1 = T_{\text{omg}} = 20$. Lite tester visar att $a_0 = 0.02$ är ett rimligt värde. Slutligen tar vi $a_2 = N(0) = 68$.

```

import matplotlib.pyplot as plt
import numpy as np
import scipy.integrate as intgr
import scipy.optimize as opt

# modelfunktion fås som lösning till diffekv
# lösning räknas ut för de t vi har i tabellen (tdata)
def fmodell(a,t):
    f = lambda t, T: -a[0]*(T - a[1]) # högerled i diffekv
    sol = intgr.solve_ivp(f, [0,126], [a[2]], t_eval = tdata)
    return sol.y[0]

def res(a,tdata,Tdata): # residual
    return Tdata - fmodell(a,tdata)

tdata = np.array([0,10,20,40,63,90,126])
Tdata = np.array([68,56.5,49,40,33.5,29,26])

a0 = [0.02,20,68]      # startgissning

a,q = opt.leastsq(res,a0,(tdata,Tdata))
print('Parametrar:',a)

# plotta modelfunktion tillsammans med uppmätta data
fig, ax = plt.subplots()
ax.plot(tdata,Tdata,'o')
t = np.linspace(0,150,200)
f = lambda t, T: -a[0]*(T - a[1])
sol = intgr.solve_ivp(f, [0,150], [a[2]], t_eval = t) # tät gridd för plottning
ax.plot(t,sol.y[0])
ax.set_xlabel('t minuter', fontsize=14)

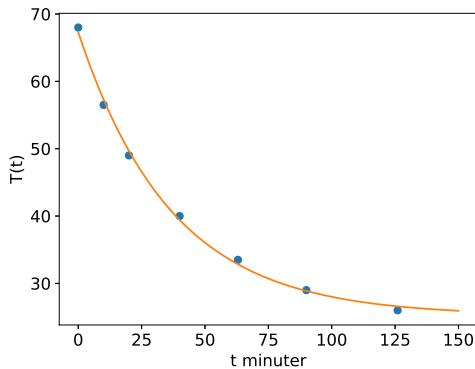
```

```
ax.set_ylabel('T(t)', fontsize=14)
ax.tick_params(labelsize=14)
```

Då vi kör programmet skriver Python ut

```
Parametrar: [2.71213004e-02 2.52040217e+01 6.73035808e+01]
```

Vi har alltså $k = 0.027$. Vidare är omgivningens temperatur $T_{\text{omg}} = 25.2$. Anpassningen ger $T(0) = 67.3$, vilket ligger nära värdet 68°C från tabellen. Den anpassade modelfunktionen, vilken fås som lösning till differentialekvationen, är plottad tillsammans med data i figur 16.3. \square



Figur 16.3: Modelfunktion given som lösning till differentialekvationen tillsammans med uppmätta data. Modellparametrarna har bestämts med hjälp av minstkvadratanpassningar till data.

16.5 Modeller för populationsutveckling

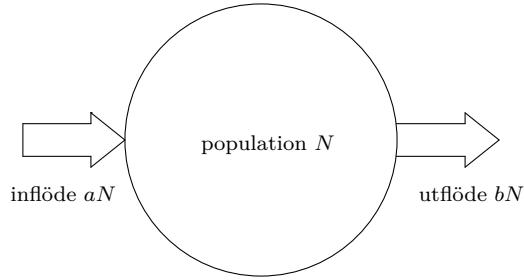
Vi ska sätta upp en modell för antalet individer N i en population förändrar sig med tiden t . En population ökar genom födslar och immigration (invandring) och minskar genom dödsfall och emigration (utvandring). För enkelhetens skull antar vi att populationen är sluten, dvs. att immigration och emigration är försumbara.

16.5.1 Obegränsad tillväxt

Det är, som en första modellhypotes, rimligt att anta att både antalet som föds per tidsenhet (inflödet) och antalet som dör per tidsenhet (utflödet) är proportionella mot antalet individer N i populationen (om det föds 10 individer per tidsenhet vid en population om 1000 så föds det 100 individer per tidsenhet vid en population om 10 000). Modellen illustreras grafiskt i figur 16.4. Matematiskt formulerad blir modellhypotesen

$$\underbrace{\frac{dN}{dt}}_{\text{förändring per tid}} = \underbrace{aN}_{\text{inflöde}} - \underbrace{bN}_{\text{utflöde}} = \underbrace{rN}_{\text{nettoflöde}}$$

Om inflödet är större än utflödet blir nettoflödet r positivt och populationen växer hela tiden. Omvänt, om inflödet är mindre än utflödet blir nettoflödet r negativt och populationen minskar hela tiden. Följande program löser differentialekvationen i tidsintervallet $[0, 50]$ för $r = 0.03$ och $r = -0.03$.



Figur 16.4: En sluten population ökar genom födslar (inflöde) och minskar genom dödsfall (utflöde). Inflödet och utflödet per tidsenhet är proportionella mot populationen N ,

```

import matplotlib.pyplot as plt
import numpy as np
import scipy.integrate as intgr

fig, ax = plt.subplots()
r = 0.03
f = lambda t, N: r*N
sol = intgr.solve_ivp(f,[0,50],[100000],\
                      t_eval=np.linspace(0,50,100))      # tät gridd för plottning
ax.plot(sol.t,sol.y[0],label='r > 0')    # sol.y är en matris, lösningen
                                              # ges i första raden
r = -0.03
f = lambda t, N: r*N
sol = intgr.solve_ivp(f,[0,50],[100000],\
                      t_eval=np.linspace(0,50,100))      # tät gridd för plottning
ax.plot(sol.t,sol.y[0],label='r < 0')    # sol.y är en matris, lösningen
                                              # ges i första raden
ax.set_xlabel('t',fontsize=14)
ax.set_ylabel('N(t)',fontsize=14)
ax.tick_params(labelsize=14)
ax.legend(fontsize=14)

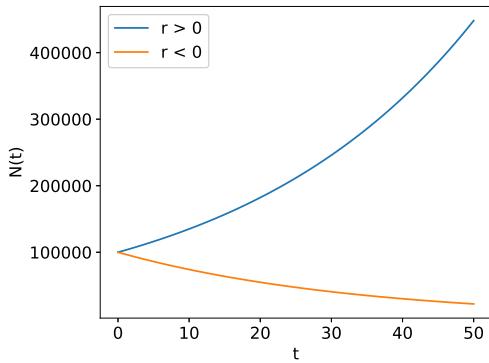
```

Motsvarande plott visas i figur 16.5. Om nettoflödet $r > 0$ växer populationen exponentiellt. Om nettoflödet $r < 0$ avtar populationen exponentiellt. Avläsning i plotten får vi att fördubblingstiden för $r = 0.03$ är cirka 23 år (kan också bestämmas med hjälp av `opt.root`, se exempel 13.1).

16.5.2 Täthetsberoende konkurrens

En svaghet med modellen i det tidigare avsnittet är att den förutsätter att ökningen av populationen kan ske obegränsat. I en mera realistisk modell måste vi ta hänsyn till miljön i form av ändlig födotillgång, boplatser etc. till slut sätter gränser för tillväxten. Man måste tänka sig att r inte är en konstant, utan en funktion av N som minskar med ökande N . Antag att miljön har en bärarkapacitet på K individer av en given population. Vi har då att $r(N) = 0$ då $N = K$ (då populationen nått sin bärarkapacitet måste nettoflödet vara noll). Dessutom kan vi anta att $r(N)$ antar sitt största värde r_{\max} för $N = 0$. Vi vet inte i detalj hur $r(N)$ ser ut men det enklaste antagandet vi kan göra är att r minskar linjärt från $r = r_{\max}$ vid $N = 0$ till $r = 0$ för $N = K$. Vi har då att

$$r(N) = r_{\max} \left(1 - \frac{N}{K}\right).$$



Figur 16.5: Population som funktion av tiden. Om nettoflödet $r > 0$ växer populationen exponentiellt. Om nettoflödet $r < 0$ avtar populationen exponentiellt.

Det är lätt att se att $r(N) = r_{\max}$ för $N = 0$ och att $r(N) = 0$ för $N = K$. Vår modell blir då

$$\frac{dN}{dt} = r_{\max} \left(1 - \frac{N}{K}\right) N.$$

En omskrivning ger

$$\frac{dN}{dt} = r_{\max} N - \underbrace{\frac{r_{\max}}{K} N^2}_{\text{täthetsberoende konkurrens}},$$

där vi tolkar den sista termen som negativ återkoppling genom konkurrens (sannolikheten att två individer i en population möts, och därmed konkurrerar, är proportionell mot $N(N - 1) \approx N^2$). System med inbyggda negativa återkopplingar är självreglerande. Differentialekvationen ovan kallas ofta för den logistiska ekvationen.

Värdet på r_{\max} beror på djurarten, och det är svårt att finna tillförlitliga värden. Det finns dock ett klart samband beroende på kroppsvekt: djurarter med mindre kroppsvekt har större värde på r_{\max} än sådana med större kroppsvekt. För däggdjur har följande approximativa relation föreslagits¹

$$r_{\max} = 0.6 V^{-0.25},$$

där V är djurets kroppsvekt i kg.

Följande program löser ekvationen i tidsintervallet [0, 20] år givet en bärarkapacitet $K = 1000$. Lösningen ges för råtta (vikt 0.250 kg) och vildsvin (vikt 100 kg) med en startpopulation på 10 individer, $N_0 = 10$.

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.integrate as intgr

fig, ax = plt.subplots()
K = 1000
N0 = 10
# råtta
```

¹Charnov, E. 1993. *Life History Invariants*. Oxford University Press.

```

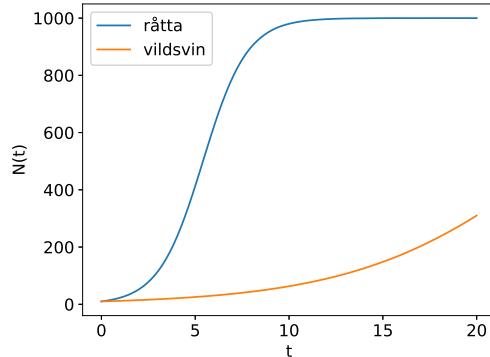
rmax = 0.6*0.250**(-0.25)
f = lambda t, N: rmax*(1 - N/K)*N
sol = intgr.solve_ivp(f,[0,20],[N0],\
                      t_eval=np.linspace(0,20,100)) # tät gridd för plottning
ax.plot(sol.t,sol.y[0],label='råtta') # sol.y är en matris, lösningen
# ges i första raden

# vildsvin
rmax = 0.6*100**(-0.25)
f = lambda t, N: rmax*(1 - N/K)*N
sol = intgr.solve_ivp(f,[0,20],[N0],\
                      t_eval=np.linspace(0,20,100)) # tät gridd för plottning
ax.plot(sol.t,sol.y[0],label='vildsvin') # sol.y är en matris, lösningen
# ges i första raden

ax.set_xlabel('t',fontsize=14)
ax.set_ylabel('N(t)',fontsize=14)
ax.tick_params(labelsize=14)
ax.legend(fontsize=14)

```

Då vi kör programmet får vi plottarna som visas i figur 16.6. Tillväxten för råtta börjar minska efter cirka 5 år. Plotten visar också den fruktansvärda hastigheten med vilken råttorna tillväxer.



Figur 16.6: Logistisk tillväxt för en population med råttor och en population med vildsvin.

16.5.3 Täthetsberoende konkurrens – bestämning av modellparametrar

År 1934 publicerade G.F. Gause boken *The struggle for existence*, Hafner Press. I boken redovisade han resultat från experiment, där han odlade encelliga urdjur (protozoer) i prover med begränsad näring. Resultatet från en av dessa experiment ges i tabell 16.2, där tiden t är i dagar och $N(t)$ är antalet urdjur per cm^3 .

Tabell 16.2: Antalet urdjur per cm^{-3} som funktion av tiden i dagar.

tid i dagar	0	1	2	3	4	5	6	7	8	9
antal per cm^3	2	10	17	29	39	63	185	258	267	392
tid i dagar	10	11	12	13	14	15	16	17	18	
antal per cm^3	510	570	650	560	575	550	480	520	500	

Vi har i det tidigare avsnittet diskuterat täthetsberoende konkurrens, och vi ska nu undersöka

hur väl den logistiska modellen

$$\frac{dN}{dt} = r_{\max} \left(1 - \frac{N}{K}\right) N$$

beskriver data.

Följande program bestämmer modellparametrarna och plottar motsvarande modelfunktion, lösning till differentialekvationen, som funktion av tiden tillsammans med experimentella data. I minstakvadratanpassningen låter vi $a_0 = r_{\max}$ och $a_1 = K$. Notera hur vi också bestämmer lösningen till differentialekvationen vid tiderna som är angivna i tabellen ($t_eval = tdata$). När vi har bestämt modellparametrarna, och ska plotta modelfunktionen, använder vi en tät gridd ($t_eval = t$). Notera även att vi inte använder $N(0) = 2$ från tabellen, utan tar $N(0)$ som en parameter a_2 , vilken bestäms vid minstakvadratanpassningen. Vi behöver startparametrar. Från tabellen ser vi att $a_2 = K \approx 500$. Lite tester ger att $a_0 = r_{\max} = 1$ verkar vara ett rimligt värde. Vi tar $a_2 = N(0) = 2$.

```

import matplotlib.pyplot as plt
import numpy as np
import scipy.integrate as intgr
import scipy.optimize as opt

# modelfunktion fås som lösning till diffekv
# lösning räknas ut för de t vi har i tabellen (tdata)
def fmodell(a,t):
    f = lambda t, N: a[0]*(1 - N/a[1])*N # högerled i diffekv
    sol = intgr.solve_ivp(f,[0,19],[a[2]],t_eval = tdata)
    return sol.y[0]

def res(a,tdata,Ndata): # residual
    return Ndata - fmodell(a,tdata)

tdata = np.arange(0,19)
Ndata = np.array([2,10,17,29,39,63,185,258,267,392,\n                 510,570,650,560,575,550,480,520,500])

a0 = [1,500,2] # startgissning
a,q = opt.leastsq(res,a0,(tdata,Ndata))
print('Parametrar:',a)
# plotta modelfunktion tillsammans med uppmätta data
fig, ax = plt.subplots()
ax.plot(tdata,Ndata,'o')

t = np.linspace(0,19,200)

f = lambda t, N: a[0]*(1 - N/a[1])*N

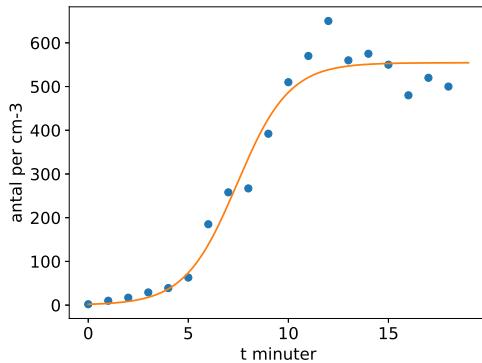
sol = intgr.solve_ivp(f,[0,19],[a[2]],t_eval = t) # tät gridd för plottnig
ax.plot(t,sol.y[0])
ax.set_xlabel('t minuter',fontsize=14)
ax.set_ylabel('antal per cm-3',fontsize=14)
ax.tick_params(labelsize=14)

```

Då vi kör programmet får vi

Parametrar: [0.76730951 554.50653553 1.83889661]

vilket innebär att $r_{\max} = 0.767$ och $K = 554$. Modellfunktionen och data visas i figur 16.7 tillsammans med data. Modellfunktionen beskriver data väl, vilket bekräftar rimligheten i den logistiska modellen.



Figur 16.7: Anpassad modellfunktion till laboratoriedata.

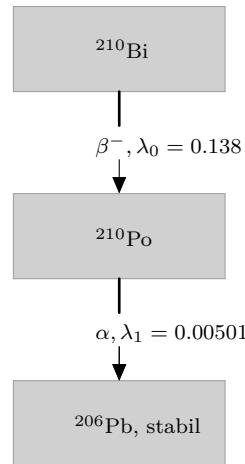
16.6 System av differentialekvationer

Ofta har vi flera systemvariabler, t.ex. $x(t)$ och $y(t)$, som beror av varandra. I dessa fall kan våra modellhypoteser (antaganden om systemet) uttryckas som en relation mellan systemvariablernas förändringshastigheter dx/dt och dy/dt och systemvariablerna själva

$$\begin{cases} \frac{dx}{dt} = f_1(t, x, y) \\ \frac{dy}{dt} = f_2(t, x, y) \end{cases}$$

Givet värden x_0 och y_0 på systemvariablerna vid tiden t_0 (när vi börjar studera systemet) kan vi beräkna approximationer till de (okända) exakta lösningarna $x(t)$ och $y(t)$ i ett antal griddpunkter på ett intervall $[t_0, t_{\text{end}}]$. Ovan har vi två systemvariabler, men i det allmänna fallet kan vi ha fler.

Exempel 16.6. I figur 16.8 har vi delar av en radioaktiv sönderfallskedja. Isotopen ^{210}Bi är radioaktiv och sönderfaller under β -strålning till ^{210}Po , vilken i sin tur sönderfaller under α -strålning till den stabila isotopen ^{206}Pb . Sönderfallskonstanten för det första sönderfallet är $\lambda_1 = 0.138$ och för det andra $\lambda_2 = 0.00501$, båda i enheten sönderfall per dygn.



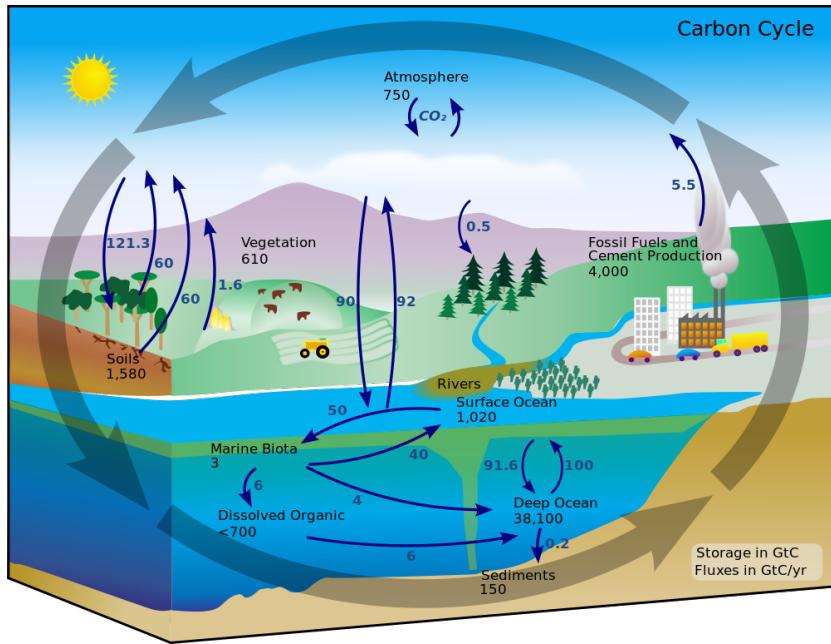
Figur 16.8: Radioaktiv sönderfallskedja. Isotopen ^{210}Bi är radioaktiv och sönderfaller under β -strålning till ^{210}Po , vilken i sin tur sönderfaller under α -strålning till den stabila isotopen ^{206}Pb .

Låt $N_0(t)$ beteckna antalet ^{210}Bi atomer och $N_1(t)$ antalet ^{210}Po atomer i ett prov. Som vi har diskuterat i exempel 16.1 är förändringen av antalet atomer ^{210}Bi per tidsenhet på grund av sönderfallet till ^{210}Po proportionellt mot antalet ^{210}Bi atomer, dvs. förändringen av antalet atomer ^{210}Bi per tidsenhet ges av $-\lambda_0 N_0$, där tecknet är negativt eftersom antalet ^{210}Bi atomer minskar. Förändringen av antalet ^{210}Po atomer per tidsenhet på grund av processen ovan är $\lambda_0 N_0$, där tecknet är positivt efter att antalet ^{210}Po atomer ökar. ^{210}Po sönderfaller i sin tur och förändringen av antalet atomer ^{210}Po per tidsenhet på grund av sönderfallet till ^{206}Pb är $-\lambda_1 N_1$, där tecknet är negativt eftersom antalet ^{210}Po atomer minskar. Detta ger sammantaget följande system av ekvationer

$$\left\{ \begin{array}{rcl} \underbrace{\frac{dN_0}{dt}}_{\substack{\text{ändring av } N_0 \\ \text{per tidsenhet}}} & = & \underbrace{-\lambda_0 N_0}_{\substack{\text{antal sönderfall} \\ \text{av } N_0 \text{ per tidsenhet}}} \\ \underbrace{\frac{dN_1}{dt}}_{\substack{\text{ändring av } N_1 \\ \text{per tidsenhet}}} & = & \underbrace{\lambda_0 N_0}_{\substack{\text{antal producerade} \\ N_1 \text{ per tidsenhet}}} - \underbrace{\lambda_1 N_1}_{\substack{\text{antal sönderfall} \\ \text{av } N_1 \text{ per tidsenhet}}} \end{array} \right.$$

□

Exempel 16.7. Kolatomer ingår i alla organiska ämnen i naturen och omsätts i samband med livsprocesserna, såväl i växter som i djur. Koldioxiden i luften tas i anspråk vid landväxternas fotosyntes. När växterna dör, återförs större delen av detta kol till luften genom bakteriell nedbrytning eller genom att trä och gräs brinner. Koldioxidutbyte äger även rum mellan atmosfär och hav. Utbytet håller luftens koldioxid i jämvikt med koldioxid löst i havet; denna är i sin tur i jämvikt med vätekarbonat- och karbonatjoner i havsvattnet, vilka jämte koldioxiden utgör grunden för det marina livet. De olika utbytena illustreras i figur 16.9, se även <https://sv.wikipedia.org/wiki/Kolcykeln>.



Figur 16.9: Utbyte av CO_2 mellan biosfären, geosfären, hydrosfären och atmosfären.

Vi ska speciellt titta på utbytet mellan atmosfär och hav. Atmosfären kan betraktas som en behållare eller pool som innehåller 750 miljarder ton kol. Genom förbränning av kol och olja tillförs atmosfären ungefär 6 miljarder ton kol per år. Atmosfären är i kontakt med havet som är en behållare eller pool som innehåller 38 000 miljarder ton kol. Om vi låter $M_0(t)$ och $M_1(t)$ beteckna kolmängden i atmosfären respektive havet i miljarder ton och $I(t)$ flödet till atmosfären från förbränning också i miljarder ton så har vi

$$\left\{ \begin{array}{l} \frac{dM_0}{dt} = \underbrace{-\frac{92}{750} M_0}_{\text{ändring av mängden kol i atmosfären per tidsenhet}} + \underbrace{\frac{90}{38000} M_1}_{\text{utflöde till havet}} + \underbrace{I}_{\text{inflöde från förbränning av kol och olja}} \\ \frac{dM_1}{dt} = \underbrace{\frac{92}{750} M_0}_{\text{ändring av mängden kol i havet per tidsenhet}} - \underbrace{\frac{90}{38000} M_1}_{\text{utflöde till atmosfären}} \end{array} \right.$$

□

16.7 Lösning av system av differentialekvationer i Python

Den inbyggda funktionen `intgr.solve_ivp` från avsnitt 16.3 kan även användas för att lösa system av differentialekvationer. Högerledet och begynnelsevärdena måste nu ges som listor, men i övrigt är där inga förändringar i lösningsförfarandet.

Exempel 16.8. Vi har ekvationssystemet

$$\left\{ \begin{array}{l} \frac{dN_0}{dt} = -\lambda_0 N_0 \\ \frac{dN_1}{dt} = \lambda_0 N_0 - \lambda_1 N_1 \end{array} \right.$$

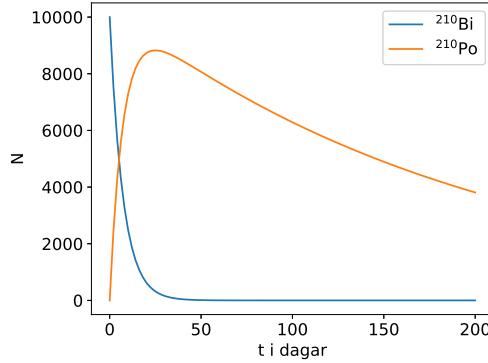
där $\lambda_0 = 0.138$ och $\lambda_1 = 0.00501$. Ekvationssystemet beskriver hur ^{210}Bi sönderfaller till ^{210}Po som i sin tur sönderfaller till ^{206}Pb . För att lösa systemet i tidsintervallet $[0, 200]$ med startvärdena $N_0 = 10000$ och $N_1 = 0$ och plotta lösningarna ger vi kommandona

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.integrate as intgr

lam0 = 0.138
lam1 = 0.00501
f = lambda t,N : [-lam0*N[0],
                    lam0*N[0] - lam1*N[1]]

sol = intgr.solve_ivp(f,[0,200],[10000,0],t_eval=np.linspace(0,200,100))
fig, ax = plt.subplots()
ax.plot(sol.t,sol.y[0],label=r"$^{210}\text{Bi}$") # N0 första raden, sol.y[0]
ax.plot(sol.t,sol.y[1],label=r"$^{210}\text{Po}$") # N1 andra raden, sol.y[1]
ax.set_xlabel("t i dagar",fontsize=14)
ax.set_ylabel("N",fontsize=14)
ax.tick_params(labelsize=14)
ax.legend(fontsize=14)
```

Plotten visas i figur 16.10. Från plotten ser vi att antalet ^{210}Bi atomer, N_0 , hela tiden minskar och fyller på mängden ^{210}Po atomer, N_1 , som ökar till en början. Efter ett tag tar sönderfallet av ^{210}Po till ^{206}Pb över och mängden ^{210}Po minskar. Grafisk avläsning ger att antalet ^{210}Po atomer är som störst vid tiden $t = 25$. \square



Figur 16.10: Antalet ^{210}Bi och ^{210}Po atomer som funktion av tiden i dagar.

Exempel 16.9. Ekvationssystemet som styr flödet av kol mellan atmosfären, pool med mängd $M_0(t)$ (miljarder ton), och havet, pool med mängden $M_1(t)$ (miljarder ton) ges av

$$\begin{cases} \frac{dM_0}{dt} = -\frac{92}{750}M_0 + \frac{90}{38000}M_1 + I \\ \frac{dM_1}{dt} = \frac{92}{750}M_0 - \frac{90}{38000}M_1 \end{cases}$$

där I är mängden kol (miljarder ton) som tillförs atmosfären via förbränning av kol och olja. Följande program löser ekvationssystemet i tidsintervallet $[0, 50]$ med de initiala värdena $M_0(0) = 750$ och $M_1(0) = 38000$ och plottar mängden kol i atmosfären. Vi löser i två fall. I fall ett antar

att all förbränning av kol och olja har upphört, dvs. vi tar $I = 0$. I fall två antar vi att vi fortsätter att förbränna kol och olja i den takt som vi gör idag, dvs. $I = 6$ (miljarder ton).

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.integrate as intgr

fig, ax = plt.subplots()

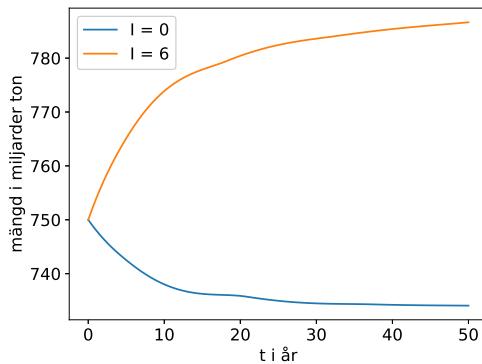
# I = 0
f = lambda t,M : [-92/750*M[0] + 90/38000*M[1],
                    92/750*M[0] - 90/38000*M[1]]
sol = intgr.solve_ivp(f,[0,50],[750,38000],t_eval=np.linspace(0,50,100))
ax.plot(sol.t,sol.y[0],label='I = 0') # plotta mängd i atmosfären

# I = 6
f = lambda t,M : [-92/750*M[0] + 90/38000*M[1] + 6,
                    92/750*M[0] - 90/38000*M[1]]
sol = intgr.solve_ivp(f,[0,50],[750,38000],t_eval=np.linspace(0,50,100))
ax.plot(sol.t,sol.y[0],label='I = 6') # plotta mängd i atmosfären

ax.set_xlabel("t i år", fontsize=14)
ax.set_ylabel("mängd i miljarder ton", fontsize=14)
ax.tick_params(labelsize=14)
ax.legend(fontsize=14)
```

Plotten visas i figur 16.11. Utan förbränning skulle mängden kol i atmosfären gå ned och stabilisera sig på en nivå 734 miljarder ton. Den lilla 'bumpen' i kurvan beror på att den numeriska lösningen inte är så noggrann. Man kan hantera detta genom att ändra optioner i kommandot `intgr.solve_ivp` som styr noggrannheten. Vi går inte in på detta här.

Naturligtvis är kolbalansen oerhört mycket mera komplicerad än vad vi har gett sken av i denna enkla modell. En realistisk modell som går att köra med Python ges i Suzuki, M. m.fl. 1993 *Ecological Modelling* Volume 70, Issues 3–4, sid. 161–194. Med hjälp av denna modell kan vi mycket noggrant förutsäga mängden CO₂ i atmosfären under olika utsläppsscenerier. Aktuell utsläppsdata av CO₂ finns här <https://ourworldindata.org/co2-emissions>. För att relatera till vår modell, som anger mängden kol och inte mängden koldioxid, har vi att 1 g kol motsvarar 3.6 g koldioxid. □



Figur 16.11: *Mängden kol (i miljarder ton) i atmosfären som funktion av tiden t i år. Den ena kurvan är för fallet att vi slutar förbränna kol och olja så att I = 0. Den andra kurvan är för fallet att vi fortsätter att förbränna i den takt vi gör idag, dvs. I = 6 miljarder ton.*

16.8 Tillämpning: smittspridning

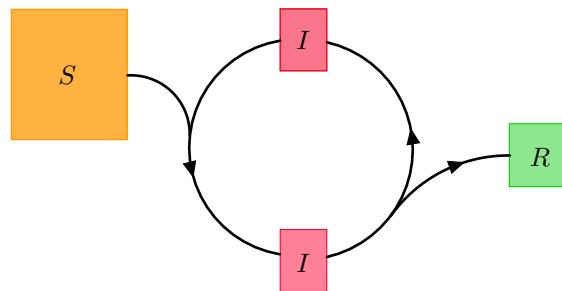
Spridning av smittsamma sjukdomar, t.ex. påssjuka, mässling, polio, kikhosta, COVID-19, är ett allvarligt hot mot folkhälsan. Vaccination är ett av 1900-talets viktigaste medicinska framsteg och helt avgörande för att hindra spridning av smittsamma sjukdomar i befolkningen och därmed befrämja folkhälsan. Vaccination räddar årligen tusentals liv, bara i Sverige.² Som en viktig tillämpning av dynamiska modeller ska vi titta på den så kallade SIR-modellen för spridning av sjukdomar.

16.8.1 SIR-modellen

I SIR-modellen delar vi upp populationen av N individer i tre grupper, vilka varierar med tiden t :

- $S(t)$ är de som är mottagliga (eng. susceptible) för sjukdomen men ännu inte smittade
- $I(t)$ är de som är infekterade (eng. infected) av sjukdomen
- $R(t)$ är de som har återhämtat sig (eng. recovered) från sjukdomen och som nu har immunitet.

SIR-modellen beskriver förändringshastigheten för populationerna i var och en av grupperna i termer av två parametrar β och γ . Antalet möten mellan en mottaglig person och en infekterad person är proportionell mot SI/N (jämför avsnitt 16.5.2). Sannolikheten för att en mottaglig då skall bli infekterad är då $\beta SI/N$, där β beskriver sjukdomens effektiva kontakthastighet (sjukdomens smittsamhet). Antalet återhämtade per tidsenhet är proportionell mot I och ges av γI , där parametern γ är återhämtningshastigheten. $1/\gamma$ är då tiden i medeltal som en infekterad kan föra sjukdomen vidare.



Figur 16.12: Smittspridning enligt SIR-modellen. Genom möte mellan mottagliga (Susceptible), S och infekterade (Infected), I , minskar antalet mottagliga samtidigt som antalet infekterade ökar med samma antal. Antalet infekterade minskar genom att folk återhämtar sig (Recover), varvid antalet återhämtade R hela tiden ökar.

Differentialekvationerna som beskriver modellen är

$$\left\{ \begin{array}{l} \frac{dS}{dt} = -\beta \frac{SI}{N} \\ \frac{dI}{dt} = \beta \frac{SI}{N} - \gamma I \\ \frac{dR}{dt} = \gamma I \end{array} \right.$$

och härleddes av Kermack och McKendrick 1927. Följande program implementerar modellen och tillåter oss att köra simuleringar, där vi ändrar på parametrarna β och γ .

²Tack till Prof. Karl Ekdahl vid European Centre for Disease Prevention and Control för diskussioner om smittspridning, vaccination och flockimmunitet.

```

import numpy as np
import scipy.integrate as intgr
import matplotlib.pyplot as plt

# Populationsstorlek
N = 1000
# Initialt antal smittade, I0, och återhämtade, R0, personer
I0 = 1
R0 = 0
# Alla övriga är mottagliga så initialt har vi
S0 = N - I0 - R0

beta = float(input('Ge kontakthastighet '))
recovertime = float(input('Återhämtningstid i dagar '))
gamma = 1/recovertime

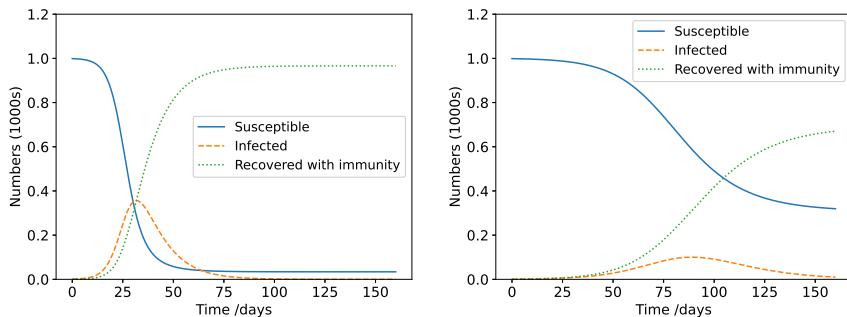
# Modellens differentialekvationer
f = lambda t, y: [-beta*y[0]*y[1]/N,
                    beta*y[0]*y[1]/N - gamma*y[1],
                    gamma*y[1]]

# Tidsgridd i dagar
t = np.linspace(0, 160, 160)
# Integrera ekvationerna på tidsgridden.
sol = intgr.solve_ivp(f,[0,160],[S0,I0,R0],t_eval = t)

# Plotta S(t), I(t) och R(t)
fig, ax = plt.subplots()
ax.plot(sol.t,sol.y[0]/1000,label='Susceptible')
ax.plot(sol.t,sol.y[1]/1000,'--',label='Infected')
ax.plot(sol.t,sol.y[2]/1000,':',label='Recovered with immunity')
ax.set_xlabel('Time /days',fontsize=14)
ax.set_ylabel('Numbers (1000s)',fontsize=14)
ax.set_yscale(0,1.2)
ax.legend(fontsize=14)
ax.tick_params(labelsize=14)

```

Modellen, körd med $\beta = 0.35$ och $\gamma = 1/10$ (genomsnittlig återhämtningstid 10 dagar), ger plotten till vänster i figur 16.13. Vi ser att i stort sett hela populationen blir smittad och får sjukdomen. Om vi till exempel genom social distansering får ner kontakthastigheten blir förloppet ett helt annat. Modellen, körd med $\beta = 0.17$ och $\gamma = 1/10$, ger plotten till höger i figur 16.13. Nu är förloppet lugnare och bara 40 % av populationen blir smittade innan sjukdomen dör ut.



Figur 16.13: Antal mottagliga, smittade och återhämtade som funktion av tiden. Till vänster har vi kurvorna för $\beta = 0.35$ och till höger för $\beta = 0.17$. I båda fallen är $\gamma = 1/10$.

16.8.2 SIR-modellen med vaccination

SIRV-modellen (Susceptible-Infectious-Recovered-Vaccinated model) är en utvidgning av SIR-modellen som tar hänsyn till vaccination av den mottagliga gruppen av populationen. Modellen baseras på följande differentialekvationer

$$\begin{cases} \frac{dS}{dt} = -\beta \frac{SI}{N} - vS \\ \frac{dI}{dt} = \beta \frac{SI}{N} - \gamma I \\ \frac{dR}{dt} = \gamma I \\ \frac{dV}{dt} = vS \end{cases}$$

där $V(t)$ är antalet vaccinerade. β , γ och v är effektiva kontakthastigheten, återhämtandehastigheten och vaccinationshastigheten, dvs. andelen mottagliga som vaccineras per tidsenhet. Följande program löser ekvationerna med hänsyn taget också till vaccination.

```
import numpy as np
import scipy.integrate as integr
import matplotlib.pyplot as plt

# Populationsstorlek
N = 1000

beta = float(input('Ge kontakthastighet '))
recovertime = float(input('Återhämtningstid i dagar '))
gamma = 1/recovertime
percent = float(input('Procent av befolkningen som är vaccinerade '))
v = float(input('Vaccinationshastighet '))

# Initialt antal smittade, I0, återhämtade, R0,
# och vaccinated, V0, personer
I0 = 1
R0 = 0
V0 = percent*0.01*N
# Alla övriga är mottagliga så initialt har vi
S0 = N - I0 - R0 - V0

# Skapa en array för att spara de olika populationerna över tiden
S = np.zeros(151)
I = np.zeros(151)
R = np.zeros(151)
V = np.zeros(151)

# Initiera första värdena
S[0] = S0
I[0] = I0
R[0] = R0
V[0] = V0

# Lösa ekvationerna över tiden
for t in range(1, 151):
    S[t], I[t], R[t], V[t] = integr.odeint(lambda t, y: [v*y[3] - beta*y[0]*y[1]/N, beta*y[0]*y[1]/N - gamma*y[1], gamma*y[1], -v*y[3]], y=[S[t-1], I[t-1], R[t-1], V[t-1]], t=t)[0]

# Plotta resultaten
plt.plot(S, 'blue', label='Susceptible')
plt.plot(I, 'orange', label='Infected')
plt.plot(R, 'green', label='Recovered with immunity')
plt.plot(V, 'red', label='Vaccinated')
plt.legend()
plt.xlabel('Time /days')
plt.ylabel('Numbers (1000s)')
plt.show()
```

```

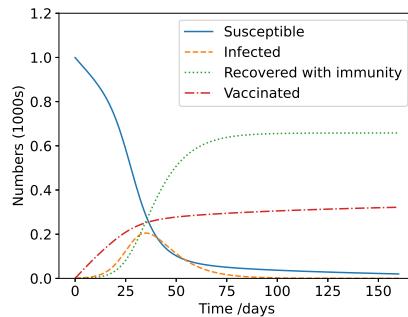
# Modellens differentialekvationer
f = lambda t, y: [-beta*y[0]*y[1]/N - v*y[0],
                    beta*y[0]*y[1]/N - gamma*y[1],
                    gamma*y[1],
                    v*y[0]]

# Tidsgridd i dagar
t = np.linspace(0, 160, 160)
# Integrera ekvationerna på tidsgridden.
sol = intgr.solve_ivp(f,[0,160],[S0,I0,R0,V0],t_eval = t)

# Plotta S(t), I(t), R(t) och V(t)
fig, ax = plt.subplots()
ax.plot(sol.t, sol.y[0]/1000,label='Susceptible')
ax.plot(sol.t, sol.y[1]/1000,'--',label='Infected')
ax.plot(sol.t, sol.y[2]/1000,':',label='Recovered with immunity')
ax.plot(sol.t, sol.y[3]/1000,'-.',label='Vaccinated')
ax.set_xlabel('Time /days',fontsize=14)
ax.set_ylabel('Numbers (1000s)',fontsize=14)
ax.set_ylim(0,1.2)
ax.legend(fontsize=14)
ax.tick_params(labelsize=14)

```

Modellen, körd med $\beta = 0.35$, $\gamma = 1/10$ (genomsnittlig återhämtningstid 10 dagar), vaccinationshastighet $v = 0.01$ (motsvarar att 1 % av de mottagliga vaccineras per dag) och att ingen i befolkningen är vaccinerad när infektionen starta ger plotten i figur 16.14. Om man googlar "SIR-model, COVID, vaccination" ser man hur många vetenskapliga artiklar som bygger på modeller av ovanstående typ. Givet all uppmärksamhet i media förstår man också betydelsen av simuleringar som underlag för att fatta stora och viktiga beslut kring sjukvård, erbjudande av vaccinationer osv.



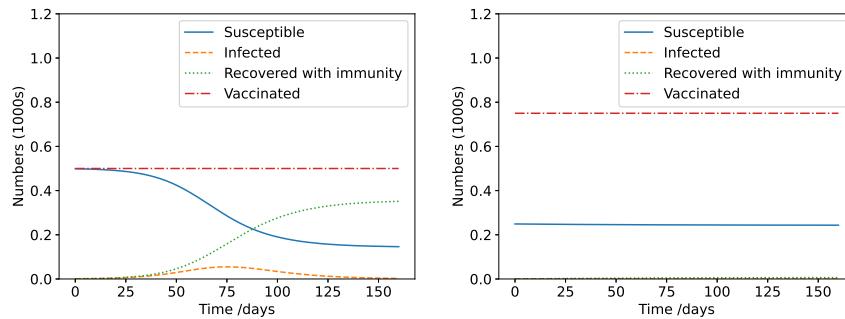
Figur 16.14: Antal mottagliga, smittade, återhämtade och vaccinerade som funktion av tiden då vi kör modellen med $\beta = 0.35$, $\gamma = 1/10$ och $v = 0.01$.

16.8.3 Flockimmunitet

Flockimmunitet är ett viktigt begrepp inom infektionsepidiologin. Flockimmunitet är ett indirekt skydd mot smittsamma sjukdomar, som uppstår när en stor andel av en population har utvecklat immunitet mot en sjukdom och därigenom skyddar även icke-immuna individer. I en population, där en stor andel av individerna är immuna, är det troligare att spridningen hindras, därför att spridningen stoppas när smittan når en immun individ. Ju större den immuna

andelen är, desto mindre risk att icke-immuna individer kommer i kontakt med en smittad. En enskild person kan bli immun genom att återhämta sig från ett naturligt insjuknande eller genom vaccination. En del individer kan inte bli immuna, av medicinska skäl, och för denna grupp är flockimmuniteten ett viktigt skydd. När väl en viss tröskelnivå har nåtts, kan flockimmuniteten bidra till att smittan gradvis utrotas från populationen. Om detta sker globalt, kan det resultera i att antalet smittade går ner till noll, alltså att sjukdomen är utrotad. Denna metod användes för att utrota smittkopper – det sista kända fallet diagnostiseras 1977 – och för lokal utrotning av flera sjukdomar. I Sverige erbjuds barn och ungdomar vaccinationer som ger skydd mot elva sjukdomar: rotavirusinfektion, difteri, stelkramph, kikhusta, polio, infektioner orsakade av *Haemophilus influenzae* typ b, allvarlig sjukdom orsakad av pneumokocker, mässling, påssjuka, röda hund och humant papillomvirus (HPV). Folkhälsomyndigheten rekommenderar dessutom vaccination mot hepatit B till alla spädbarn.

Vi ska studera flockimmunitet med hjälp av vårt program. Då vi kör programmet med $\beta = 0.35$, $\gamma = 1/10$, $v = 0$ och en vaccinationstäckning på 50 % får vi plotten till vänster i figur 16.15. Då vi kör programmet med $\beta = 0.35$, $\gamma = 1/10$, $v = 0$ och en vaccinationstäckning på 75 % får vi plotten till höger i figur 16.15. I det senare fallet har vaccinationstäckningen på 75 % gett flockimmunitet och smittan sprider sig inte.³



Figur 16.15: Illustration av flockimmunitet. Till vänster visas smittförloppet vid en vaccinationstäckning på 50 %. Till vänster visas smittförloppet vid en vaccinationstäckning på 75 %. I det senare fallet har vi flockimmunitet och smittan sprider sig inte.

³Flockimmunitet är inte helt tillämpbart på COVID, men väl på de andra uppräknade sjukdomarna.

Del III

Modellering i perspektiv

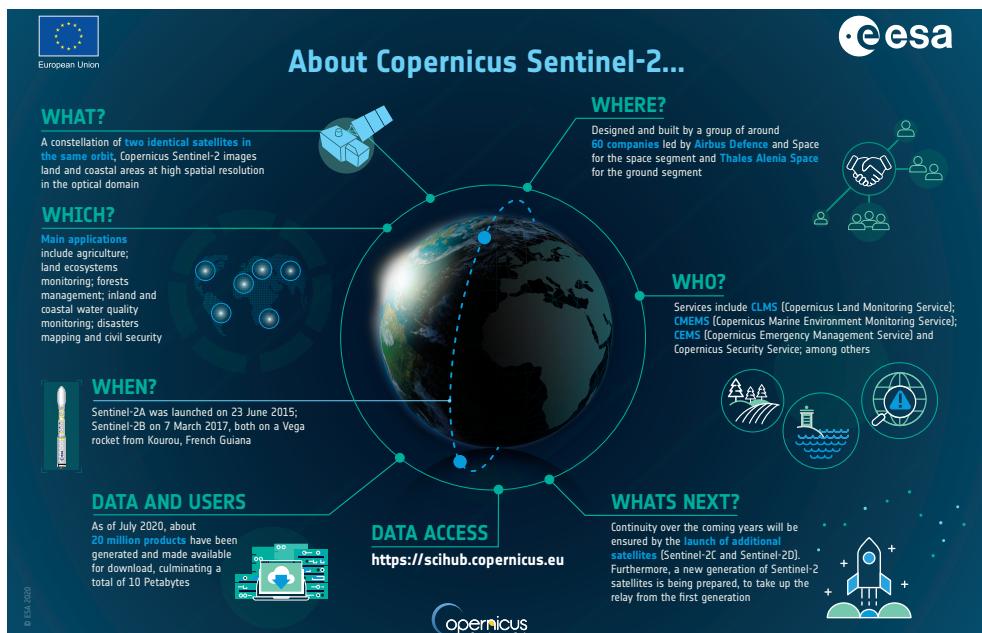
Kapitel 17

Modellering för att rädda och bevara planeten

I slutet av kursen känns det riktigt att sätta modellering i perspektiv. Vi har, inom ramen för kursen, arbetat med modellering och databehandling i det lilla. Liknande metoder och angreppssätt används också i det stora. Modellering är avgörande för att hantera frågorna kring klimat, miljö och bevarandet av vår planet. Modellering gör det möjligt att titta in i framtiden och presentera olika scenarier. Dessa ligger till grund för beslut, politiska och ekonomiska, hur vi ska utforma våra gemensamma framtid för att mildra effekterna av klimatförändringarna och bevara planeten.

17.1 Copernicus och Sentinel

Copernicus är EU:s miljöprogram <https://www.copernicus.eu/sv/om-copernicus/copernicus-i-korthet>. Kärnan i Copernicus är de jordobserverande Sentinel-satelliterna, vilka samlar in enorma mängder data relaterat till klimat och miljö <https://www.copernicus.eu/en/about-copernicus/infrastructure-overview/discover-our-satellites>. I figur 17.1 ges information om Sentinel-2, som används bland annat för landmonitorering relaterat till klimatförändringar.

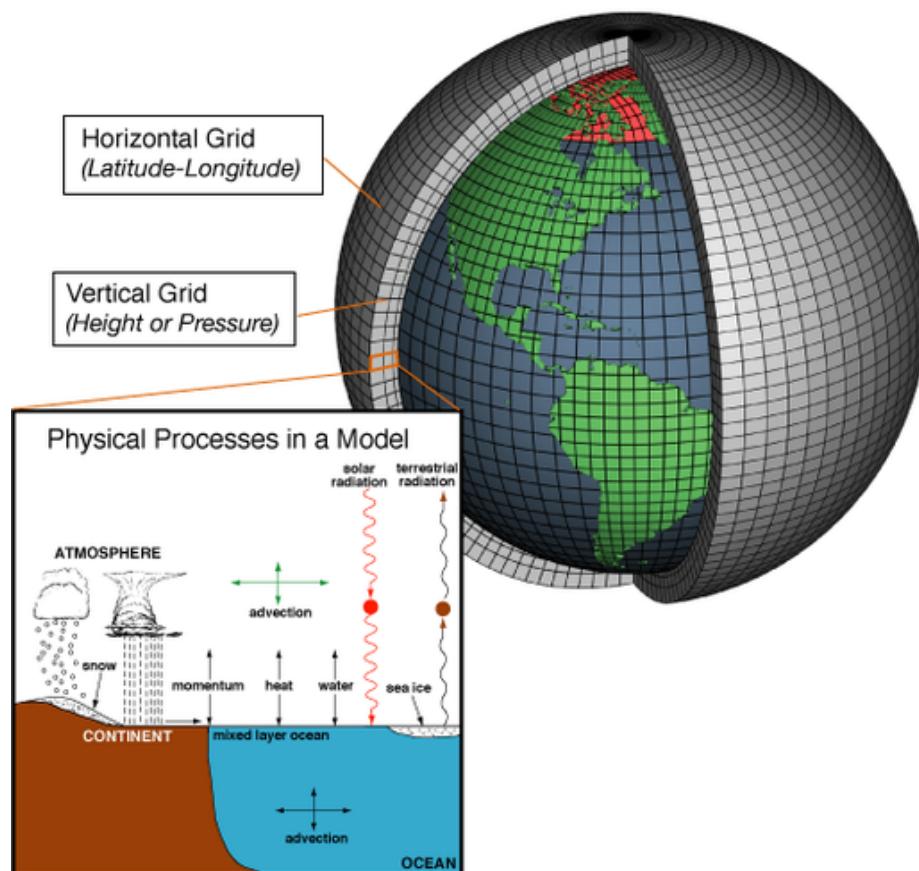


Figur 17.1: Information om den jordobserverande satelliten Sentinel-2.

Copernicus samlar även in information från fältbaserade system, t.ex. markstationer, som levererar data från ett stort antal sensorer på land, till havs och i luften. Data, som sträcker sig årtionde tillbaka, behandlas och analyseras och tillgängliggörs, vilket gör det möjligt att upptäcka mönster, se förändringar, skapa prognoser och presentera framtidsscenarier.

17.2 Klimatmodeller

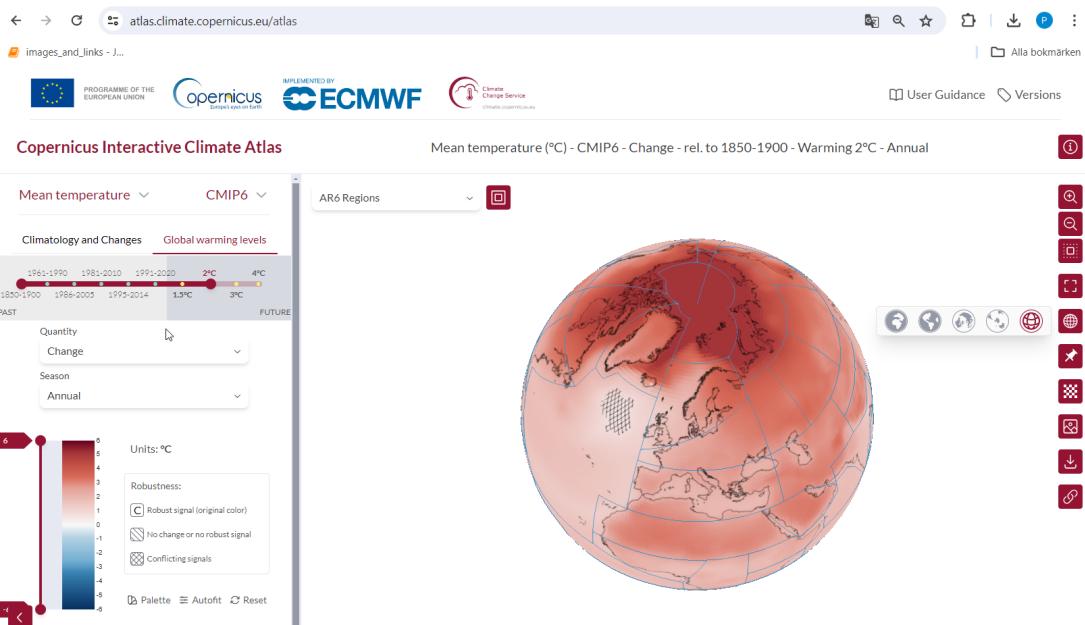
Det är av särskilt stor vikt att skapa prognoser för klimatet. Sådana prognoser är baserade på klimatmodeller https://en.wikipedia.org/wiki/Climate_model. Klimatmodeller är mycket komplexa och datorintensiva. Klimatmodeller är system av differentialekvationer baserade på lagar från fysiken, kemin och hydrodynamiken, där en mängd återkopplingsmekanismer och deras styrka bestäms (eller skattas) utifrån insamlad data, från bland annat Copernicus. För att köra modellen, dvs. göra en simulering, delar man in planeten i en tredimensionell grid (celler). Man applicerar differentialekvationerna och beräknar vindar, värme- och strålningstransport, relativ fuktighet, yt-hydrologi i varje cell och kopplar sedan samman närliggande celler. Principerna för modelleringen visas i figur 17.2.



Figur 17.2: Klimatmodeller delar in planeten i celler. I varje cell applicerar man system av differentialekvationer och beräknar en mängd variabler. Närbeliggande celler kopplas sedan samman. Bilden tagen från https://celebrating200years.noaa.gov/breakthroughs/climate_model/#model.

17.3 Copernicus interaktiva klimatatlas

Det sista steget i modellering är att använda resultaten från simuleringar för att fatta informerade beslut, t.ex. politiska eller ekonomiska. Som ett led i detta har Copernicus släppt en interaktiv klimatatlas (länk till atlasen finns här). Atlasen presenterar de viktigaste klimatvariablerna (och deras extremer) framåt i tiden baserat på olika relevanta scenarier, t.ex. 1.5 °C, 2 °C, 3 °C och 4 °C globala uppvärmningsnivåer. Klimatvariablerna kan presenteras månadsvis och för olika regioner, men kan också aggregeras. Atlasen har som mål att hjälpa användare att utveckla evidensbaserade riktlinjer och strategier för arbetet med att mildra effekterna av, och anpassa sin verksamhet till, den globala uppvärmningen. I figur 17.3 visas en ögonblicksbild av atlasen.



Figur 17.3: *Copernicus klimatatlas gör det bland annat möjligt att se de lokala effekterna av en höjning av jordens medeltemperatur med 2 °C. Klimatatlasen är resultatet av omfattande modellering och simulering och representerar den bästa kunskapen vi har för tillfället.*

17.4 Modellerings av den gröna ekonomin

Klimatmodelleringarna visar på nödvändigheten av riktlinjer och strategier för arbetet med att mildra effekterna av de pågående klimatförändringarna. De visar också på det akuta behovet av att ställa om ekonomin till en fossilfri och cirkulär grön ekonomi. För att bidra till detta har FN:s miljöorgan utvecklat ett ramverk för integrerad modellerings av den gröna ekonomin (Integrated Green Economy Modelling (IGEM) Framework) (länk finns här). Ramverket erbjuder en möjlighet att integrera tre modelleringsstekniker (system dynamik, jämviktsmodeller och in- och utgående sociala redovisningsmatriser) för att skatta effekterna av gröna omställningar av ekonomi och produktion. Denna typ av modellerings är baserad på det matematiska matrisbegreppet, och infördes av Nobelpristagaren Wassily Leontief. Om du är intresserad, följ länken här.

17.5 Engagera dig

Kunskap är makt. Fortsätt att utbilda dig och engagera dig för en bättre framtid, där vi tar hand om varandra och vår planet.