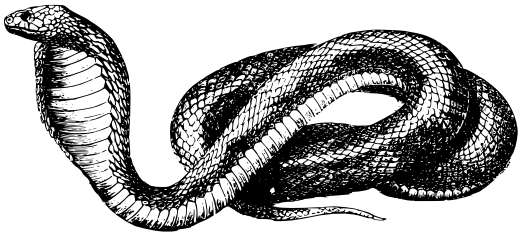

KOMPENDIUM

PROGRAMMERING MED PYTHON

Christian Åberg

November 2020

Innehåll

Kapitel 1

Introduktion

Det är svårt att lära sig programmera! Till en början måste man lära sig en mängd språkregler och koncept som inte alltid känns helt intuitiva:

- Variabler
- Datatyper
- If-satser
- While-loopar
- Selektion
- Iteration

Listan kan göras lång på nya begrepp och koncept man måste ta till sig! Att växa som programmerare tar tid. Tålamodet sätts ofta på prov när man gång på gång stöter på syntaktiska konstigheter, logiska felaktigheter och koncept som är svåra att ta till sig. De flesta programmerare är dock överens om att när man väl tagit sig över den initiala tröskeln som programmerare så blir hantverket bara roligare och roligare. Man tänker ofta tillbaka till sina tidigare dagar som programmerare och funderar för sig själv:

> *Hur kunde jag tycka att de här var så svårt?!*

En värld öppnar sig med projekt och appar man kan bygga. Arbetsgivare skriker efter din kompetens!

Låt din nyfikenhet och kreativitet blomstra. Kommer du på ett litet kul projekt du skulle kunna bygga? Bygg det! Stirra dig inte blind på allt du inte kan. Låt dig istället glädjas för allt du redan lärt dig! Hitta motivation i det för att fortsätta framåt!

1.1 Upphovsrättslig notis

Kapitel 2

Variabler och datatyper

För nästan samtliga programmeringstekniska koncept behöver vi kunna spara undan data för senare användning. I det här kapitlet introducerar vi variabler, datatyper och hur vi under körning hämtar data från användaren.

2.1 Variabler

Variabler har ett namn, ett värde och definieras med likhetstecken:

```
MinVariabel = "Christian"
```

Man kan referera till variabler vid bland annat definitionen av nya variabler och som parametrar till funktioner:

```
MinSiffra = 18
EnNySiffra = MinSiffra + 5
print(EnNySiffra)
# OUTPUT: 23
```

2.2 Datatyper

Datan man lagrar i variabler kan vara av olika typer. Tar man hänsyn till egendefinierade klasser finns det i princip en oändlig mängd olika datatyper i Python, eftersom utvecklare kan definiera sina egna efter behov.

Som tur är fokuserar vi endast på de mest grundläggande datatyperna i den här kursen (de som följer med programspråket). Men vi kommer mot slutet av kursen även utöka funktionaliteten av våra script genom att importera datatyper från tredjepartsutvecklare.

Sättet du matar in data i din källkod styr vilken datatyp den är:

```
# Tilldelning av heltal
a = 5

# Tilldelning av sträng
b = "5"

# Tilldelning av sträng
c = "a"

# Tilldelning av värdet av annan variabel
d = a

# Vad skrivs ut här?
print(d)
```

Det är viktigt att du förstår och ser skillnaden på olika datatyper och förstår varför de inte alltid är kompatibla:

```
a = 5
b = "5"
c = a + b # Varför kraschar programmet?
```

Strängar

Texter (ord och meningar) lagras i datatypen sträng (string på engelska). Strängar definieras med hjälp av citattecken:

```
en_mening = "Detta är en sträng"
print(en_mening)
```

Glömmer du citattecknet definierar du inte längre en sträng. Då refererar du istället till ett variabelnamn:

```
Christian=1337
MittNamn=Christian
print(MittNamn)
# OUTPUT: 1337
```

Strängar har några hjälpsamma metoder som är inbyggda i datatypen. Tänk på metoder som ett verktyg som kan användas för att modifiera data på olika sätt:

```
mening="alla ord börjar på stor bokstav"
ny_variabel=mening.title()
print(ny_variabel)
# OUTPUT: Alla Ord Börjar På Stor Bokstav
```

Det är omöjligt att intuitivt känna till alla metoder som är inbyggda i de olika datatyperna. Vill man ta reda på alla metoder som finns tillgängliga för Pythons inbyggda datatyper kan man referera till den officiella dokumentationen.

Google fungerar också bra. Vill du exempelvis ta reda på vilken metod som gör om samtliga bokstäver i en sträng till versaler så kan du testa googla på följande:

how to capitalize all letters in a string python

Precis som nummer så har strängar ett antal operationer. För att slå ihop två strängar kan man använda sig av addition:

```
a = 'Chris'
b = 'tian'
c = ' '
d = 'gillar programmering!'
text = a + b + c + d
print(text)
```

Istället för att statiskt definiera alla strängar direkt i källkoden så kanske man vill att användaren av ett program ska mata in en sträng under körning. Detta kan man göra med hjälp av funktionen **input()**:

```
print('Vad heter du?')
namn = input()
svar = 'Trevligt att träffas ' + namn + '!'
print(svar)
```

Nummer

Det finns tre numeriska datatyper i Python. I den här kursen tittar vi på heltal och flyttal:

```
# Heltal
a = 2
# Flyttal
b = 2.0
```

Heltal är alla nummer utan decimal-tecken medan flyttal är alla nummer med decimaltecken. Trots att heltal och flyttal är olika datatyper så är dessa nära besläktade med varandra. Det går exempelvis att utföra jämförelser mellan heltal och flyttal:

```
# Heltal
a = 2

# Flyttal
b = 2.0

if a == b:
    print('a har samma värde som b')
else:
    print('a har inte samma värde som b')
```

Det är också möjligt att utföra matematiska operationer mellan heltal och flyttal:

```
# Heltal
a = 2

# Flyttal
b = 2.0

c = a + b

print(c)
# OUTPUT: 4.0
```

Inkluderar du ett flyttal i en ekvation kommer resultatet också alltid bli ett flyttal:

```
a = 1 + 1.0

print(a)
# OUTPUT: 2.0
```

Du kan omvandla ett flyttal till ett heltal med funktionen **int()**:

```
a = 1 + 1.0
b = int(a)

print(b)
# OUTPUT: 2
```

Samma funktion kan också användas för att omvandla en sträng till ett heltal:

```
a = 5
b = "5"

c = a + int(b)

print(c)
# OUTPUT: 10
```

2.3 Användarinmatning

Vi har redan nämnt att man med funktionen **input()** kan låta användare mata in data med sitt tangentbord under körning. Men hur fungerar egentligen funktionen?

Funktionen kommer lyssna efter data från tangentbordet. När funktionen upptäcker att användaren trycker ner enter-tangenten kommer funktionen sluta lyssna och istället returnera den inlästa datan som en sträng.

```
print('Vad␣heter␣du?')
name = input()
print('Hallå␣' + name + '!')
```

Funktioner som returnerar data kan användas för att sätta värden på variabler vilket visas i exemplet ovan. Resultatet från **input** kommer lagras i variabeln **name**.

Eftersom **input** returnerar data av typen sträng kan vi utföra samtliga operationer som strängar stödjer på den returnerade datan. I exemplet som följer utför vi strängaddition på datan vi hämtade från användaren:

```
print('Vad␣är␣ditt␣förnamn?')
first_name = input()

print('Vad␣är␣ditt␣efternamn?')
last_name = input()

full_name = first_name + '␣' + last_name

print('Hallå␣' + full_name + '!')
```

Prompt

Utgå från följande script:

```
print('Ange␣namn:')
name = input()
print('Hallå', name)
```

Kommer användarinmatningen göras på samma rad som strängen *'Ange namn:'* eller på en ny rad efter strängen? Standardbeteendet för **print()** är att automatiskt skriva ut en ny rad direkt efter strängen som funktionen skriver ut:

```
Ange namn:
Gunilla
Hallå Gunilla
```

Vill man att användarinmatningen görs på samma rad som strängen *'Ange namn:'* kan man ange strängen som argument till *input()*:

```
name = input('Ange␣namn:␣')
print('Hallå', name)
```

Funktionen **input()** kommer först skriva ut strängen vi angav som argument (utan att lägga till en ny rad) och därefter lyssna efter input från användaren. Resultatet från detta blir att användarinmatningen görs direkt efter strängen *'Ange namn: '*:

```
Ange namn: Ida
Hallå Ida
```

Datatypsomvandling av användarinmatad data

Låt säga att vi försöker skapa ett program som skriver ut summan av två användarinamatade heltal:

```
a = input('a␣=␣')
b = input('b␣=␣')

c = a + b

print(a, '+', b, '=', c)
```

Detta fungerar inte riktigt som planerat:

```
a = 13
b = 37
13 + 37 = 1337
```

Summan av 13 och 37 är inte lika med 1337. Fundera ett tag på varför Python inte verkar förstå sig på enkel aritmetik.

Datan som **input()** returnerar är av typen sträng. Additionen som utförs i raden **c = a + b** är således en vanlig sträng-addition. Vill vi utföra addition enligt aritmetikens regler behöver vi först omvandla den användarinmatade datan till heltal. Detta kan vi göra samtidigt som vi beräknar c:

```
a = input('a␣=␣')
b = input('b␣=␣')

c = int(a) + int(b)

print(a, '+', b, '=', c)
```

Ännu snyggare är om vi utför omvandlingen i samband med att vi hämtar datan från användaren:

```
a = int(input('a␣=␣'))
b = int(input('b␣=␣'))

c = a + b

print(a, '+', b, '=', c)
```

Nu när vi utför en datatypsomvanling på den inmatade datan fungerar scriptet enligt kraven:

```
a = 13
b = 37
13 + 37 = 50
```

2.4 Övningsuppgifter

Övningsuppgift 2.1. Modifiera print-satsen i följande exempel så att hela strängen skrivs ut med stora bokstäver:

```
citat="datatyper␣har␣inbyggda␣metoder"
print(citat)
```

Övningsuppgift 2.2. Skapa ett program som ber användaren mata in ett flyttal. Programmet ska avrunda flyttalet till närmaste heltal och presentera talet för användaren.

Övningsuppgift 2.3. Konstruera ett program där användaren i dialog med datorn ger sitt för- och efternamn. Datorn ska avsluta körningen med att hälsa på användaren enligt exemplet nedan:

```
dator> Hallå!
dator> Vad är ditt förnamn?
du>    Johan
dator> Vad är ditt efternamn?
du>    Svensson
dator> Trevligt att träffas Johan Svensson!
```

Övningsuppgift 2.4. Skapa ett program som frågar om användarens ålder. Programmet ska svara användaren inom hur många år användaren uppnår myndig ålder (18 år).

```
Hur gammal är du?
> 13
Du är myndig inom 5 år!
```

Övningsuppgift 2.5. Konstruera ett program i vilken en användare matar in fem heltal. Avgör med den matematiska standardfunktionen **max** det högsta inmatade talet och presentera detta för användaren.

```
Ange tal:

a = 1
b = 9
c = 2
d = 6
e = 5

Det högsta inmatade heltalet är 9.
```

Övningsuppgift 2.6. Inför skolutflykter behöver lärare planera inköp av korv och dryck. Inför varje utflykt samlar en lärare in information om hur många elever som:

- Vill äta 2 vanliga korvar.
- Vill äta 3 vanliga korvar.
- Vill äta 2 veganska korvar.
- Vill äta 3 veganska korvar.

Att manuellt sammanställa all data är tidskrävande och det blir lätt fel i uträkningarna. Av denna anledning ska du skapa ett program där läraren enkelt kan mata in datan och låta programmet räkna ut:

- Hur många förpackningar med vanliga korvar som behöver köpas in (8 korvar per förpackning).
- Hur många förpackningar med veganska korvar som måste köpas in (4 korvar per förpackning).
- Hur många drycker som behöver köpas in (1 per elev)

Programmet ska också räkna ut den totala kostnaden för utflykten. Utgå från prislistan som presenteras av Tabell 2.1.

Produkt	**Kostnad**
Vanlig korv	20.95:- per förpackning
Vegansk korv	34.95:- per förpackning
Dryck	13.95:- per flaska

Tabell 2.1: Prislista

```
.: KORVKOLLEN 1.0.1 :.
----------------------
Hur många elever vill ha...
2 vanliga korvar  > 7
3 vanliga korvar  > 5
2 veganska korvar > 3
3 veganska korvar > 5
----------------------
-    INKÖPSLISTA     -
----------------------
| Vanlig korv:  4 förpackningar
| Vegansk korv: 6 förpackningar
| Dryck:        20 drickor
----------------------
| 572.5 SEK
----------------------
```

Kapitel 3

Selektion

I förra kapitlet undersökte vi hur man lagrar data av olika typer i variabler. Vi har bland annat använt variabler i matematiska ekvationer, för att konstruera strängar och för att skriva ut data på skärmen.

I det här kapitlet tittar vi på hur man villkorar exekveringen av kodblock med hjälp av if-satser och jämförelseuttryck. Vi undersöker hur man konstruerar egna jämförelseuttryck med hjälp av jämförelseoperatorer.

Jämförelseoperatorerna skiljer sig mellan olika datatyper, både sett till vilka jämförelseoperatorer som finns tillgängliga och ibland också hur dessa fungerar. I slutet på kapitlet listar vi de vanligaste jämförelseoperatorerna för nummer och strängar.

3.1 Jämförelser med if

Med hjälp av if-satser kan man testa om en variabel är av ett visst värde:

```
age = int(input("Ange␣din␣ålder:␣"))

if 18 <= age:
    print("Du␣är␣myndig!")
else:
    print("Du␣är␣inte␣myndig!")
```

Jämförelser kan göras med såväl strängar, flyttal och heltal:

```
name = input('Ange␣ditt␣namn:␣')

if name == "Christian":
    print("Jättefint␣namn!")
else:
    print("Fint␣namn!")
```

Det är viktigt att man är medveten om datatyperna för värdena man jämför. Det är nästan aldrig en bra idé att utföra jämförelser av värden med olika datatyper. Vad tror du följande script skriver ut? Testa!

```
a = 1
b = "1"

if a == b:
    print(str(a) + "␣är␣lika␣med␣" + b)
else:
    print(str(a) + "␣är␣inte␣lika␣med␣" + b)
```

3.2 Villkorssatser

I Python finns totalt tre villkorssatser (**if**, **elif** och **else**):

```
print('Ange␣en␣frukt:')
fruit = input('>␣')

if fruit == 'banan':
    print('Bananer␣är␣gula␣och␣långa!')
elif fruit == 'apelsin':
    print('Apelsiner␣är␣orange␣och␣runda!')
else:
    print('Jag␣har␣inte␣hört␣om␣den␣frukten...')
```

I exemplet ovan ska användaren mata in en frukt. Beroende på värdet användaren matar in skrivs olika strängar ut. Vid nyckelorden **if** och **elif** testas variabelns värde med jämförelseoperatorer.

```
if fruit == 'banan':
    print('Bananer␣är␣gula␣och␣långa!')
```

Om variabeln **fruit** innehåller strängen *'banan'* kommer jämförelseoperatorn (*fruit == 'banan'*) returnera *True*; annars returnerar jämförelseoperatorn *False*. En if-sats körs endast om dess tillhörande jämförelseoperator ger *True*.

```
elif fruit == 'apelsin':
    print('Apelsiner␣är␣orange␣och␣runda!')
```

Nyckelordet **elif** (*"else if"*) fungerar som en if-sats men testas och exekveras endast om föregående if- och elif-satser inte kört.

```
else:
    print('Jag␣har␣inte␣hört␣om␣den␣frukten...')
```

En else-sats körs alltid om inga av de föregående if- och elif-satserna körts. För exempelprogrammet betyder detta att meddelandet *'Jag har inte hört om den frukten...'* alltid skrivs ut om variabeln **fruit** varken innehåller strängen *'banan'* eller *'apelsin'*.

3.2.1 Strängars jämförelseoperatorer

Tabell 3.1 visar de vanligaste jämförelseoperatorerna för strängar.

Operator	Beskrivning
a == b	*True* om *a* har samma värde som *b*.
a != b	*True* om *a* inte har samma värde som *b*.
a in b	*True* om *a* är ett snitt av *b*.

Tabell 3.1: Jämförelseoperatorer för strängar

3.2.2 Nummers jämförelseoperatorer

Tabell 3.2 visar de vanligaste jämförelseoperatorerna för heltal och flyttal.

Operator	Beskrivning
a == b	*True* om *a* är lika med *b*.
a != b	*True* om *a* inte är lika med *b*.
a > b	*True* om *a* är större än *b*.
a < b	*True* om *a* är mindre än *b*.
a >= b	*True* om *a* är större än eller lika med *b*.
a <= b	*True* om *a* är mindre än eller lika med *b*.

Tabell 3.2: Jämförelseoperatorer för heltal och flyttal

3.2.3 Nästlade villkorssatser

Det är möjligt att inkludera nya if-satser i befintliga block. Dessa typer av if-satser kallas för nästlade. Nedan följer ett exempel på detta:

```
specie = input('What␣are␣you?␣')

if specie != 'monster':
    print('Phew...')
    if specie == 'human':
        print('Hello␣friend!')
    else:
        print('Your␣specie␣is␣new␣to␣me!')
else:
    print('Aaargh!!')
```

3.3 Övningsuppgifter

Övningsuppgift 3.1. Skapa ett program i vilken användaren matar in tre tal. Programmet ska med if-satser identifiera det inmatade talet med störst numeriskt värde. Presentera därefter talet för användaren.

```
Ange ett tal: 6
Ange ännu ett tal: 7
Ange ett sista tal: 2
-----
Det största inmatade talet är 7.
```

Övningsuppgift 3.2. Enligt Vårdguiden beror individers sömnbehov bland annat på vilken ålder individen har. Tabell 3.3 visar i grova drag vilket sömnbehov individer har baserat på deras ålder.

Ålder	**Sömnbehov (per natt)**
1 år	14 timmar
2 år	13 timmar
3 år	12 timmar
4 år	11,5 timmar
5-6 år	11 timmar
7 år	10,5 timmar
8-10 år	10 timmar
11 år	9,5 timmar
12 - 15 år	9 timmar
16 år	8,5 timmar
17+ år	8 timmar

Tabell 3.3: En grov generalisering av individers sömnbehov.

Skapa ett program där användaren matar in sitt namn och ålder. Programmet ska (baserat från informationen i Tabell 3.3) ge användaren ett personlig meddelande där individens beräknade sömnbehov presenteras enligt exemplet nedan:

```
Ange ditt namn: Lisa Kallesson
Ange din ålder: 7
-----
Hallå Lisa Kallesson! Enligt Vårdguidens
rekommendationer behöver individer i din ålder (7 år)
sova minst 10,5 timmar per natt.
```

Övningsuppgift 3.3. Slumpmässighet utgör ett viktigt element i många brädoch datorspel eftersom det skapar en grad av oförutsägbarhet. I datorspel brukar slumpmässighet programmeras in i spelets olika komponenter. Exempelvis så kan datorstyrda karaktärer agera och göra val enligt delvis slumpmässiga mönster för att mer likna mänskliga spelare. Ett annat exempel är i racingspel där det är vanligt att innan race med slumptal generera racinbanans temperatur och vind-egenskaper, vilket kräver att spelare har förmågan att ställa om sina körstilar.

I traditionella brädspel kan inte slumpen döljas för spelaren med datorkod. Istället är spelaren med och genererar slumpen, vanligtvis med tärningar eller en blandad kortlek.

I den här uppgiften introduceras du till hur man genererar slumptal med Python. Du ska skapa ett script som simulerar ett tärningskast och visualiserar resultatet med mycket enkel ASCII art. Utgå från följande kod:

```
import random

dice = random.randint(1, 6)
```

Funktionen **random.randint(1, 6)** genererar ett slumpmässigt heltal från 1 till 6 (i likhet med en sexsidig tärning). Det slumpade heltalet lagras sedan i variabeln **dice**.

Förläng scriptet så att en informativ text skrivs ut om vilket värde som genererats. Visualisera också tärningenskastets resultat med ASCII art i likhet med exemplen nedan:

Exempel: Vid första körning antog variabeln **dice** värdet 2.

```
Du slog en 2:a!
---------
| X     |
|       |
|     X |
---------
```

Vid andra körningen antog variabeln värdet 5:

```
Du slog en 5:a!
---------
| X   X |
|   X   |
| X   X |
---------
```

Övningsuppgift 3.4. Skapa ett program som ber användaren mata in ett land. Programmet ska sedan svara om landet är en del av Norden eller Storbritanien. Jämförelsen ska inte vara skriftlägeskänslig. Det ska med andra ord inte spela någon roll om inmatningen görs med stora eller små bokstäver. Om landet som användaren matar in inte tillhör varken Norden eller Storbritanien ska ett felmeddelande visas. Till Norden hör följande länder:

- Danmark
- Finland
- Island
- Norge
- Sverige

Till Storbritanien hör länderna England, Nordirland, Skottland och Wales.

Övningsuppgift 3.5. Skapa ett script där användaren matar in en mängd personliga egenskaper:

- Kön
- Hårfärg
- Ögonfärg

Efter inmatning ska scriptet ange vilka kända personer som matchar med egenskaperna. Följande är ett exempel på hur ditt script kan fungera:

```
Ange kön: kvinna
Ange hårfärg: brun
Ange ögonfärg: brun
-----
Egenskaperna matchar med: Emma Watson, Selena Gomez
```

Om ingen känd person matchar med egenskaperna ska ett felmeddelande skrivas. Förutom personerna (med tillhörande egenskaper) som listas i Tabell 3.4 måste ni välja ytterligare fem personer.

Namn	**Kön**	**Hårfärg**	**Ögonfärg**
Daniel Radcliffe	man	brun	brun
Rupert Grint	man	röd	blå
Emma Watson	kvinna	brun	brun
Selena Gomez	kvinna	brun	brun

Tabell 3.4: Kända personer som ska vara sökbara i ert program.

Kapitel 4

Loopar

Loopar utgör en viktig del av programmering. Med loopar kan man systematiskt upprepa kodstycken, vilket används vid allt från matematiska beräkningar, grafiska gränssnitt, sökning av stora datamängder, med mera.

4.1 Loopar med while

Man kan se while-loopen som en utökad version av if-satsen. Till skillnad från if-satsen som endast körs en gång så körs while-loopen om tills jämförelseuttrycket är falskt. Vad skrivs ut av följande uttryck?

```
number = 1
if number < 4:
    print(number)
    number += 1
```

Eftersom **if**-satser körs en och endast en gång kommer man i satsen först skriva ut värdet av **number**. Efter detta adderas värdet 1 till **number** och programmet stängs:

```
1
```

Vad händer om vi byter ut **if** mot **while**?

```
number = 1
while number < 4:
    print(number)
    number += 1
```

Precis som med if-satsen så skrivs först värdet av **number** ut. Därefter adderar man 1 till **number**. Eftersom vi nu kör i ett **while**-block så kommer en ny jämförelse göras. Är jämförelsen sann (*True*) kommer **while**-blocket köras på nytt. Detta görs tills det att jämförelsen ger ett falskt resultat (*False*):

```
1
2
3
```

4.1.1 break

Det är möjligt att bryta exekveringen av en while-loop på annat sätt än att vänta tills jämförelsen ger **False**. Detta görs med nyckelordet **break**. Vad skrivs ut av följande uttryck?

```
number = 1
while number < 4:
    print(number)
    if number == 2:
        break
    number += 1
```

Till skillnad från exemplet i föregående sektion (som skrev ut samtliga heltal mellan 1 och 3) kommer **while**-loopen att brytas (med **break**) när **number** är lika med 2. Således skrivs följande ut:

```
1
2
```

4.1.2 continue

Det är möjligt att tvinga en omstart av en while-loop. Detta görs med nyckelordet **continue**. Nyckelordet **continue** förhindar fortsatt exekvering av det aktuella while-blocket. En ny jämförelse (i while-satsen) kommer direkt utföras. Ger jämförelsen *True* körs while-blocket om som vanligt. Ger jämförelsen *False* bryts loopen. Försök lista ut vad som skrivs ut av uttrycket:

```
number = 1
while number < 4:
    print(number)
    if number == 2:
        continue
    number += 1
```

När **number** är lika med två kommer nyckelordet **continue** få while-loopen att startas om innan värdet 1 hunnit adderas till **number**. Detta leder till att while-loopen kommer startas om i all oändlighet, utan att någonsin brytas:

```
1
2
2
2 # och så vidare, i all oändlighet.
```

Det gäller med andra ord att hålla tungan rätt i mun och ha ett tydligt mål med användningen av **continue**.

4.2 Övningsuppgifter

Övningsuppgift 4.1. Konstruera ett program i vilket användaren matar in en talserie. Användaren matar in ett tal åt gången tills det att användaren matar in ett negativt tal. Då ska programmet presentera det största och lägsta inmatade talet samt summan och medelvärdet av de inmatade talen. Bland de inmatade talen ska inte det avslutande negativa talet räknas med.

```
NUMANALYZER
  v1.33.7

Tal < 8
Tal < 10
Tal < 9
Tal < 12
Tal < 14
Tal < 4
Tal < -1

Minsta tal:  4
Största tal: 14
Summa:       57
Medelvärde:  9.5
```

Övningsuppgift 4.2. Skapa ett program där användaren anger en multiplikationstabell denna vill skriva ut på skärmen. Programmet ska skriva ut tre tal från den angivna multiplikationstabellen åt gången. Därefter ska användaren frågas om denna vill ha ytterligare tre tal utskrivna. Svarar användaren **ja** ska programmet skriva ut de nästföljande tre talen från multiplikationstabellen och fråga på nytt. Svarar användaren **nej** ska programmet stängas. Ett exempel på detta visas nedan:

```
Ange multiplikationstabell> 7
7
14
21
Fortsätt? ja
28
35
42
Fortsätt? ja
49
56
63
Fortsätt? nej
```

Övningsuppgift 4.3. Higher Lower är ett spel där datorn randomiserar ett tal mellan 0 och 99. Spelarens uppgift är att gissa sig till vilket tal datorn genererat. Till varje gissning svarar datorn om numret är högre eller lägre än spelarens gissning.

Din uppgift är att implementera detta spel i Python. När spelaren gissat rätt ska datorn ge återkoppling på hur många gissningar spelaren gav. En exempellösning visas nedan:

```
.: THE HIGHER LOWER GAME :.
---------------------------
Welcome to The Higher Lower
Game. I  will  randomise  a
number between  0  and  99.
Can you guess it?
---------------------------
Your guess > 50
HIGHER!
Try again > 75
HIGHER!
Try again > 80
LOWER
Try again > 70
HIGHER!
Try again > 75
HIGHER!
Try again > 76
HIGHER!
Try again > 77
HIGHER!
Try again > 78
---------------------------
78 is correct!
It took you 8 guesses.
Good job!
```

Kapitel 5

Felhantering

När man skapar program med Python så är det viktigt att man har logik som tar hand om eventuella fel som kan uppstå under körning. Särskilt viktigt är detta när man skriver kod som interagerar med omvärlden (exempelvis vid användarinmatning och API-anrop).

5.1 Exceptions

I följande program matar användare in ett heltal. Programmet beräknar och skriver ut talets kvadrat:

```
tal = input('Ange␣ett␣heltal:␣')
tal = int(tal)
kvadrat = tal * tal
print(tal, 'i␣kvadrat␣är', kvadrat)
```

Detta fungerar utmärkt så länge användaren inte råkar mata in ett värde som inte är ett heltal:

```
Ange ett heltal: femtio2
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    tal = int(tal)
ValueError: invalid literal for int() with base 10: 'femtio2'
```

Ur felmeddelandet kan vi utläsa att ett fel uppstod i scriptet *main.py*" rad 2:

```
tal = int(tal)
```

Felet är av typen **ValueError**. Python försöker också ge oss en pedagogisk beskrivning av felet:

> *invalid literal for int() with base 10: 'femtio2'*

Strängen *'femtio2'* är inte formaterad på ett sätt som funktionen **int()** kan omvandla till ett heltal. Istället för att returnera någonting felaktigt väljer funktionen **int()** istället att lyfta ett felmeddelande (på engelska *"raise exception"*).

5.1.1 Felsäker kod

I princip alla fel som orsakar en abrupt krasch av programmet beror på ett felmeddelanden som lyfts och inte fångats av en annan del av koden (likt exemplet ovan). Att skriva ett program som inte kraschar handlar på många sätt om att du som programmerare är medveten om vad som kan gå fel i koden och därmed skriver din kod på ett sådant sätt så att dessa potentiella fel kan tas om hand. Detta gör du med nyckelorden **try** och **except**.

```
tal = input('Ange␣ett␣heltal:␣')

try:
    tal = int(tal)
    print(tal, 'är␣ett␣heltal')
except ValueError:
    print(tal, 'är␣inte␣ett␣heltal')

print('Stänger␣programmet...')
```

Om någon av operationerna i try-blocket lyfter ett fel av typen **ValueError** så kommer körningen hoppa direkt till except-blocket. Om felet inte uppstår så kommer inte heller except-blocket exekvera.

Om man för exemplet ovan anger ett giltigt heltal så körs hela try-blocket. Eftersom samtliga operationer i try-blocket exekverade utan några fel så körs inte except-blocket.

```
Ange ett heltal: 1337
1337 är ett heltal
Stänger programmet...
```

Anger man ett värde som **int()** inte kan omvandla till ett heltal lyfter funktionen ett felmeddelande av typen **ValueError**. Felmeddelandet fångas av except-satsen vilket förhindrar en krasch:

```
Ange ett heltal: 133t
133t är inte ett heltal
Stänger programmet...
```

Försök skriva om programmet i början på detta stycke så att programmet inte kraschar vid felaktig inmatning!

5.2 Övningsuppgifter

Övningsuppgift 5.1. Konstruera ett program i vilken användaren matar in ett heltal. Programmet ska skriva ut det dubbla värdet av det inmatade heltalet.

```
tal > 5
RESULTAT: 10
```

Om användaren matar in ett värde som inte kan översättas till ett heltal ska ett felmeddelande skrivas ut likt exemplet nedan.

```
tal > hej
FEL: 'hej' kan inte översättas till ett heltal
```

Övningsuppgift 5.2. Konstruera ett program där användaren ska mata in värdet för två flyttal **a** och **b**. Efter användaren matat in variabelvärdena ska programmet skriva ut kvoten mellan **a** och **b**.

```
**********************
* The Great Divider *
----------------------

  Beräknar c för uttrycket:

    a / b = c

----------------------
a = 10
b = 2
----------------------
10.0 / 2.0 = 5.0
```

Ditt program ska (med **try** och **except**) implementera felhantering för både användarinmatning och division med 0.

```
----------------------
a = tre
FEL: Ogiltigt nummer
a = 3
b = noll
FEL: Ogiltigt nummer
b = 0
----------------------
FEL: Division med 0
```

Övningsuppgift 5.3. Konstruera ett program i vilken användaren matar in ett i förväg okänt antal nummer. Programmet ska fortsätta låta användaren mata in nummer tills det att användaren matar in *'exit'*. Då ska programmet skriva ut information om de inmatade heltalen:

- Antal inmatade tal (kardinalitet)
- Summan av de inmatade heltalen
- Medelvärdet av de inmatade heltalen

Om användaren varken skriver ut ett giltigt nummer eller *'exit'* ska ett felmeddelande skrivas ut.

```
.: MATHLETE v2.0 :.
--------------------
> 4
> 9
> tre
FEL: Ogiltigt nummer
> 5
> 6
> exit
--------------------
Kardinalitet: 4
Summa:        24.0
Medelvärde:   6.0
```

Kapitel 6

Strängar

Grundläggande hantering av strängar har vi redan ägnat oss åt vid tidigare kapitel. Vi vet bland annat hur man lagrar en sträng i en variabel och skriver ut variabelns värde på skärmen.

```
meddelande = 'Jag␣gillar␣strängar'
print(meddelande)
```

Vi har också tittat på några grundläggande operatorer för strängar. Den kanske vanligaste operatorn för strängar är addition.

```
förnamn = 'Lisa'
efternamn = 'Gunillasson'
fullnamn = förnamn + '␣' + efternamn
print(fullnamn)
```

En annan behändig operator för strängar är multiplikation.

```
h = 'HEJ!␣'
meddelande = 3 * h # Ekvivalent med h + h + h
print(meddelande)
# HEJ! HEJ! HEJ!
```

I det här kapitlet går vi några steg längre. Vi undersöker bland annat vilka inbyggda metoder och funktioner Python erbjuder för strängar. Vi tittar också på hur man indexerar och itererar individuella bokstäver i strängar. Avslutningsvis undersöker vi grunderna till hur man skapar användargränsnitt.

6.1 Datatypen string

Med datatypen sträng (på engelska *string*) hanterar man texter i Python. Det är viktigt att du blir bekväm att hantera strängar om du tänkt skapa program som interagerar med användare, textfiler eller API:er.

6.1.1 Vart kommer strängar ifrån?

Strängar i dina script kan komma från flera olika källor. Du kan i källkoden på egen hand definiera strängar. Läser du in data från användare eller filer så lagras ofta dessa också som datatypen sträng.

Egendefinierade Du kan statiskt deklarera en sträng genom att omsluta en text med enkla eller dubbla citattecken.

```
text = 'God dag'
text = "God dag"
```

Sträng från användare När du läser in data från användare lagras detta som en sträng.

```
namn = input()
print(namn) # namn har datatypen 'string'
```

Sträng från textfil När du läser in data från textfiler lagras även denna (som standard) som en sträng i datorns minne.

```
with open('textfil.txt') as f:
    text = f.read()
print(text) # text har datatypen 'string'
```

6.1.2 Escape-sekvenser

För strängar är backslash (\) en speciell karaktär. I strängar representerar backslash början på en escape-sekvens. Tecknet brukar därför refereras till som escape-karaktären.

Escape-sekvenser används för att representera olika karaktärer i strängar. En del av dessa karaktärer kan inte skrivas i källkod och behöver därför skrivas som en escape-sekvens. Nedan presenteras två olika tillämpningsområden för escape-sekvenser.

Citationstecken

Utgå från följande text:

> *Pojken sa ”akta” till sin kamrat*

Vi får problem när vi försöker lagra texten som en sträng vi omsluter med dubbla citattecken:

```
meddelande = "Pojken sa "akta" till sin kamrat"
print(meddelande)
```

Syntaxmarkeringen ser inte helt korrekt ut. Testar man köra koden så kraschar Python med felet *SyntaxError*. Python klarar inte av att tolka texten vi försöker lagra i **meddelande**. Vad är det som gör att kodens syntax är felaktig?

När man inleder en sträng med dubbla citattecken kommer Python anta att strängen avslutas vid närmast efterföljande citattecken. För koden ovan utgör *”Pojken sa ”* enligt Python en fullständig sträng. Det som följer efter detta tillhör inte längre strängen Python läste in. Efter första strängen följer ordet akta. Ordet akta är inte en del av första strängen.

I bästa fall kunde Python anta att **akta** var en variabel som användaren ville addera till första och andra strängen. Men vill man göra detta behöver man enligt Pythons språkregler använda plustecken mellan strängen och variabeln vi vill slå ihop:

```
akta = "hejsan"
meddelande = "Pojken␣sa␣" + akta + "␣till␣sin␣kamrat"
print(meddelande)
```

Men detta är inte alls vad vi försöker göra!

Vi vill att Python ska tolka meningen *”Pojken sa ”akta” till sin kamrat”* som en enda sträng. Detta kan vi göra genom att representera samtliga dubbla citattecken i strängen med motsvarande escape-sekvens. Detta gör vi genom att skriva escape-karaktären framför citattecknen:

```
meddelande = "Pojken␣sa␣\"akta\"␣till␣sin␣kamrat"
print(meddelande)
```

För just det här exemplet ska nämnas att det finns en lösning som inte kräver escape-sekvenser. Vill vi undvika skriva dubbla citattecken som escape-sekvenser kan vi istället omsluta strängen med enkla citattecken:

```
meddelande = 'Pojken␣sa␣"akta"␣till␣sin␣kamrat'
print(meddelande)
```

Radbrytning

Hittills har alla strängar vi arbetat med bestått av endast en rad:

```
text = 'Den␣här␣strängen␣består␣endast␣av␣en␣rad'
```

Men en sträng kan bestå av flera rader. Detta märken man (om inte tidigare) när man arbetar med textfiler. Datorer lagrar radbrytningar på samma sätt som den lagrar vanliga bokstäver (som en samling karaktärer). I datorspråk brukar karaktären för radbrytning kallas för *newline*-karaktären.

I strängar representeras *newline*-karaktären av escape-sekvensen $\backslash n$:

```
text = 'RAD␣1\nRAD␣2\nRAD␣3\nRAD␣4'
print(text)
```

Ska du i din källkod definiera en längre sträng med flera rader är det inte särskilt kul att behöva skriva escape-sekvenser vid varje radbrytning. Strängar som omsluts av tre enkel eller dubbla citattecken kan skrivas över flera rader.

```
text = '''RAD 1
RAD 2
RAD 3
RAD 4'''
```

6.1.3 Metoder

Datatypen sträng klarar av mer än att bara lagra texter. Datatypen introducerar även en samling metoder som är kapabla till att hantera och modifiera datan som är lagrad i strängen.

Metoder har sitt ursprung i den objektorienterade programmeringen. Metoder är funktioner som är inbyggda i datatyper (som exempelvis strängar, flyttal eller heltal). För att förstå hur man använder metoder tror jag det lämpar sig med praktiska exempel. Nedan lagrar jag en sträng i variabeln **text**:

```
text = 'jag tycker om racing'
```

Metoder anropas som funktioner. Men eftersom metoderna är inbyggda i datatypen behöver vi referera till datan vars metod vi vill anropa:

```
u = text.upper()
print(u) # JAG TYCKER OM RACING
```

Strängar har en inbyggd metod ***string*.upper()** som gör om samtliga av strängens karaktärer till versaler. Metoden ändrar inte själva strängen som anropas, utan returnerar endast resultatet från metod-anropet. I exemplet ovan lagrar jag metod-anropets resultat i en variabel jag döper till **u**.

Metoder kan användas direkt på data som definierats i källkoden. Vad tror du följande skriver ut?

```
print('jag tycker om racing'.upper())
```

Nedan tittar vi på några vanliga metoder för strängar.

lower

Likt hur ***string*.upper()**, som presenteras i föregående stycke, omvandlar samtliga karaktärer till versaler omvandlar istället ***string*.lower()** samtliga karaktärer till gemener:

```
meddelande = 'FLYTTA PÅ DIG'
meddelande = meddelande.lower() # 'flytta på dig'
```

replace

Metoden ***string*.replace(old, new)** används för att ersätta samtliga förekomster av argumentet **old** med argumentet **new**:

```
info = 'Johan gillar picca'
info = info.replace('c', 'z') # 'Johan gillar pizza'
```

Med metoden kan du ersätta allt från enskilda bokstäver till hela ord:

```
ordspråk = 'Rosor␣är␣röda,␣violer␣är␣explosiva.'
ordspråk = ordspråk.replace('explosiva', 'blå')
# 'Rosor är röda, violer är blå.'
```

Sätter du argumentet **new** som en tom sträng resulterar detta istället i att du raderar samtliga förekomster av argumentet **old**:

```
error = 'ERROR:␣Not␣found'
error_type = error.replace('ERROR:␣', '') # 'Not found'
```

strip

Ibland händer det att en sträng har inledande och avslutande mellanslag:

' *den här strängen börjar och slutar med mellanslag* '

Detta kanske på grund av att en användare råkat lägga till dessa vid inmatning. Med metoden ***string*.strip()** är det möjligt att rensa bort mellanslag som omsluter en text:

```
text = '␣␣␣god␣dag␣␣␣␣␣'
text = text.strip() # 'god dag'
```

center

Metoden ***string*.center(width)** fyller ut en sträng med mellanslagskaraktärer så att strängen får lika många karaktärer som specificeras av argumentet **center** och så att texten i *string* hålls centrerad:

```
info = 'Hello␣World'
info = info.center(20)
#     Hello World      '
```

Denna metod är användbar när du skapar grafiska gränssnitt till dina program:

```
ui_width = 20

print(ui_width * '*')
print('MAIN␣MENU'.center(ui_width))
print(ui_width * '-')

# ********************
#       MAIN MENU
# --------------------
```

Notera för hur du exemplet ovan endast behöver modifiera variabeln **ui_width** för att göra ditt gränssnitt bredare. Samtliga element anpassas automatiskt. Lätthanterligt och elegant!

ljust

Metoden ***string*.ljust(width)** lägger till mellanslag till texten i *string* tills strängen har lika många karaktärer som argumentet **width**.

```
text = 'en␣blå␣bil'
text = text.ljust(20) # 'en blå bil          '
```

Metoden fungerar alltså på samma sätt som ***string*.center(width)** fast med skillnaden att texten vänster-justeras. Detta är användbar när man vill skriva ut data i tabell-format:

```
p_name = 'Gunilla␣Johnson'
p_age = 33
p_gender = 'female'

c_width = 8

print('NAME:'.ljust(c_width) + p_name)
print('AGE:'.ljust(c_width) + str(p_age))
print('GENDER:'.ljust(c_width) + p_gender)
```

Notera hur du i exemplet ovan endast behöver modifiera variabeln **c_width** för att ändra första kolumnens bredd.

6.1.4 Indexering

Med indexering kan man referera till enskilda karaktärer i strängar. Du skriver ***string*[i]** för att referera till karaktären i *string* med index **i**:

```
bokstäver = 'ABCDEF'
print(bokstäver[0]) # A
print(bokstäver[1]) # B
print(bokstäver[2]) # C
print(bokstäver[3]) # D
print(bokstäver[4]) # E
print(bokstäver[5]) # F
print(bokstäver[6])
# Traceback (most recent call last):
#   File "main.py", line 8, in <module>
#     print(bokstäver[6])
# IndexError: string index out of range
```

Strängens första karaktär har index 0, strängens andra karaktär har index 1 och så vidare. Refererar du till ett index som är längre än strängen kraschar Python med ett *IndexError*, vilket jag visar i exemplet ovan.

Det är möjligt att indexera bakifrån med negativa tal. Var dock noga med att inte indexera längre än strängens storlek.

```
bokstäver = 'ABCDEF'
print(bokstäver[-1]) # F
print(bokstäver[-2]) # E
print(bokstäver[-3]) # D
print(bokstäver[-4]) # C
print(bokstäver[-5]) # B
print(bokstäver[-6]) # A
print(bokstäver[-7])
# Traceback (most recent call last):
#   File "main.py", line 8, in <module>
#     print(bokstäver[-7])
# IndexError: string index out of range
```

6.1.5 Antalet karaktärer

Funktionen **len(*string*)** kan användas för att ta reda på antalet karaktärer som finns i en lista.

```
bokstäver = 'ABCDEF'
l = len(bokstäver)
print(l) # 6
```

6.1.6 Iteration av karaktärer

Kombinerar vi funktionen **len(*string*)** med en **while**-loop och tillämpar indexering kan vi skriva ut var enskild bokstav på en ny rad. Testa gärna köra följande program. Se till att du förstår hur det fungerar:

```
bokstäver = 'ABCDEF'
char_len = len(bokstäver)
i = 0
while i < char_len:
    bokstav = bokstäver[i]
    print(bokstav)
    i += 1
```

6.2 Övningsuppgifter

Övningsuppgift 6.1. Du ska konstruera ett program som räknar förekomsten av bokstäver i texter. Genom användarinmatning ska programmet hämta både strängen som ska undersökas och vilken bokstav som ska räknas.

Uppgiften måste lösas genom att strängens bokstäver itereras med en while-loop. Jämförelsen av bokstäver ska inte vara skiftlägeskänslig.

Ett lösningsförslag visas nedan:

```
Ange en text: God dag alla glada studenter från Gbg!
Ange bokstav: g

Bokstaven g förekommer 5 gånger i texten.
```

Övningsuppgift 6.2. Konstruera ett program som översätter inmatade texter enligt rövarspråket och presenterar resultatet för användaren.

Rövarspråket är ett enkelt kodspråk där man efter varje konsonant lägger till ett o och därefter samma konsonant igen.

Ordet *'hej'* översätts enligt rövarspråkets regler till *'hohejoj'* eftersom bokstäverna H och J är konsonanter.

Följande bokstäver är konsonanter och behöver därför översättas av ditt script:

b, c, d, f, g, h, j, k, l, m, n, p, q, r, s, t, v, w, x och z

Ett lösningsförslag visas nedan:

```
Robber Translate
----------------
Svenska    < jag är en rövare
Rövarspråk > jojagog äror enon rorövovarore
```

Övningsuppgift 6.3. Konstruera ett program som kan avgöra om en inmatat mening är ett palindrom eller inte.

Ett palindrom är ett ord eller en mening där bokstävernas följd (blanksteg och skiljetecken exkluderade) är oförändrad när du läser upp meningen baklänges. Jämförelser av bokstäver i palindrom ska inte vara skiftlägeskänslig.

Strängen *'Ni talar bra latin'* blir baklänges *'nital arb ralat iN'*. Exkluderar vi mellanslag är teckenföljden identisk (*'nitalarbralatin'*). Strängen är därför ett palindrom. Nedan följer några lösningsförslag:

```
Ange sträng: Sirap i Paris
'Sirap i Paris' är ett palindrom
```

```
Ange sträng: Vatten är gott
'Vatten är gott' är inte ett palindrom
```

Kapitel 7

Användargränssnitt

Bygger man program som ska köras av andra är det viktigt att tänka på hur användaren interagerar med programmet. I det här kapitlet undersöker vi grunderna till hur man bygger interaktiva användargränssnitt.

7.1 Interaktiva användargränssnitt

Analysera scriptet nedan. Försök att på egen hand förstå hur det fungerar:

```
print('.: FÄRG-GISSAREN 2.0 :.')
print('-' * 23)
print(':: REGLER ::'.center(23))
print('Gissa en färg!'.center(23))
print('Gissar du rätt färg'.center(23))
print('vinner du spelet!'.center(23))
print('-' * 23)

times = 1
color = input('Gissa färg > ')
while color != 'gul':
    print('Fel gissning, försök igen...')
    color = input('Gissa färg > ')
    times += 1

print('-' * 23)
print('Korrekt gissat efter', times, 'försök!')
```

Exemplet ovan är ett spel där användaren ska gissa sig fram till färgen gul. När användaren gissar rätt färg får denna veta hur många gissningar som gjordes för att få fram rätt svar.

Låt oss titta närmre på en minimerad version av loopen:

```
color = input('Gissa␣färg␣>␣')
while color != 'gul':
    color = input('Gissa␣igen␣>␣')
print('Korrekt')
```

Loopar av den här typen kan med fördel användas för att konstruera enkla former av interaktiva användargränssnitt.

Nedan följer ett program där användare (med kommandon) kan instruera datorn att skriva ut värdet av olika matematiska konstanter. Med utgångspunkt från **while**-loopen ovan gjorde jag några förändringar. Istället för att gissa en färg låter vi användaren göra ett val från en meny. Istället för att brytas när användaren matar in strängen *'gul'* bryts loopen när användaren matar in strängen *'exit'*:

```
print('.:␣The␣Constant␣Printer␣:.')
print('--------------------------')
print('e␣␣␣␣|␣Eulers␣tal')
print('pk␣␣␣|␣Pythagoras␣konstant')
print('pi␣␣␣|␣Pi')
print('exit␣|␣Stäng␣programmet')
print('--------------------------')

command = input('>␣')
while command != 'exit':
    if command == 'e':
        print(2.71828)
    elif command == 'pk':
        print(1.41421)
    elif command == 'pi':
        print(3.14159)
    else:
        print('FEL:␣Okänt␣kommando␣(' + command + ')')
    command = input('>␣')

print('Stänger␣programmet!')
```

Beroende på vilket kommando (**e**, **pk** eller **pi**) användaren matar in skrivs olika värden ut. Med satser av typen **if** och **elif** jämförs därför användarens inmatning mot de olika kommandona så att korrekt värde skrivs ut. Matar användaren in ett kommando som inte existerar går programmet in i **else**-satsen där ett felmeddelande skrivs ut. Loopen avslutas med att användaren matar in ett nytt kommando, varpå loopen körs om på nytt tills det att användaren matar in kommandot **exit**.

7.2 Rensa terminalfönster

Det finns ett potentiellt problem med programmet vi skrev i föregående stycke (The Constant Printer). För varje kommando användaren skriver in kommer instruktionerna flyttas högre och högre upp i terminalfönstret. Detta eftersom varje kommando resulterar i att två nya rader skrivs ut (längst ner) i terminalen. När användaren skrivit in tillräckligt många kommandon syns inte längre instruktionerna.

För att undvika att hamna i situationer där programmets instruktioner försvinner, på grund av för många utskrifter, kan det vara smart att mellan kommandon rensa terminalfönstret och skriva ut instruktionerna på nytt.

Windows, Mac och Linux tillhandahåller inbyggda kommandon för att rensa terminalfönstret. Kommandona skiljer sig mellan operativsystem, men i slutet av det här stycket presenterar vi en metod som (för ovan nämnda operativsystem) är plattformsoberoende.

7.2.1 Windows

Med kommandot **cls** kan vi rensa terminalfönstret i Windows. För att köra operativsystemsspecifika kommandon i Python använder vi funktionen **system** från bilbioteket **os**.

Följande kod visar hur man rensar terminalfönstret i Windows:

```
import os
print('Hallå')
os.system('cls')
print('Hej␣då')
```

Eftersom terminalen rensas efter första print-satsen (som skriver ut *'Hallå'*) kommer den utskriften att raderas. Resultatet från scriptet blir följande:

```
Hej då
```

7.2.2 Mac och Linux

I Mac och Linux används kommandot **clear** för att rensa terminalfönstret:

```
import os
print('Hallå')
os.system('clear')
print('Hej␣då')
```

Precis som med Windows kommer endast den sista print-satsen synas, eftersom den första rensas bort:

```
Hej då
```

7.2.3 Plattformsoberoende implementation

Ska ett script kunna köra på samtliga plattformar är det viktigt att man undviker plattformspecifika lösningar. Arbetar man med stora Python-projekt vill man så långt det är möjligt undvika separata script för Windows, Mac och Linux.

Med attributen **name** från bilbioteket **os** kan vi hämta information om vilket operativsystem scriptet körs på. Körs scriptet på Windows är attributen satt till *'nt'*. Kör man på Mac eller Linux är attributen satt till *'posix'*:

```
import os

print('Hallå')

if os.name == 'nt':
    print('Du␣kör␣Windows')
elif os.name == 'posix':
    print('Du␣kör␣Mac␣eller␣Linux')

print('Hej␣då')
```

Ser du vart jag är på väg med detta? Nu när vi har implementerat en kontroll om vilket operativsystem användaren kör så kan vi ju se till att skicka motsvarande kommando för det operativsystem scriptet körs på för att rensa terminalfönstret.

```
import os

print('Hallå')

if os.name == 'nt':
    os.system('cls')
elif os.name == 'posix':
    os.system('clear')

print('Hej␣då')
```

Trots att vi skickar plattformsspecifika kommandon till operativsystemet har vi med kontrollen kunnat bevara vårt script plattformsoberoende. Snyggt!

7.3 Ett förbättrat användargränssnitt

Som nämndes i föregående stycke finns potentiella problem med användargränssnitt som aldrig rensar skärmen. I det här stycket ska vi bygga om The Constant Printer (från sektion 7.1) så att skärmen rensas och instruktionerna skrivs ut på nytt mellan kommandon.

I förra implementationen undersökte uttrycket till while-satsen det inmatade kommandot, likt följande minimaliserade exempel:

```
command = input('> ')
while command != 'exit':
    if command == 'pi':
        print(3.14)
    else:
        print('FEL: Okänt kommando')
    command = input('> ')
```

För att behålla en vettig struktur som passar för att rensa terminalfönstret så kommer jag byta ut while-satsens jämförelseoperator mot det booleska värdet **True**. En while-sats med uttrycket **True** kommer alltid startas om tills det att loopen manuellt bryts av nyckelordet **break**. Jag lägger därför till en elif-sats som kontrollerar när användaren matar in kommandot **exit**:

```
while True: # Bryts när användaren matar in 'exit'
    command = input('> ')
    if command == 'pi':
        print(3.14)
    elif command == 'exit':
        break
    else:
        print('FEL: Okänt kommando')
```

Som nämns av kurslitteraturen ska man helst undvika evighetsloopar som bryts med **break** eftersom detta gör koden mer svårläst. Men för just det här fallet tycker jag att koden får en bättre och tydligare struktur. För tydlighets skull inkluderade jag dock en kommentar till while-satsen som förtydligar när loopen bryts.

Hela programmet implementeras i while-loopen med följande struktur:

1. Rensa terminalfönster
2. Skriv ut instruktioner
3. Hämta kommando från användaren
4. Utför operationer för inmatat kommando
    - Om *'exit'*; bryt programmets evighetsloop
    - Om kommandot är okänt; skriv ut felmeddelande
5. Pausa exekvering
6. Gå till 1

Resultatet hittar du på nästa sida.

Tips! Tillämpa strukturen ovan på Case 1.

```
import os

while True:
    # 1. Rensa terminalfönster
    if os.name == 'nt':
        os.system('cls')
    elif os.name == 'posix':
        os.system('clear')

    # 2. Skriv ut instruktioner
    print('.: The Constant Printer :.')
    print('--------------------------')
    print('e    | Eulers tal')
    print('pk   | Pythagoras konstant')
    print('pi   | Pi')
    print('exit | Stäng programmet')
    print('--------------------------')

    # 3. Hämta kommando från användaren
    command = input('> ')

    # 4. Utför operationer för inmatat kommando
    if command == 'e':
        print(2.71828)
    elif command == 'pk':
        print(1.41421)
    elif command == 'pi':
        print(3.14159)
    elif command == 'exit': # bryt programmets evighetsloop
        break
    else: # skriv ut felmeddelande
        print('FEL: Okänt kommando (' + command + ')')

    print('--------------------------')
    # 5. Pausa exekvering
    input('Tryck enter för att fortsätta...')
    # 6. Gå till 1
```

7.4 Övningsuppgifter

Övningsuppgift 7.1. I den här uppgiften ska du konstruera en digital anslagstavla. Anslagstavlan ska ha stöd för tre separata poster som användaren efter behag kan förändra:

```
.: basicBILLBOARD :.
********************
P1: Detta är första posten.
P2: Detta är andra posten.
P3: Detta är tredje posten.
--------------------
c | Ändra post
e | Stäng program
--------------------
meny >
```

Anslagstavlans tre poster finns lagrade i variablerna med prefix POST. Tillsammans med programmets instruktioner skrivs dessa ut för användaren:

```
POST_1 = ''
POST_2 = ''
POST_3 = ''

while True:
    # [ ] 1. Rensa terminalfönster
    # [X] 2. Skriv ut instruktioner
    print('.: basicBILLBOARD :.')
    print('********************')
    print('P1:', POST_1)
    print('P2:', POST_2)
    print('P3:', POST_3)
    print('--------------------')
    print('c | Ändra post')
    print('e | Stäng program')
    print('--------------------')
    # [ ] 3. Hämta kommando från användaren
    # [ ] 4. Utför operationer för inmatat kommando
    # [X] 5. Pausa exekvering
    input('Tryck enter för att fortsätta...')
    # [X] 6. Gå till 1
```

Slutför delarna som saknas i scriptet ovan. Du väljer själv hur användaren modifierar posterna. Ta gärna inspiration från vårt interaktiva lösningsförslag[1].

[1] https://dva128demo.mdh.repl.co/#week3/basicBILLBOARD

Kapitel 8

Listor

Såhär långt har vi, bland annat, läst om datatyperna heltal, flyttal och strängar. Heltal och flyttal är datatyper som anpassats för att hantera nummer. Datatypen sträng är anpassad för att hantera texter.

I det här kapitlet introducerar vi en ny datatyp. Datatypen är varken anpassad för att hantera nummer eller texter (det räcker med dom vi redan har). Nej, den här datatypen är enbart konstruerad för att hantera datamängder. Datatypen vi talar om heter på engelska list. På svenska brukar datatypens namn direktöversättas till lista.

8.1 Datatypen list

Datatypen *list* i Python används för att bunta ihop datamängder:

```
användare = ["Adam", "Lisa", "Sven", "Lina"]
```

En lista definieras med hakparenteser i källkoden. Listans innehåll definierar du innanför hakparenteserna. I exemplet ovan ser vi att listan **användare** innehåller fyra strängar.

Datan du stoppar in i en lista kan vara av vilken typ som helst (exempelvis strängar eller nummer). Det är dock sällan en bra idé att blanda datatyperna i en och samma lista då detta ofta gör din kod svårtläst och tyder på dålig kodstruktur.

Om du inte definierar ytterligare data i listan betraktas den som tom:

```
todo = [] # En tom lista, redo att fyllas med data!
```

Senare i kompendiet kommer ni läsa om hur man utför operationer individuellt på samtliga element i en lista. I det här stycket ligger däremot fokus på hur man i överlag hanterar listor och dess element.

8.1.1 Referering av element

När man vill hämta, modifiera eller ta bort ett element från en lista måste man veta vilken position (index) elementet har i listan. Första element i listan har alltid index 0, nästa element har index 1, elementet efter det har index 2, osv:

```
användare = ["Adam", "Lisa", "Sven", "Lina"]

print(användare[0]) # Adam

print(användare[1]) # Lisa

print(användare[2]) # Sven

print(användare[3]) # Lina
```

8.1.2 Modifikation av element

Man kan modifiera elementen i en befintlig lista genom att först referera till elementets index och med likhetstecken sätta det nya värdet:

```
användare = ["Adam", "Lisa", "Sven", "Lina"]

print(användare)
# ['Adam', 'Lisa', 'Sven', 'Lina']

användare[1] = "Ida"

print(användare)
# ['Adam', 'Ida', 'Sven', 'Lina']
```

8.1.3 Tillägg av element

Vill man lägga till ett element till en lista behöver man använda den inbyggda metoden **append()**:

```
användare = ["Adam", "Lisa", "Sven", "Lina"]

print(användare)
# ['Adam', 'Lisa', 'Sven', 'Lina']

användare.append("Sara")

print(användare)
# ['Adam', 'Lisa', 'Sven', 'Lina', 'Sara']
```

8.1.4 Borttagning av element

För att ta bort element från en lista använder man nyckelordet **del** och refererar till index för det element som ska tas bort:

```
användare = ["Adam", "Lisa", "Sven", "Lina"]

print(användare)
# ['Adam', 'Lisa', 'Sven', 'Lina']

del användare[2]

print(användare)
# ['Adam', 'Lisa', 'Lina']
```

8.1.5 Inbyggda metoder

Förutom metoden **append()** som lägger till element till en lista finns också metoder för att omorganisera, sortera och modifiera listor på olika sätt:

```
numbers = [6, 3, 1, 5]

print(numbers)
# [6, 3, 1, 5]

numbers.sort()

print(numbers)
# [1, 3, 5, 6]

numbers.remove(5)

print(numbers)
# [1, 3, 6]
```

8.1.6 Slicing

Utgå från följande lista:

```
namn=["Maria", "Anna", "Eva", "Erik", "Lars", "Karl"]
```

Anta att vi vill dela upp listan i två separata listor:

- kvinnonamn
- mansnamn

Första halvan av listan (index 0 till 2) består av kvinnonamn. Andra halvan av listan (index 3 till 5) består av mansnamn. Med hjälp av slicing kan vi dela upp listan:

```
namn=["Maria", "Anna", "Eva", "Erik", "Lars", "Karl"]

kvinnonamn = namn[0:3]
mansnamn = namn[3:6]

print('Kvinnonamn:')
for k_namn in kvinnonamn:
    print("␣*␣" + k_namn)

print('Mansnamn:')
for m_namn in mansnamn:
    print("␣*␣" + m_namn)
```

8.2 Övningsuppgifter

Utgå från följande grund vid varje övningsuppgift i det här kapitlet. Placera alltid din lösning efter definitionen av **todos**:

```
todos = [
    'Städa',
    'Handla',
    'Plugga',
    'Ge␣blod'
]

# Din lösning här...
```

Övningsuppgift 8.1. Skriv ut följande strängar genom att korrekt referera till dessa från listan (**todos**):

- *'Städa'*
- *'Plugga'*
- *'Handla'*

Övningsuppgift 8.2. Genom interaktion med användare ska ditt script lägga till en sträng till listan.

För att verifiera din lösning behöver du skriva ut strängrepresentationen av listan innan och efter den modifierats.

```
['Städa', 'Handla', 'Plugga', 'Ge blod']

Lägg till ny todo: Hämta hund

['Städa', 'Handla', 'Plugga', 'Ge blod', 'Hämta hund']
```

Övningsuppgift 8.3. Ditt script ska nu låta användare ta bort en todo genom att mata in dess index.

För att verifiera din lösning behöver du skriva ut strängrepresentationen av listan innan och efter den modifierats.

```
['Städa', 'Handla', 'Plugga', 'Ge blod']

Ta bort todo (index): 2

['Städa', 'Handla', 'Ge blod']
```

Övningsuppgift 8.4. Ditt script ska nu låta användare ta bort en todo genom att mata in dess värde.

För att verifiera din lösning behöver du skriva ut strängrepresentationen av listan innan och efter den modifierats.

```
['Städa', 'Handla', 'Plugga', 'Ge blod']

Ta bort todo (värde): Handla

['Städa', 'Plugga', 'Ge blod']
```

Övningsuppgift 8.5. Ditt script ska inledningsvis låta en användare genom inmatning lägga till en ny todo. Därefter ska listan i **todos** sorteras i bokstavsordning.

För att verifiera din lösning behöver du skriva ut strängrepresentationen av listan innan och efter den modifierats.

```
['Städa', 'Handla', 'Plugga', 'Ge blod']

Lägg till todo: Arbeta

['Arbeta', 'Ge blod', 'Handla', 'Plugga', 'Städa']
```

Övningsuppgift 8.6. Genom inmatning ska en användare kunna undersöka om en todo finns i listan. Om den inmatade strängen finns i listan ska användaren informeras om detta. Om strängen inte finns ska programmet fråga om användaren vill lägga till strängen till listan.

För att verifiera din lösning behöver du skriva ut strängrepresentationen av listan innan och efter den modifierats.

```
['Städa', 'Handla', 'Plugga', 'Ge blod']

Ange todo: Spela

'Spela' finns inte i listan.

Vill du lägga till denna (J/N)? J

Todo tillagd!

['Städa', 'Handla', 'Plugga', 'Ge blod', 'Spela']
```

Kapitel 9

Iteration

Det finns lägen när man vill utföra en eller flera handlingar för samtliga element i en lista. Anta att vi har en lista med personer som ska delta vid en föreläsning:

```
deltagare = ["lina", "gunilla", "erik"]
```

Låt säga att jag vill att mitt script ska hälsa på samtliga deltagare. Detta kan uppnås genom att jag skriver en separat print-sats för varje element i listan:

```
deltagare = ["lina", "gunilla", "erik"]

print('Välkommen␣' + deltagare[0])
print('Välkommen␣' + deltagare[1])
print('Välkommen␣' + deltagare[2])
```

Det fungerar bra att göra på detta vis så länge listan är av känd storlek och tillräckligt liten för att det ska vara praktiskt görbart. Skulle listan innehålla 1000 element skulle mitt script bli olidligt lång:

```
deltagare = ["lina", "gunilla", "erik", ..., "örjan"]

print('Välkommen␣' + deltagare[0])
print('Välkommen␣' + deltagare[1])
print('Välkommen␣' + deltagare[2])
# ...
# Rader borttagna
# ...
print('Välkommen␣' + deltagare[999])
```

Det är vanligt i programmering att man utför samma handling för alla element i en lista. Av denna anledning finns det väl etablerade metoder som förenklar vid just denna typ av programmering.

9.1 Listors längd

Innan vi utforskar iteration med **while** behöver vi bekanta oss med funktionen **len()**. Funktionen **len()** låter oss under körning ta reda på en listas längd.

```
deltagare = ["lina", "gunilla", "erik"]

antal_deltagare = len(deltagare) # 3
```

Funktionen returnerar ett heltal som vi bland annat kan skriva ut på skärmen eller (som i exemplet ovan) lagra i variabler.

9.2 Iteration med while

Låt oss för enkelhets skull utgå från listan vi definierade lite tidigare. Är det möjligt att skriva ut samtliga element från listan utan att manuellt referera till index för varje element?

```
deltagare = ["lina", "gunilla", "erik"]
```

I kapitel 4 introducerade vi while-loopen, med vilken vi bland annat kan få att räkna från 0 till 2.

```
i = 0
while i < 3:
    print(i)
    i += 1
```

Utskriften av scriptet blir:

```
0
1
2
```

Har du lagt märke till att vi med while-loopen lyckas skriva ut index för varje element i listan ovan? Detta betyder att vi kan referera till **i** som index för att skriva ut elementen.

```
deltagare = ["lina", "gunilla", "erik"]

i = 0
while i < 3:
    print(deltagare[i])
    i += 1
```

Utskriften av scriptet blir:

```
lina
gunilla
erik
```

Loopen vi precis konstruerade fungerar utmärkt så länge antalet element i **deltagare** är precis 3. Skulle listan ha färre element resulterar detta i att scriptet kraschar. Skulle listan ha fler element kommer inte alla element från listan skrivas ut.

Att manuellt skriva ett maxvärde för **i** i **while**-loopens jämförelseoperator är av denna anledning en dålig idé. Bättre är om vi under körning sätter maxvärdet genom att beräkna listans längd. Det är precis detta som **len()** gör.

```
deltagare = ["lina", "gunilla", "erik", "carl"]

i = 0
while i < len(deltagare):
    print(deltagare[i])
    i += 1
```

9.3 Iteration med for

Det är alltså möjligt att med **while** iterera en listas element:

```
deltagare = ["lina", "gunilla", "erik"]

i = 0
while i < len(deltagare):
    person = deltagare[i]
    print('Välkommen␣' + person)
    i += 1
```

Med nyckelordet **for** kan vi uppnå samma logik fast med färre rader kod:

```
deltagare = ["lina", "gunilla", "erik"]

for person in deltagare:
    print('Välkommen␣' + person)
```

Kommandona och operationerna som definierats i **for**-loopen kommer köras för varje element som finns i listan.

Koden blir oftast mycket kortare och mer lättläst när man använder **for**-loopar. Nackdelen är att man inte kommer åt elementens numeriska index lika enkelt som när man använder en **while**-loop.

9.3.1 Numerisk iteration

Hur skulle du lösa följande uppgift?

> *Skriv ut alla heltal från 0 till och med 9, utan att använda while.*

Det enklaste sättet (beror på hur man ser på saken) är att skriva ut samtliga nummer i separata **print**-satser:

```
print(0)
print(1)
print(2)
print(3)
print(4)
print(5)
print(6)
print(7)
print(8)
print(9)
```

Uppgiften kan också lösas genom att vi specificerar samtliga nummer i en lista som vi sedan itererar med hjälp av en **for**-loop:

```
mängd = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

for nummer in mängd:
    print(nummer)
```

Båda lösningar fungerar! Låt oss formulera om frågan något:

> *Skriv ut alla heltal från 0 till och med 9999, utan att använda while.*

Vi kan inte med enkelhet modifiera tidigare lösningar för att lösa den nya uppgiften. Att manuellt skriva in 10000 nummer är mödosamt och det blir lätt fel.

Med funktionen **range()** kan vi generera en datatyp som är mycket lik listan i det senare exemplet (i och med att vi kan iterera den). Som parameter till funktionen anger man listans storlek:

```
mängd = range(10000)

for nummer in mängd:
    print(nummer)
    # 0
    # 1
    # 2
    # ...
    # 9997
    # 9998
    # 9999
```

9.4 Övningsuppgifter

Övningsuppgift 9.1. Visa hur man med en for-loop kan beräkna:

1. somman av alla heltal mellan 0 och 1000000
2. somman av alla udda heltal mellan 0 och 500

Övningsuppgift 9.2. Utgå från följande kod:

```
registrerade=["Anna", "Eva", "Erik", "Lars", "Karl"]
avanmälningar=["Anna", "Erik", "Karl"]

# Ange din kod här!

print(registrerade)
```

Modifiera scriptet så att listan **registrerade** töms på alla namn som specificerats i listan **avanmälningar**. Uppgiften måste lösas med en **for**-loop.

Övningsuppgift 9.3. Utgå från följande kod:

```
förnamn=["Maria", "Erik", "Karl"]
efternamn=["Svensson", "Karlsson", "Andersson"]

# Din kod här...
```

Visa hur man med nästlade **for**-loopar kan skriva ut samtliga kombinationer av de för och efternamn som specificerats i listorna ovan.

Övningsuppgift 9.4. I övningsuppgifterna till förra kapitlet utförde du ett antal operationer på en todo-lista:

```
todos = ['Städa', 'Handla', 'Plugga', 'Ge blod']
```

Att skriva ut listan med en print-sats är inte särskilt vackert och lämpar sig därför inte för användarvänliga gränssnitt:

```
['Städa', 'Handla', 'Plugga', 'Ge blod']
```

Visa hur man med en for-loop kan snygga till utskriften genom att skriva ut varje todo på en egen rad:

```
.: TODOIFY :.
*************
- Städa
- Handla
- Plugga
- Ge blod
```

Övningsuppgift 9.5. En stack är en mycket vanlig datastruktur (särskilt inom maskinvarunära programmeringsspråk som C och C++) som innehåller en linjär ordnad följd av element (likt listor i Python). Till skillnad från Python-listor (där man kan ta bort och lägga till element vid vilken position som helst) följer en stack ”sist in, först ut” principen. En stack stödjer två operationer:

- push
- pop

Push innebär att man lägger till ett element i slutet av stacken. Python-listors motsvarighet till detta är dess inbyggda metod **append()**.

```
bilar = ['Mercedes', 'Volvo', 'Toyota']
bilar.append('Kia')
print(bilar) # ['Mercedes', 'Volvo', 'Toyota', 'Kia']
```

Pop innebär att man plockar bort ett element i slutet av stacken. Python-listors motsvarighet till detta är dess inbyggda metod **pop()**.

```
bilar = ['Mercedes', 'Volvo', 'Toyota']
bilar.pop()
print(bilar) # ['Mercedes', 'Volvo']
```

Konstruera ett program där en användare interaktivt kan lägga till och ta bort strängar från en list-baserad stack. Ta gärna inspiration från vårt interaktiva exempel[1].

```
  .: STACKMASTER v1.33.7 :.
------------------------------
 - Mercedes
 - Volvo
 - Toyota
------------------------------
           | MENU |
------------------------------
push | Push element to stack
pull | Pull element from stack
exit | Exit program
------------------------------
MENU >
```

[1] https://asciinema.org/a/yftNX2US4kdfBOCYz6jBk67WS

Kapitel 10

Filhantering

När man i script behöver memorera data, som exempelvis strängar, nummer eller listor, så används variabler. Variabler lagras (vanligtvis) i datorns RAM-minne, vilket är datorns snabbaste lagringsplats.

Om nu RAM är så snabbt, varför körs inte hela operativsystemet och dess processer i RAM-minnet?

Kostnad Att köpa ett RAM-minne med storleken 16 gigabyte kostar ungefär lika mycket som att köpa en hårddisk av typen SSD med storleken 1 terrabyte. Och då ska man vara medveten om att SSD-diskar hör till de dyrare typerna av hårddiskar. På grund av dess höga kostnad brukar vanliga persondatorer inte ha mer än 8 gigabyte RAM. Åtta gigabyte RAM skulle inte ens räcka till att lagra operativsystemet Windows 10 (som kräver lagringsutrymme om minst 32 gigabyte).

Flyktigt Ett RAM-minne är beroende av en kontinuerlig elektrisk spänning för att memorera data. Så även om du skulle hitta ett sätt att installera Windows 10 på ett RAM-minne så skulle RAM-minnet ”glömma” all data när strömmen går eller när du stänger av din dator. Datorminnen som förlorar sin data när datorn stängs av kallas i datorterminologi för flyktiga.

Mekaniska hårddiskar och SSD-diskar är exempel på icke-flyktiga minnen eftersom dessa datorminnen behåller sin data även utan elektrisk spänning. Av den anledningen lagras operativsystem och filer på dessa typer av diskar i datorn.

RAM-minnet används endast som en temporär lagringsplats för processer att snabbt lagra i och hämta data från. Moderna operativsystem tillämpar tekniker för att isolera RAM-minnet mellan processer. Det vore en säkerhetsrisk om vilken process som helst obehindrat kan börja läsa och modifiera RAM-minnet för exempelvis Google Chrome.

Moderna operativsystem tillämpar också tekniker för att rensa RAM-minnet för processer efter dessa stängts. För att behålla information mellan omstarter behöver du därför spara undan data till ett icke-flyktiga minne. Detta gör du enklast genom att skriva din data till en fil i operativsystemet.

10.1 Att läsa från fil

För att öppna en fil i Python använder vi den inbyggda funktionen **open()**. Funktionen kommer bland annat kommunicera med det underliggande operativsystemet för att komma åt den binära datan från datorns hårddisk. Som tur är behöver du inte förstå hur **open()** gör allt detta. Det du däremot behöver förstå är hur du arbetar med den datatyp som **open()** returnerar.

Funktionen **open()** returnerar ett objekt som agerar som en direktlänk mot filen du vill arbeta med. Första argumentet i funktionen är en sträng innehållande sökvägen mot filen du vill öppna:

```
f = open('en-mapp/fil.txt')
```

Variabeln **f** innehåller nu ett objekt med tillhörande metoder för att arbeta med filen vi precis öppnade. Metoden **read()** returnerar en sträng bestående av filens innehåll:

```
f = open('en-mapp/fil.txt')
MinVariabel = f.read()
print(MinVariabel)
```

När vi arbetat färdigt med filen behöver vi berätta detta för operativsystemet. Metoden **close()** stänger vår koppling mot filen:

```
f = open('en-mapp/fil.txt')
MinVariabel = f.read()
print(MinVariabel)
f.close()
```

10.2 Att skriva till fil

Har vi tänkt skriva till en fil måste vi i berätta detta för operativsystemet i samband med att vi öppnar filen. Detta gör vi genom att lägga till strängen *'w'* (write) som andra argument till **open()**:

```
f = open('en-mapp/fil.txt', 'w')
```

För att skriva en sträng till filen vi öppnat använder vi metoden **write()**. Kom ihåg att stänga filen när du skrivit färdigt till den:

```
f = open('en-mapp/fil.txt', 'w')
f.write('God␣dag.')
f.close()
```

10.3 Filhantering med with

Nyckelordet **with** är användbart när man arbetar med filer:

```
with open('en-mapp/fil.txt') as f:
    MinVariabel = f.read()

print(MinVariabel)
```

I exemplet lagras fil-objektet och dess metoder i variabeln **f**. Alla handlingar vi utför mot filen är indenterade i with-satsen. Python kommer automatiskt stänga kopplingen mot filen och tömma variabeln **f** när with-satsen tagit slut. Att använda sig av **with** minimerar och förenklar koden och är därför att rekommendera i de flesta fall.

Det går lika bra att skriva till filer med **with**:

```
attendants = ['Lisa', 'Kalle', 'Olivia', 'Johan']

with open('en-mapp/fil.txt', 'w') as f:
    for attendant in attendants:
        f.write('Hello␣' + attendant + '!\n')
```

10.4 Övningsuppgifter

Övningsuppgift 10.1. Skapa ett script som visar en vägskylt enligt följande:

```
  |-----------------------------------------|
  |  #  --------------------------      #    |
  | ### |   Welcome to Västerås   |  #  ###   |
  | ### -------------------------- ### ###   |
  |  |         |          |          |   |  # |
  |-----------------------------------------|
  | C | Change sign message
  | E | Exit program
  |-----------------------
  | >
```

Implementera funktionalitet för att ändra vägskyltens meddelande. Meddelandet ska lagras i en fil med namn **sign.txt**. Eftersom meddelandet lagras i en fil så ska såklart meddelandet som användaren sätter bestå mellan körningar, vilket visas i detta interaktiva exempel[1].

Övningsuppgift 10.2. Ladda ner följande textfil till din dator:

http://dva128.s3.eu-north-1.amazonaws.com/kompendium/numbers.csv

Filen innehåller en miljon heltal mellan 0 till 9 som randomiserats av en slumpgenerator. Din uppgift är att skriva ett script som beräknar förekomsten av varje heltal och presenterar detta för användaren likt exemplet nedan:

```
---------------
- NUMANALYZER -
---------------
| 0 | 100336
| 1 | 100078
| 2 | 100055
| 3 | 99918
| 4 | 100102
| 5 | 99432
| 6 | 100036
| 7 | 100394
| 8 | 99549
| 9 | 100100
---------------
```

[1] https://asciinema.org/a/NYQBtrjGUeZV3fZF3aDE4EjfH

Övningsuppgift 10.3. Ladda ner följande textfil till din dator:

http://dva128.s3.eu-north-1.amazonaws.com/kompendium/database.csv

Filen är en databas innehållandes 20000 fiktiva personer:

```
ID,FORENAME,SURNAME,GENDER,YEAR
0,Robin,Viklund,male,1965
1,Marie,Henriksson,female,1962
2,Ann-Christin,Karlsson,female,1988
3,Per,Mattsson,male,2010
...
```

Varje rad representerar en person. Personens attribut separeras med komma. Radens första attribut innehåller ett unikt ID varje person tilldelas när dessa läggs till i textfilen. Andra och tredje kolumnen innehåller personens för och efternamn. Fjärde kolumnen innehåller personens kön. Sista kolumnen innehåller personens födelseår.

Din uppgift är att skapa ett program som hjälper användare söka personer i textfilen. Ditt program ska klara att:

- söka och skriva ut information om en person baserat på dennes (av användaren inmatat) ID.
- skriva ut alla personer som matchar ett (av användaren) inmatat för- eller efternamn.

```
.: PEOPLES DATABASE :.
----------------------
get_id | Get person by ID
scan_f | List people by FORENAME
scan_s | List people by SURNAME
exit   | Exit program
----------------------
| >
```

Lägg gärna till fler sökfunktioner om du är ambitiös. Titta gärna på vårt interaktiva lösningsförslag innan du börjar[2].

[2]https://asciinema.org/a/nLNpjJRtL7kIt2XLrhbyCOO5s

Kapitel 11

JavaScript Object Notation

Variabeln **l33t** innehåller en lista med fyra strängar:

```
l33t = ['1', '3', '3', '7']
```

Skriv ett program som lagrar listan **l33t** i en fil med namn **lista.txt**. Listan ska vara formaterad på ett sådant sätt att listan enkelt kan läsas in och återskapas av ett annat script.

Nu när vi precis lärt oss hur **with** fungerar så kan man försöka skriva listan till filen med metoden **write()**:

```
with open('file.txt', 'w') as f:
    f.write(l33t)
```

Detta förorsakar desvärre ett felmeddelande:

```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    f.write(l33t)
TypeError: write() argument must be str, not list
```

Metoden **write()** förväntar sig att datan vi skriver till filer är av typen sträng. Alla andra datatyper kommer generera ett liknande felmeddelande. Variabeln **l33t** är av datatypen lista, inte sträng, därav felmeddelandet. Om vi först datatypsomvandlar **l33t** med den inbyggda funktionen **str()** kan vi skriva till filen:

```
l33t = ['1', '3', '3', '7']

with open('file.txt', 'w') as f:
    f.write(str(l33t))
```

Scriptet kör utan krasch och genererar filen **file.txt** med följande innehåll:

```
['1', '3', '3', '7']
```

Enligt kravspecifikationen ska listan vara formaterad på ett sådant sätt (i textfilen) att den enkelt kan läsas in och återskapas av ett annat script. I ett försök att göra detta börjar vi med att skapa ett script som läser in filens innehåll:

```
with open('file.txt') as f:
    l33t = f.read()
print(l33t)
# OUTPUT: ['1', '3', '3', '7']
```

Inläsningen från filen fungerade bra, men **l33t** innehåller endast en sträng (som ser ut som en lista) såhär lång. Vi kan exempelvis inte referera till listans första element. Istället får vi strängens första karaktär:

```
print(l33t[0])
# OUTPUT: [
```

Vi behöver hitta ett sätt att datatypsomvandla **l33t** från sträng tillbaks till lista. Funktionen **list()** kan användas för att omvandla itererbara objekt till listor.

```
with open('file.txt') as f:
    l33t = list(f.read())
print(l33t)
# OUTPUT: ['[', "'", '1', "'", ',', ' ', "'", '3' ...
```

Det där ser ju inte riktigt klokt ut. Vi fick en lista där varje karaktär i strängen är sitt eget element. För att gå vidare har vi två val:

1. Vi skriver logik som tolkar strängen och konstruerar en korrekt lista åt oss.
2. Vi börjar söka efter inbyggda funktioner som med beprövade metoder och industriell praxis sköter datatypsomvandlingar (mellan bland annat listor och strängar) åt oss.

Jag tycker att Alternativ 2 låter som det bättre förslaget. Tillåt mig därför introducera JavaScript Object Notation (JSON).

11.1 Textformatet JSON

JSON är en öppen standard som reglerar hur listor och dictionaries (som ni läser om i nästa kapitel) representeras i strängformat. Douglas Crockfold specificerade formatet redan 2000. JSON blev dock standardiserat först 2013 i ECMA-404, av samma organisation som ansvarar för nya JavaScript-standarder.

JSON-formatet har stöd för sex datatyper:

- nummer

- strängar
- booelska-datatyper
- listor
- dictionaries
- null-typen

I exemplet nedan är **MyList1** en vanlig Python-lista med element av olika datatyper. **MyList2** är en JSON-formaterad strängrepresentation av **MyList1**:

```
MyList1 = ['Christian', 'Åberg', 1337, None]
MyList2 = '["Christian", "\u00c5berg", 1337, null]'
```

Vad är skillnaderna?

1. Datatyperna skiljer sig åt. **MyList1** är av datatypen lista medans **MyList2** är av datatypen sträng.
2. Python stödjer både apostrof och citationstecken för strängar. JSON stödjer endast citationstecken.
3. JSON använder speciella koder för att representera specialkaraktärer. I exemplet kan vi se att *'Å'* omvandlats till *'\u00'*.
4. Pythons datatyp **None** är ekvivalent med JSON-datatypen **null**.

11.1.1 Datatypen list till textformatet JSON

Låt säga att vi har en variabel **MyVar** som vi vill omvandla till en JSON-formaterad sträng:

```
MyVar = ['Christian', 'Åberg', 1337, None]
```

Python har inbyggda funktioner för att göra detta. För att komma åt dessa funktioner behöver vi importera biblioteket **json**. Bilbioteket har en funktion **dumps()** som omvandlar datan vi anger som argument till en JSON-formaterad strängrepresentation:

```
import json

MyVar = ['Christian', 'Åberg', 1337, None]
MyVar = json.dumps(MyVar)
print(MyVar)
# OUTPUT: ["Christian", "\u00c5berg", 1337, null]
```

Efter omvandlingen är **MyVar** en sträng.

11.1.2 Textformatet JSON till datatypen list

Hur gör vi för att omvandla följande JSON-sträng till motsvarande objekt i Python?

```
["Christian", "Aberg", 1337, null]
```

För att omvandla en JSON-formaterad sträng till en lista använder vi funktionen **loads** från biblioteket **json**. Som argument till **loads** anger vi den JSON-formaterade strängen.

```
import json

MyVar = '["Christian", "Aberg", 1337, null]'

MyVar = json.loads(MyVar)

for item in MyVar:
    print('-', item)
    # Christian
    # Aberg
    # 1337
    # None
```

Efter omvandlingen är **MyVar** en lista.

11.1.3 Lagra list i fil med JSON

Med vår ny-vunna kunskap om hur vi kan omvandla listor till JSON-formaterade strängar tycker jag vi försöker skriva om scriptet vi skapade i början på det här kapitlet:

```
l33t = ['1', '3', '3', '7']

with open('file.txt', 'w') as f:
    l33t = str(l33t) # inte särskilt bra...
    f.write(l33t)
```

Vi tar och omvandlar **l33t** till en sträng med **json.dumps()** istället för **str()**. Eftersom vår sparade data nu följer JSON-specifikationen är det läge att byta filens namn till **file.json**.

```
import json

l33t = ['1', '3', '3', '7']

with open('file.json', 'w') as f:
    l33t = json.dumps(l33t)
    f.write(l33t)
```

Öppnar vi **file.json** ser vi följande:

```
["1", "3", "3", "7"]
```

11.1.4 Läsning av JSON-formaterad list från fil

Filen **file.json** innehåller nu följande text:

```
["1", "3", "3", "7"]
```

Vi tar nu och läser in filen. Datan lagras i variabeln **l33t** som en sträng:

```
with open('file.json') as f:
    l33t = f.read()

print(l33t)
# ["1", "3", "3", "7"]
print(l33t[0])
# [
```

Eftersom strängen innehåller en JSON-formaterad lista kan vi omvandla denne till en Python-lista med funktionen **loads()** från tredjepartsbiblioteket **json**:

```
import json

with open('file.json') as f:
    l33t = f.read()
    l33t = json.loads(l33t)

print(l33t)
# ['1', '3', '3', '7']
print(l33t[0])
# 1
```

11.2 Övningsuppgifter

Övningsuppgift 11.1. Variabeln **random_stuff** är en lista innehållande data av olika typer:

```
random_stuff = [1337, 13.37, 'Ååh Yää!']
```

Visa hur man med biblioteket **json** omvandlar listan till motsvarande JSON-formaterade strängrepresentation. Skriv därefter ut JSON-objektet på skärmen:

```
[1337, 13.37, "\u00c5\u00e5h Y\u00e4\u00e4!"]
```

Övningsuppgift 11.2. Variabeln **my_chars** innehåller en JSON-formaterad strängrepresentation av en lista:

```
my_chars = '["abc", "\u00e5\u00e4\u00f6", "123"]'
```

Visa hur man med biblioteket **json** kan omvandla JSON-objektet till en lista i Python. Skriv därefter ut varje objekt på en ny rad genom att iterera listan:

```
abc
ääö
123
```

Övningsuppgift 11.3. Konstruera ett script där användare matar in en serie heltal. Samtliga inmatade heltal samt summan av dessa ska presenteras för användaren:

```
.: intMEMORIZER :.
******************
* 1
* 3
* 7
------------------
SUMMA: 11
------------------
  mata in heltal
0 stänger scriptet
------------------
>
```

Endast unika heltal ska memoreras; dubletter ska alltså ignoreras. Matar användaren in heltalet 0 ska scriptet stängas.

Ditt script ska memorera heltalen mellan körningar. Heltalen behöver därför lagras i ett icke-flyktigt minne och formateras på ett sådant sätt att varje script kan ta vid där förra scriptet slutade.

Testa gärna vårt interaktiva lösningsförslag[1].

[1] https://dva128demo.mdh.repl.co/#week4/intMEMORIZER

Kapitel 12

Dictionaries

I det här kapitlet introducerar vi en ny datatyp (datatypen dictionary). Men innan vi börjar titta på den nya datatypen är det på sin plats med en kort repetition om hur listor fungerar i Python.

12.1 Återblick till datatypen list

Innan man börjar läsa om dictionaries är det en god idé att först se till att man har fullt förstår grunderna till hur listor fungerar i Python, då det finns många likheter mellan datatyperna. Listor arbetade du med i Kapitel 8, men det är alltid bra med lite repetition.

En lista är en abstrakt datatyp som innehåller en mängd andra datatyper:

```
car_manufacturers = ["Toyota", "Volkswagen", "Hyundai"]
```

Listor är ordnade i den mening att du hämtar element i samma ordning som du matar in dem. En lista kan också specificeras över flera rader för läslighets skull:

```
car_manufacturers = [
    "Toyota",
    "Volkswagen",
    "Hyundai"
]
```

Det finns en mängd fördelar med att använda sig av listor. Bland annat kan man lägga till och ta bort element från listor under körning. Men den absolut viktigaste funktionen listor erbjuder är iteration av listans element:

```
for car_manufacturer in car_manufacturers:
  print(car_manufacturer + "␣is␣a␣car␣manufacturer.")
```

Det är svårt (särskilt när man är en oerfaren programmerare) att lista ut när listor och iteration är den föredragna lösningen till ett givet problem. Men med tiden kommer du märka hur effektiva listor är på att minimera och förenkla kod.

12.2 Datatypen dict

Dictionaries är (precis som listor) en abstrakt datatyp. Till skillnad från listor låter inte dictionaries dig definiera ordnade element. Istället består dictionaries av nyckel-/värde-par:

```
person = {"namn": "Christian", "ålder": 25}

print("Hej! Jag heter " + person["namn"] + "!")
print("Jag är " + str(person["ålder"]) + " år gammal.")
```

Precis som listor så kan dictionaries skrivas ut över flera rader (detta är starkt rekommenderat för läslighetens skull):

```
person = {
    "namn": "Christian",
    "ålder": 25
}
```

12.2.1 Hämtning av element

För att referera mot ett element i ett dictionary refererar man mot dess nyckel enligt följande:

```
print(person['namn'])
# Christian
```

Notera att det också går bra att dynamiskt referera mot en nyckel som är lagrad i en variabel.

```
person = {
    'förnamn': 'Lisa',
    'efternamn': 'Svensson',
    'ålder': 32
}

attr = input('Ange attribut > ')

try:
    print(person[attr])
except KeyError:
    print('FEL: Attribut existerar inte')
```

Försöker du referera mot en nyckel som inte finns kommer Python lyfta ett felmeddelande av typen **KeyError**. Låter du en användare mata in en nyckel kan det vara bra att fånga dessa felmeddelanden, vilket visas av exemplet ovan.

12.2.2 Elementmodifikation

För att sätta värdet på ett element refererar du först till elementen och sätter värdet enligt följande:

```
car = {
    'type': 'Volvo',
    'year': 2005
}

car['type'] = 'Ford'

print(car)
#
# {
#     'type': 'Ford',
#     'year': 2005
# }
#
```

12.2.3 Tillägg av element

Försöker du modifiera en nyckel som inte existerar läggs nyckeln och värdet till i dictionaryt.

```
vector = {
    'x': 5,
    'y': 2
}

vector['z'] = 3

print(vector)
#
# {
#     'x': 5,
#     'y': 2,
#     'z': 3
# }
#
```

12.2.4 Borttagning av element

Med nyckelordet **del** kan du ta bort ett element från ett dictionary.

```
character = {
    'name': 'Grindhuld',
    'type': 'orc',
    'occupation': 'warrior'
}

del character['occupation']

print(character)
# {
#     'name': 'Grindhuld',
#     'type': 'orc'
# }
```

12.2.5 Iteration av dictionary

Med en **for**-loop kan man iterera ett dictionarys nycklar:

```
character = {
    'forename': 'Grindhuld',
    'surename': 'orc',
    'occupation': 'warrior'
}

for key in character:
    print(key)
    # forename
    # surename
    # occupation
```

Med en liten modifikation av **for**-loopen kan vi istället iterera samtliga värden:

```
character = {
    'forename': 'Grindhuld',
    'surename': 'orc',
    'occupation': 'warrior'
}

for key in character:
    print(character[key])
    # Grindhuld
    # orc
    # warrior
```

12.2.6 Nästlade dictionaries

Ett dictionaries element kan vara av valfri datatyp. Ett dictionary kan alltså innehålla ytterligare dictionaries (detta kallas för nästlade dictionaries):

```
person = {
    'name': 'Lisa Svensson',
    'resident': {
        'type': 'apartment',
        'rent': 5500
    }
}
```

För att referera till det nästlade dictionaryt behöver vi ange dess nyckel inom hakparenteser till variabeln **person**:

```
print(person['resident'])
# {
#     'type': 'apartment',
#     'rent': 5500
# }
```

För att referera till det nästlade dictionaryts element lägger vi till ytterligare hakparenteser:

```
print(person['resident']['type']) # 'apartment'
```

Det finns ingen begränsning för hur många extra hakparenteser du kan lägga till (så länge det finns ytterligare dictionaries att referera mot).

12.2.7 Lista i dictionary

Ett dictionary kan innehålla en lista. Samma logik som vi lärde oss i sektion 12.2.6 gäller här. Försök se om du kan lista ut hur man refererar till den öppna porten *1337*:

```
server = {
    'type': 'firewall',
    'open_ports': [
        1000,
        1234,
        1337
    ]
}
```

Heltalet *1337* är ett element i listan med nyckel *'open_ports'*:

```
print(server['open_ports']) # [1000, 1234, 1337]
```

Avslutningsvis lägger vi till en hakparentes där vi refererar mot index för *1337*:

```
print(server['open_ports'][2])
```

12.3 Format för dict

Tänk er att en variabel **person** innehåller följande objekt:

```
person = {
    "förnamn" : "Johan",
    "efternamn" : "Svensson",
    "ålder" : 25,
    "husdjur" : {
        "namn" : "Morris",
        "ålder" : 3,
        "typ" : "hund"
    }
}
```

Variabeln innehåller ett dictionary som beskriver en person med namn Johan Svensson. Av objektet kan man läsa ut både för- och efternamn. Även ålder och detaljerad information om personens husdjur ingår. Har personen flera husdjur kan det vara fördelaktigt att istället låta nyckeln **husdjur**:s värde att vara av datatypen lista:

```
person = {
    "förnamn" : "Johan",
    "efternamn" : "Svensson",
    "ålder" : 25,
    "husdjur" : [
        {
            "namn" : "Morris",
            "ålder" : 3,
            "typ" : "hund"
        },
        {
            "namn" : "Lisa",
            "ålder" : 2,
            "typ" : "katt"
        },
    ]
}
```

I nästa kapitel kommer ni hämta data från något som kallas API:er. Dessa API:er kommer svara era script med data likt det ni ser ovan. Vi kan för inlärningens skull låtsas att variabeln **person** innehåller svaret från ett API genom vilket man kan hämta ut information om husdjursägare som registrerats hos ett populärt community.

För att script ska kunna tolka API:ets svar behöver objektens struktur först fastställas och därefter dokumenteras av API:ets utvecklare. Detta kan exempelvis göras med hjälp av en punktlista. Varje punkt i listan nedan representerar en obligatorisk nyckel för objektet. I varje parentes fastställs vilken datatyp som hör till nycklarna. Stanna gärna upp och jämför punktlistan mot objektet i **person** så du förstår hur dokumentationen korrelerar mot det faktiska objektet:

- förnamn *(string)*
- efternamn *(string)*
- ålder *(int)*
- husdjur *(list of dict)*
  - namn *(string)*
  - ålder *(int)*
  - typ *(string)*

Oberoende för vilken person du hämtar information om kan du vara säker på att objektets struktur följer reglerna:

```
ny_person = {
    "förnamn" : "Gunilla",
    "efternamn" : "Ek",
    "ålder" : 53,
    "husdjur" : [
        {
            "namn" : "Putte",
            "ålder" : 1,
            "typ" : "katt"
        }
    ]
}
```

Det enda som formatmässigt skiljer objekten åt är att listan i nyckeln *"husdjur"* har olika antal element. Detta strider dock inte mot reglerna, eftersom längden för nyckeln *"husdjur"* inte är fastställd. Om en person i communityt inte har några husdjur kan listan till och med vara tom:

```
ny_person = {
    "förnamn" : "Ida",
    "efternamn" : "Olsson",
    "ålder" : 33,
    "husdjur" : []
}
```

12.4 Tolkning av dict

Ska vi få våra Python-script att anropa och tolka svar från API:er är det viktigt att vi vet hur man med kod läser data från dictionaries. Variabeln **person** följer formatet som presenterades i föregående stycke:

```
person = {
    "förnamn" : "Johan",
    "efternamn" : "Svensson",
    "ålder" : 25,
    "husdjur" : [
        {
            "namn" : "Morris",
            "ålder" : 3,
            "typ" : "hund"
        },
        {
            "namn" : "Lisa",
            "ålder" : 2,
            "typ" : "katt"
        },
    ]
}
```

Vi ska modifiera koden så att scriptet under körning tolkar dictionary-objektet som är lagrat i variabeln **person**. Genom att tolka objektet ska programmet skriva ut följande:

```
Johan Svensson är 25 år gammal och har 2 husdjur:
* En 3 år gammal hund som heter Morris
* En 2 år gammal katt som heter Lisa
```

Programmet ska tolerera att nyckeln *"husdjur"* har olika längd mellan körningar. Med andra ord ska programmet inte utgå från att listan alltid har två element.

12.4.1 Hämta data ur dict

Inledningsvis ska vi endast hämta och skriva ut strängarna som lagrats i nycklarna *"förnamn"*, *"efternamn"* och *"ålder"*. Antalet husdjur får vi ut genom att räkna elementen i listan *"husdjur"*:

```
namn = person["förnamn"] + "␣" + person["efternamn"]
hlen = len(person["husdjur"]) # antal husdjur
ålder = person["ålder"]

print(namn, "är", ålder, "år␣gammal␣och␣har", hlen, "husdjur:")
```

12.4.2 Iterera list i dict

Nu ska vi skriva ut information om varje husdjur som lagrats i nyckeln *”husdjur”*. För att referera till själva listan skriver vi:

```
person["husdjur"]
#[
#   {'namn': 'Morris', 'ålder': 3, 'typ': 'hund'},
#   {'namn': 'Lisa', 'ålder': 2, 'typ': 'katt'}
#]
```

För att referera till de enskilda elementen i listan skriver vi:

```
person["husdjur"][0]
# {'namn': 'Morris', 'ålder': 3, 'typ': 'hund'}
person["husdjur"][1]
# {'namn': 'Lisa', 'ålder': 2, 'typ': 'katt'}
```

För att ytterligare referera till elementens nyckel/värde-par skriver vi:

```
person["husdjur"][0]["namn"]
# 'Morris'
person["husdjur"][1]["namn"]
# 'Lisa'
```

Vi kan utnyttja denna egenskap för att skriva ut detaljerad information om varje husdjur:

```
h_namn = person["husdjur"][0]["namn"]
h_ålder = person["husdjur"][0]["ålder"]
h_typ = person["husdjur"][0]["typ"]
print("* En", h_ålder, "år gammal", h_typ, "som heter", h_namn)

h_namn = person["husdjur"][1]["namn"]
h_ålder = person["husdjur"][1]["ålder"]
h_typ = person["husdjur"][1]["typ"]
print("* En", h_ålder, "år gammal", h_typ, "som heter", h_namn)
```

Problemet med föregående exempel är att vi förutsätter att listan i nyckeln *”husdjur”* endast har två element. Skulle listan endast ha ett element skulle vårt program krascha (när vi försöker referera till element som inte existerar). Skulle listan ha fler än två element skulle endast de två första skrivas ut. Hur kan vi lösa detta?

Detta är ett bra exempel på där iteration hjälper:

```
for h in person["husdjur"]:
    print("* En", h['ålder'], "år gammal", h['typ'],
    "som heter", h['namn'])
```

Med iteration kommer samma print-sats skrivas ut, oberoende listans storlek. Koden blir kortare och samtidigt mer lättläst.

12.4.3 Lösning på uppgift

Med detta har vi nu en fullständig lösning till uppgiften som presenterades i början på detta stycke:

```
person = {
    "förnamn" : "Johan",
    "efternamn" : "Svensson",
    "ålder" : 25,
    "husdjur" : [
        {
            "namn" : "Morris",
            "ålder" : 3,
            "typ" : "hund"
        },
        {
            "namn" : "Lisa",
            "ålder" : 2,
            "typ" : "katt"
        },
    ]
}

namn = person["förnamn"] + "␣" + person["efternamn"]
hlen = len(person["husdjur"]) # antal husdjur
ålder = person["ålder"]

print(namn, "är", ålder, "år␣gammal␣och␣har", hlen, "husdjur:")

for h in person["husdjur"]:
    print("*␣En", h['ålder'], "år␣gammal", h['typ'],
    "som␣heter", h['namn'])
```

Kör du scriptet ovan får vi samma utskrift som kraven för uppgiften:

```
Johan Svensson är 25 år gammal och har 2 husdjur:
* En 3 år gammal hund som heter Morris
* En 2 år gammal katt som heter Lisa
```

12.5 Övningsuppgifter

Dictionaries lämpar sig väl för att lagra namngivna anteckningar likt Evernote eller Google Keep. De metoder och funktionaliteter som följer med dictionaries gör det enkelt att lägga till, ta bort samt modifiera anteckningar som lagras i en sådan.

Nedan är ett exempel på hur namngivna anteckningar kan lagras i ett dictionary. Varje element i dictionariet utgör en anteckning. Elementets nyckel utgör anteckningens titel. Elementets värde utgör anteckningens innehåll:

```
notes = {
    'Meddelande␣från␣skolan': 'Friluftsdag␣på␣tisdag',
    'Kom␣ihåg!': 'Ta␣bilen␣till␣verkstad',
    'Inför␣tentamen': 'Gör␣alla␣instuderingsuppgifter'
}
```

Utgå från variabeln i samtliga övningsuppgifter i det här kapitlet. Lägg till din lösning efter variabelns definition.

Övningsuppgift 12.1. Låt en användare välja vilken rubrik som ska skrivas ut genom att mata in dess titel:

```
Anteckning > Kom ihåg!
-----
Ta bilen till verkstad
-----
```

Om användaren matar in en rubrik som inte existerar ska ett felmeddelande skrivas ut:

```
Anteckning > Att handla
FEL: Anteckning finns inte
```

Övningsuppgift 12.2. Visa hur man genom att iterera dictionaryt med en for-loop kan lista titlar för samtliga anteckningar:

```
.: ANTECKNINGAR :.
******************
- Meddelande från skolan
- Kom ihåg!
- Inför tentamen
------------------
```

Övningsuppgift 12.3. Visa hur man genom iteration av dictionaryt kan skriva ut samtliga anteckningars titel och text:

```
-----
Titel: Meddelande från skolan
Text:  Friluftsdag på tisdag
-----
Titel: Kom ihåg!
Text:  Ta bilen till verkstad
-----
Titel: Inför tentamen
Text:  Gör alla instuderingsuppgifter
```

Övningsuppgift 12.4. Låt en användare lägga till (eller ändra) en anteckning genom att mata in dess titel och text. Skriv avslutningsvis ut samtliga artiklar för att bekräfta så att förändringen slog igenom:

```
Lägg till artikel:
  titel > Inför resa
  text  > Packa kläder och strumpor
-----
Titel: Meddelande från skolan
Text:  Friluftsdag på tisdag
-----
Titel: Kom ihåg!
Text:  Ta bilen till verkstad
-----
Titel: Inför tentamen
Text:  Gör alla instuderingsuppgifter
-----
Titel: Inför resa
Text:  Packa kläder och strumpor
```

Övningsuppgift 12.5. Låt en användare ta bort en anteckning genom att mata in dess titel. Bekräfta att artikeln togs bort genom att skriva ut samtliga artiklar:

```
Ta bort artikel: Kom ihåg!
-----
Titel: Meddelande från skolan
Text:  Friluftsdag på tisdag
-----
Titel: Inför tentamen
Text:  Gör alla instuderingsuppgifter
```

Övningsuppgift 12.6. Konstruera ett fullständigt program med tillhörande användargränssnitt där användare kan hantera personliga anteckningar. Varje anteckning ska bestå av två komponenter:

- Titel
- Text (anteckningens innehåll)

Genom användargränssnittet ska användaren kunna lista, visa, skapa, ändra och ta bort anteckningar.

Nedan följer ett exempel på hur scriptets användargränssnitt kan se ut:

```
.:  ALWAYSNOTE  :.
-- gold edition --
******************
- Meddelande från skolan
- Kom ihåg!
- Inför tentamen
------------------
view | view note
add  | add note
rm   | remove note
exit | exit program
------------------
menu >
```

Programmet ska memorera anteckningar mellan körningar. Anteckningarna behöver därför lagras i ett icke-flyktigt minne och formateras på ett sådant sätta att varje script kan ta vid där förra scriptet slutade.

Tips! Dictionaries kan (precis som listor) omvandlas till JSON-formaterade strängar. Strängar är enkla att spara till (och läsa från) filer.

Testa gärna vårt interaktiva lösningsförslag[1].

[1] https://dva128demo.mdh.repl.co/#week5/alwaysnote

Kapitel 13

Application Programming Interfaces

Ett Application Programming Interface (API) är en kommunikationspunkt dit script och appar kan vända sig för att skicka och hämta information av olika sort. Uttrycket API har många tillämpningar inom programmering, vilket kan skapa en del förvirring. För tydlighets skull vill jag poängtera att vi i den här kursen syftar på API:er som levereras över HyperText Transfer Protocol. Samma protokoll som används av webbläsare för att hämta ner källkoden till hemsidor:

```
http://www.google.com/
```

Idag använder de flesta hemsidor HyperText Transfer Protocol Secure (HTTPS) vilket är en krypterad och säkrare variant av HTTP:

```
https://www.google.com/
```

När din webbläsare hämtar ner källkoden till en hemsida svarar webbservern med märkspråket HyperText Markup Language (HTML). Som ni ser i exemplet nedan är HTML vanlig text:

```
<html>
  <body>
    <img src="logo.jpg" alt="Logotyp">
    Välkommen till min hemsida.
  </body>
</html>
```

Som webbdesigner arbetar man med att designa hemsidor genom att definiera hemsidans utseende med hjälp av HTML.

13.1 Representational State Transfer

I och med att webben växte så utökades tillämpningsområdet för HTTP till mer än att bara leverera källkoden för hemsidor till webbläsare. Till stor del beror denna förändring på att fler och fler personer började använda appar framför webbläsare för att konsumera webbtjänster.

År 2000 introducerade och definierade Roy Thomas Fielding uttrycket Representational State Transfer (REST). REST är en modell för mjukvara som ofta ligger till grund för många webbtjänster som levererar API:er över HTTP. Bland de webbtjänster som uttryckligen följer standarden hör:

- Facebook Messenger
- Twitter
- Amazon Web Services
- Google Cloud Platform
- Microsoft Azure

Den 25:e november 2015 antog Europaparlamentet och Europeiska unionens råd direktivet (EU) 2015/2366 om betaltjänster på den inre marknaden (PSD 2). Direktivet syftar till att utveckla marknaden för elektroniska betalningar och skapa förutsättningar för säkra och effektiva betalningar. Direktivet tvingar företag (under vissa omständigheter) att ge tredjepartsbetaltjänstleverantörer tillgång till kundernas konton. För att uppfylla direktivets krav tillhandahåller bland andra Swedbank och Nordea HTTP-baserade API:er av typen REST för tredjepartsleverantörer att kommunicera mot.

Kunskaper om hur man anropar API:er av typen REST är värdefulla för dig som ser en framtid inom mjukvaruutveckling, artificiell intelligens, datornätverk eller inom finanssektorn. Eftersom detta inte är en kurs i mjukvaruarkitetkur kommer vi inte behandla de regler som definieras av REST. Fokus ligger istället på hur vi med hjälp av Python-script kommunicerar med enkla REST-liknande API:er.

13.1.1 Vad är ett API?

Det kanske lättaste sättet att förstå vad ett API är för något är nog genom att direkt ansluta mot ett genom sin webbläsare. Eftersom API:er levereras över HTTP så är detta möjligt. Nedan följer ett API vars uppgift är att beskriva egenskaperna av nummer du efterfrågar (testa byta ut 1337 till ett annat heltal):

```
https://4bbh01oqwj.execute-api.eu-north-1.amazonaws.com
/numcheck?integer=1337
```

Som svar till din API-förfrågan fick du en JSON-formaterad text. Denna kan du i Python med enkelhet omvandla till ett dictionary. Av dictionary-objektet kan

du utläsa om det efterfrågade talet är ett jämt tal samt om det är ett primtal. Du fick också en lista över talets faktorer.

Facebook använder ett API för att hämta och publicera inlägg. Messenger använder ett API för att hämta och skicka meddelanden mellan användare. Netflix använder ett API för att hämta en lista över tillgängliga filmer. SMHI erbjuder ett API som script och appar kan använda för att hämta väderprognoser för olika positioner i Sverige:

```
https://opendata-download-metfcst.smhi.se/api/category/pmp3g
/version/2/geotype/point/lon/16.158/lat/58.5812/data.json
```

Som du säkert märkte är inte SMHI:s API byggt för att tolkas direkt av människor. Ett lite mer pedagogiskt exempel skulle kunna se ut såhär:

```
{
    "ort": "Norrköping",
    "prognoser": [
        {
            "datum": "1 juni 2020",
            "prognos": "soligt"
        },
        {
            "datum": "2 juni 2020",
            "prognos": "molnigt"
        },
        {
            "datum": "3 juni 2020",
            "prognos": "regnigt"
        },
        {
            "datum": "4 juni 2020",
            "prognos": "soligt"
        },
        {
            "datum": "5 juni 2020",
            "prognos": "soligt"
        }
    ]
}
```

13.2 Anrop

Vad menas med att anropa API:er i Python? Som nämnts tidigare beror detta på vilken typ av API det rör sig om. I denna kurs fokuserar vi på HTTP-baserade API:er som returnerar JSON-formaterade texter (som vi i våra script sedan kan omvandla till dictionaries och listor). Så för oss är slutmålet med att anropa

API:er att hämta olika dictionary-objekt utifrån olika URL:er.

Det finns inbyggda funktioner i Python som låter oss göra HTTP-anrop[1]. Förutom detta finns även tredjepartsbibliotek som förenklar HTTP-anrop ytterligare. I denna kurs använder vi oss av tredjepartsbiblioteket **requests** för att utföra API-anrop över HTTP.

Nedan följer ett exempel på hur man med **requests** kan anropa ett API och med hjälp av funktionen **json.loads(*str*)** tolka dess svar:

```
import requests
import json
url = 'https://54qhf521ze.execute-api.eu-north-1.amazon
aws.com/weather/stockholm/'
r = requests.get(url)
# API:et svarar med en sträng (texten vi ser om vi
# öppnar resursen med en webbläsare). Vi kan referera
# till denna sträng genom attributen 'text' i r.
response_string = r.text
# Eftersom texten är JSON-formaterad så kan den
# omvandlas till ett dictionary med json.loads.
response_dictionary = json.loads(r.text)
# Nu innehåller response_dictionary objektet vi
# hämtade från API:et.
print(response_dictionary)
```

Installationen av **requests** skiljer sig åt mellan operativsystem och versioner. Därför ger jag inga instruktioner om hur tillägget installeras. Prata istället direkt med er lärare eller labbassistent. Testa skriva av exemplet ovan och se om API-anropet fungerar för er. URL:en som anropas är:

```
https://54qhf521ze.execute-api.eu-north-1.amazonaws.com/
weather/stockholm
```

Besöker ni adressen med en webbläsare ser ni att API:et svarar med en väderprognos för den angivna staden de närmaste fem dagarna:

```
{"city": "stockholm", "forecasts": [{"date": "14␣Jan...
```

Tips! Planera inte era klädval enligt API:ens prognoser. Samtliga prognoser randomiseras vid varje anrop.

[1] https://docs.python.org/3/library/http.client.html

13.3 Övningsuppgifter

Vi rekommenderar att ni utför samtliga API-anrop med biblioteket **requests**.

Övningsuppgift 13.1. Till följande API ska man ange ett positivt heltal (i en query string parameter med namn **integer**). API:et returnerar ett objekt som beskriver heltalets egenskaper:

```
https://4bbh01oqwj.execute-api.eu-north-1.amazonaws.com/
numcheck?integer=100
```

Skapa ett program som ber användaren mata in ett positivt heltal. Om användaren matar in ett korrekt heltal ska scriptet kalla på API:et för att hämta information om talet och på ett snyggt formaterat sätt skriva ut responsen i konsolen.

```
Ange ett positivt heltal: 57750
57750 är ett jämt nummer. Numret är inte ett primtal.
Numrets faktorer är 2, 3, 5, 5, 5, 7 och 11.
```

Övningsuppgift 13.2. Till följande API anger man en av följande städer:

- Stockholm
- Göteborg
- Malmö
- Uppsala
- Västerås

API:et returnerar ett objekt som representerar väderprogrnoser för den angivna staden de fem följande dagarna:

```
https://54qhf521ze.execute-api.eu-north-1.amazonaws.com/
weather/stockholm
```

Skapa ett program där användaren matar in en stad. Programmet ska hämta väderprognoser för den angivna staden och på ett snyggt sätt presentera dessa för användaren.

```
Enter the name of the city
for which you want forecasts:
> Stockholm
----------------------
      FORECASTS
**********************
14 January      Rainy
15 January      Sunny
16 January      Clear
17 January      Sunny
18 January      Storm
----------------------
```

Övningsuppgift 13.3. Via följande API kan man hämta information om berömda musiker och band. Genom denna URL hämtar man en lista på artister som finns i systemet:

```
https://5hyqtreww2.execute-api.eu-north-1.amazonaws.com/
artists/
```

Varje artist representeras av ett objekt innehållande artistens namn och en unik identifiering:

```
{
    "name": "Avicii",
    "id": "cd822d767d6816e8a720234bc26043a39baf6d4cd"
}
```

För att hämta ytterligare information om den enskilda artisten lägger man på identifieringen till den ursprungliga URL:en:

```
https://5hyqtreww2.execute-api.eu-north-1.amazonaws.com/
artists/cd822d767d6816e8a720234bc26043a39baf6d4cd
```

Identifierings-strängarna byts ut varje minut. Av denna anledningen ger länken ovan ett felmeddelande. Testa bygga din egen förfrågan istället.

Skapa ett program som genom kommunikation med API:et skriver ut en lista över tillgängliga artister. Programmet ska sedan be användaren mata in en artist. Fördjupande information ska hämtas och skrivas ut för den valda artisten.

Var noga med att inte hårdkoda artisternas identifieringar; dessa ogiltigförklaras efter 1-2 minuter.

```
--- ARTIST DB ---
Ariana Grande
Avicii
Blink-182
Brad Paisley
Ed Sheeran
Imagine Dragons
Maroon 5
Scorpions
-----------------
Select artist:
> avicii
-----------------
Avicii
**********
Genres: progressive house, electro house
Years active: 2006-2018
Members: Tim Bergling
-----------------
```

Kapitel 14

Funktioner

Funktioner gör det enkelt att återanvända och simplifiera repetetativ kod. Även om du ännu inte definierat din första funktion så har du högst sannolikt använt en mängd redan befintliga funktioner:

```
# Funktionen print() skriver ut information på skärmen.
print("Hello world!")

# Funktionen max() returnerar det högsta värdet från
# ett itererbart objekt (exempelvis en lista).
MyNumbers = [5, 9, 2]
large = max(MyNumbers)
```

Fördelen med funktioner är att endast funktionens skapare behöver veta och förstå hur den implementerats och programmerats. Övriga användare behöver endast förstå hur man använder funktionen. Vi behöver exempelvis inte veta exakt hur **max()** hittar det största talet i en lista. Det vi måste komma ihåg vid användningen av funktionen är dock att alltid ange listan som argument vid anropet:

```
large = max(MyNumbers)
```

Kör vi **max()** utan argument kommer programmet krascha. Funktionen är inte byggd för att köras så:

```
large = max()
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# TypeError: max expected 1 arguments, got 0
```

Vi behöver inte begränsa oss till att endast använda fördefinierade funktioner i Python. Vi kan även definiera våra egna, vilket är vad det här kapitlet handlar om.

14.1 Definition

Innan du definierar en funktion behöver du bestämma dig för:

- Funktionens namn
- Hur många argument funktionen behöver för att köra
- Argumentens namn

Låt säga att jag vill skapa en funktion **minimum()** som returnerar det minsta talet från en lista. För denna funktion behöver jag ett argument som innehåller listan vilken funktionen ska analysera. Låt oss kalla detta argument för **collection**:

```
def minimum(collection):
    # Funktionens implementation skrivs här.
```

En funktion behöver en implementation. Implementationen gör så att funktionen fungerar som den ska. En implementation till **minimum()** skulle kunna se ut såhär:

```
def minimum(collection):
    smallest = collection[0]
    for number in collection:
        if number < smallest:
            smallest = number
    print("The␣smallest␣number␣is", smallest)
```

14.2 Anrop

När en funktion definierats och implementerats kan man vid behov anropa den. När man anropar funktioner är det viktigt att antalet argument och att argumentens datatyp stämmer överens med vad funktionen förväntar sig. Funktionen **minimum()** vi definierade tidigare förväntar sig endast ett argument och den ska vara av datatypen **list**:

```
def minimum(collection):
    smallest = collection[0]
    for number in collection:
        if number < smallest:
            smallest = number
    print("The␣smallest␣number␣is", smallest)

numbers = [5, 2, 7, 4, 9]
minimum(numbers)
# OUTPUT: The smallest number is 2
minimum([6,5,100,7,4])
# OUTPUT: The smallest number is 4
```

14.3 Returnering av värden

Med funktionen **max()** kan vi tilldela variabler det högsta värdet från en lista:

```
numbers = [5, 2, 7, 4, 9]
large = max(numbers)
print(large)
# OUTPUT: 9
```

Den nuvarande implementationen av **minimum()** skriver endast ut det minsta numret på skärmen med hjälp av **print()** funktionen. Vi kan inte använda den för att tilldela variabler det lägsta värdet:

```
def minimum(collection):
    smallest = collection[0]
    for number in collection:
        if number < smallest:
            smallest = number
    print("The␣smallest␣number␣is", smallest)

numbers = [5, 2, 7, 4, 9]
small = minimum(numbers)
# OUTPUT: The smallest number is 2
print(small)
# OUTPUT: None
```

För att funktioner ska kunna användas för att tilldela värden till variabler behöver funktionen returnera ett slutgiltigt värde med hjälp av nyckelordet **return** likt exemplet nedan:

```
def double(number):
    return number * 2

my_num = double(2)
print(my_num)
```

Det räcker alltså inte med att **minimum()** skriver ut det lägsta talet med **print()**. Istället vill vi att funktioner returnerar det lägsta talet:

```
def minimum(collection):
    smallest = collection[0]
    for number in collection:
        if number < smallest:
            smallest = number
    return smallest

numbers = [5, 2, 7, 4, 9]
small = minimum(numbers)
print(small) # OUTPUT: 2
```

14.4 Argument

Vid definitionen av funktioner framgår det hur många argument funktionen godtar:

```
# Godtar två argument: first_name och last_name
def greet (first_name, last_name):
    print("Hello", first_name, last_name)

greet("Lisa", "Svensson")
greet("Gunnar", "Karlsson")
```

Anropar man funktionen med för få argument kraschar programmet:

```
greet("Gunnar")
# TypeError: greet() missing 1 required positional
# argument: 'last_name'
```

I Python kan man ange standardvärden för argument när man definierar funktioner. Detta gör argumentet valdbar vid anrop av funktionen. Om man inte anger argumentet vid anropet antar det standardvärdet:

```
def greet (first_name, last_name="Svensson"):
    print("Hello", first_name, last_name)

greet("Lisa", "Andersson")
# Skriver ut: Hello Lisa Andersson
greet("Gunnar")
# Skriver ut: Hello Gunnar Svensson
```

14.5 Moduler

Arbetar man med större projekt kan det vara bra att dela upp källkoden till program i flera filer. I Python är det möjligt att separera grundprogrammet från funktionerna med olika filer. Funktionerna kan vi sedan importera till huvudprogrammet med nyckelordet **import**. Filer som innehåller funktioner och vars syfte är att importeras till andra filer kallas i Python-termer för moduler. Låt säga att vi vill separera funktionerna från grundkoden i följande program:

```
def hello (name):
    print("Good␣day", name)

def goodbye (name):
    print("Farewell", name)

hello("Lina")
goodbye("Lina")
```

Funktionerna vill vi istället lagra i en fil med namn **communication.py**. Skapa filen och flytta funktionerna dit:

```
def hello (name):
    print("Good␣day", name)

def goodbye (name):
    print("Farewell", name)
```

Istället för att definiera så ska vi ska vi importera funktionerna till **main.py**. Så länge **communication.py** lagras i samma mapp som **main.py** så kan samtliga funktioner importeras med kommandot **import communication** (utan filändelsen .py). Vi måste referera till **communication** varje gång vi vill anropa funktionerna:

```
import communication

communication.hello("Lina")
communication.goodbye("Lina")
```

Är man extra tydlig med vilka funktioner man vill importera från **communication.py** behöver man inte referera till modulen vid anrop:

```
from communication import hello, goodbye

hello("Lina")
goodbye("Lina")
```

14.6 Övningsuppgifter

Övningsuppgift 14.1. I amerikanska cykeltävlingar brukar distansen för loppen anges i längdenheten mile (Engelsk mil). Vid europeiska tävlingar används istället längdenheten kilometer. Att manuellt omvandla mellan längdenheterna via miniräknare är mödosamt. Därför ska du skapa ett program som förenklar vid just denna typ av omvandling.

Programmet ska be användaren mata in en distans. Programmet ska automatiskt känna igen enheten för distansen användaren matade in och utföra en omvandling. Matar användaren exempelvis **15 km** så görs följande omvandling:

```
Ange distans > 15 km
15 km motsvarar 9.32056788356001 miles.
```

Matar användaren exempelvis in **500 miles** så gör programmet följande omvandling:

```
Ange distans > 500 miles
500 miles motsvarar 804.672 km.
```

Omvandlingarna ska utföras direkt av två funktioner:

```
def km_to_miles(dist):
    # Din kod här...

def miles_to_km(dist):
    # Din kod här...
```

Övningsuppgift 14.2. Du ska skapa en modul (med namn **ui**) som innehåller en samling funktioner som förenklar vid utveckling av gränssnitt för terminalapplikationer:

```
------------------------------
|          EXEMPEL           |
******************************
| Detta är ett exempel på hur
| ett gränssnitt kan se ut.
------------------------------
|    ..vad vill du göra?     |
------------------------------
| A | Visa inköpslista
| B | Lägg till vara
| C | Ta bort vara
| X | Stäng programmet
------------------------------
| Val >
```

Exemplet ovan genererades av följande programkod:

```
import ui

ui.line()
ui.header('EXEMPEL')
ui.line(True)
ui.echo('Detta är ett exempel på hur')
ui.echo('ett gränsnitt kan se ut.')
ui.line()
ui.header('..vad vill du göra?')
ui.line()
ui.echo('A | Visa inköpslista')
ui.echo('B | Lägg till vara')
ui.echo('C | Ta bort vara')
ui.echo('X | Stäng programmet')
ui.line()
ui.prompt('Val')
```

Din uppgift är att implementera funktionerna för **ui**. Visa sedan att koden ovan ger samma resultat i er implementation som i exemplet. Funktionerna som behöver implementeras är:

line(dots=False) Skriver ut en serie bindestreck på skärmen. Om det valbara argumentet dots sätts till True skrivs istället asterisker (*) ut:

```
line()
# OUTPUT: ------------------------------
line(True)
# OUTPUT: ******************************
```

header(text) Skriver ut en rubrik på skärmen. Rubriken ska centreras inom fönsterbredden:

```
ui.header('EXEMPEL')
# OUTPUT: |           EXEMPEL            |
```

echo(text) Skriver ut en text på skärmen. Ram visas endast på vänster sida:

```
ui.echo('Detta är ett exempel på hur')
# OUTPUT: | Detta är ett exempel på hur
```

prompt(text) Hämtar input från användaren. Skriver ut en vänster-ram och strängen anroparen satt för argumentet text:

```
ui.prompt('Val')
# OUTPUT: | Val > *INPUT*
```

clear() Rensar skärmen.

Övningsuppgift 14.3. Utgå från följande kod:

```
import random

numbers = []

for x in range(10):
    numbers.append(random.randint(0,9))
```

Variabeln **numbers** innehåller en lista som består av 10 slumpade heltal.

Din uppgift är att konstruera en funktion **get_even(*list*)** som returnerar samtliga jämna heltal från *list*. Listan i *list* ska inte ändras av funktionen.

Hämta och skriv ut alla jämna heltal från **numbers** med din nya funktion. Skriv därefter ut **numbers** för att bekräfta att listan inte ändrats av funktionen:

```
even = get_even(numbers)
print('even:', even)
print('numbers:', numbers)
```

Övningsuppgift 14.4. I variabeln **teams** finns ett dictionary som beskriver Grupp E under världsmästerskapet i fotboll 2018:

```
teams = {
    'Brazil': {
        'wins': 0,
        'draws': 0,
        'losses': 0,
        'goals_for': 0,
        'goals_against': 0
    },
    'Serbia': {
        'wins': 0,
        'draws': 0,
        'losses': 0,
        'goals_for': 0,
        'goals_against': 0
    },
    'Switzerland': {
        'wins': 0,
        'draws': 0,
        'losses': 0,
        'goals_for': 0,
        'goals_against': 0
    },
    'Costa_Rica': {
        'wins': 0,
        'draws': 0,
        'losses': 0,
        'goals_for': 0,
        'goals_against': 0
    }
}
```

Din uppgift är att konstruera en funktion **add_game** som fyller dictionaryt med matchinformation för en match. Funktionen ska ta fyra argument:

home_team *string* Den nation som spelade hemmalag

home_score *integer* Antalet mål som gjordes av hemmalaget

away_team *string* Den nation som spelade bortalag

away_score *integer* Antalet mål som gjordes av bortalaget

Nyckeln *'wins'* ska ökas med ett för det lag som gjort flest mål under en match.

Nyckeln *'losses'* ska ökas med ett för laget som gjort minst mål.

Har båda lag gjord lika många mål ska istället nyckeln *'draws'* ökas med ett för båda lagen.

Nyckeln *'goals_for'* används för att hålla reda på hur många mål ett lag gjort under turneringen.

Nyckeln *'goals_against'* används för att hålla reda på hur många mål ett lag släppt in under en turnering.

När du implementerat **add_game** ska du fylla **teams** med matchinformation genom att köra följande:

```
# 17 June 2018 #
add_game('Costa␣Rica', 0, 'Serbia', 1)
add_game('Brazil', 1, 'Switzerland', 1)
# 22 June 2018 #
add_game('Brazil', 2, 'Costa␣Rica', 0)
add_game('Serbia', 1, 'Switzerland', 2)
# 27 June 2018 #
add_game('Serbia', 0, 'Brazil', 2)
add_game('Switzerland', 2, 'Costa␣Rica', 2)
```

Bekräfta att din funktion fungerar som den ska genom att, efter att du fyllt variabeln med matchdata, jämföra **teams** mot tabellen nedan:

```
-------------------------------------
| Nation      | W | D | L | GF | GA |
-------------------------------------
| Brazil      | 2 | 1 | 0 |  5 |  1 |
| Serbia      | 1 | 0 | 2 |  2 |  4 |
| Switzerland | 1 | 2 | 0 |  5 |  4 |
| Costa Rica  | 0 | 1 | 2 |  2 |  5 |
-------------------------------------
```

Övningsuppgift 14.5. *Fortsättning på Övningsuppgift 14.4.*

När vi läste in matchdata i föregående övningsuppgift lagrade vi samtliga nationer som element i ett dictionary:

```
teams = {
    'Brazil': {
        'wins': 2,
        'draws': 1,
        'losses': 0,
        'goals_for': 5,
        'goals_against': 1
    },
    'Serbia': {
        'wins': 1,
        'draws': 0,
        'losses': 2,
        'goals_for': 2,
        'goals_against': 4
    }
}
```

Att hantera samtliga nationer i ett dictionary förenklar vid inläsningsförfarandet eftersom vi enkelt kan referera till respektive nation genom att ange dess namn som nyckel:

```
teams['Brazil']['wins'] += 1
```

I nästa kapitel läser du om hur man sorterar listor i Python. Sortering är något vi behöver implementera om vi vill skriva ut snygga tabeller där lagen sorterats i poäng-ordning.

Ska vi i framtiden sortera matchresultaten är det bättre om vi lagrar samtliga lag som element i en lista istället för ett dictionary:

```
[
    {
        'country': 'Brazil',
        'wins': 2,
        'draws': 1,
        'losses': 0,
        'goals_for': 5,
        'goals_against': 1
    },
    {
        'country': 'Serbia',
        'wins': 1,
        'draws': 0,
        'losses': 2,
        'goals_for': 2,
        'goals_against': 4
    }
]
```

Konstruera en funktion **make_list(*dict*)** som omvandlar matchresultat från dictionary (likt formatet i variabeln **teams**) till en lista (likt exemplet ovan).

Övningsuppgift 14.6. *Fortsättning på Övningsuppgift 14.5.*

Med funktionerna du skapat i tidigare uppgifter har nu har vi nu samlat in och lagrat matchdata i följande format:

```
teams = [
    {
        'country': 'Brazil',
        'wins': 2,
        'draws': 1,
        'losses': 0,
        'goals_for': 5,
        'goals_against': 1
    },
    {
        'country': 'Serbia',
        'wins': 1,
        'draws': 0,
        'losses': 2,
        'goals_for': 2,
        'goals_against': 4
    },
    {
        'country': 'Switzerland',
        'wins': 1,
        'draws': 2,
        'losses': 0,
        'goals_for': 5,
        'goals_against': 4
    },
    {
        'country': 'Costa_Rica',
        'wins': 0,
        'draws': 1,
        'losses': 2,
        'goals_for': 2,
        'goals_against': 5
    }
]
```

Konstruera en funktion **print_table(*list*)** till vilken man anger en lista (likt exemplet ovan) som argument. Genom att iterera listan ska funktionen skriva ut en numrerad tabell som ser ut enligt följande:

```
-------------------------------------------------
| #  | Nation      | W | D | L | GF | GA | GD | P |
-------------------------------------------------
| 1  | Brazil      | 2 | 1 | 0 |  5 |  1 |  4 | 7 |
| 2  | Serbia      | 1 | 0 | 2 |  2 |  4 | -2 | 3 |
| 3  | Switzerland | 1 | 2 | 0 |  5 |  4 |  1 | 5 |
| 4  | Costa Rica  | 0 | 1 | 2 |  2 |  5 | -3 | 1 |
-------------------------------------------------
```

Notera att det i exemplet ovan förekommer två extra kolumner: GD (målskillnad) och P (poäng). Målskillnaden räknas ut genom att subtrahera gjorda mål med insläppta mål. Poängen räknas ut genom att multiplicera antalet vinster med tre och addera antalet oavgjorda matcher. Gör dessa uträkningar i funktionen.

Kapitel 15

Sortering

I det här kapitlet introducerar vi sortering i Python. Kapitlet undersöker hur man sorterar listor med funktionen **sorted(*list*)**.

Hur en lista sorteras beror på vilka datatyper listan innehåller. Listor med blandade datatyper kan vanligtvis inte sorteras. I kapitlet tittar vi specifikt på hur man sorterar följande:

- Listor av nummer
- Listor av strängar
- Listor av dictionaries

Funktionen **sorted** döljer den underliggande sorteringsalgoritmen. Sorteringsalgoritmer är inte något som kapitlet behandlar. Detta är istället något ni läser om i fortsättningskursen *Datastrukturer, algoritmer och programkonstruktion med Python*.

15.1 Funktionen sorted

Den inbyggda funktionen **sorted** kan användas för att sortera listor (eller andra itererbara objekt). Funktionen tar in en lista som argument och returnerar en ny sorterad variant av listan:

```
l = [3, 7, 1, 3]
s = sorted(l)
print(s)
# OUTPUT: [1, 3, 3, 7]
```

Vanligtvis är det inte möjligt att sortera listor med blandande datatyper. I exemplet nedan används funktionen **sorted** på en lista som innehåller både

strängar och heltal. Funktionen kraschar eftersom det inte är möjligt att jämföra strängar och heltal med jämförelseoperatorn <:

```
l = [3, 7, 'hej', 1, 3, 'cykel']
s = sorted(l)
# Traceback (most recent call last):
#   File "main.py", line 2, in <module>
#     s = sorted(l)
# TypeError: '<' not supported between instances of
#            'str' and 'int'
```

I följande stycken tittar vi specifikt till hur **sorted** fungerar med olika datatyper.

15.1.1 Listor av nummer

Regeln om att inte blanda datatyper i listor som ska sorteras gäller inte för heltal och flyttal eftersom dessa kan jämföras med jämförelseoperatorn <:

```
if 4 < 5.5:
    print('heltalet␣4␣än␣mindre␣än␣flyttalet␣5.5')
```

Vi kan därför sortera listor med båda datatyperna:

```
l = [2.72, 1, 3.14, 5, 1.41, 3]
s = sorted(l)
print(s)
# OUTPUT: [1, 1.41, 2.72, 3, 3.14, 5]
```

Vill vi sortera i omvänd ordning lägger vi till argumentet **reverse** till funktionen och sätter argumentet till det booleska värdet *True*:

```
l = [2.72, 1, 3.14, 5, 1.41, 3]
s = sorted(l, reverse=True)
print(s)
# OUTPUT: [5, 3.14, 3, 2.72, 1.41, 1]
```

15.1.2 Listor av strängar

Visste du att jämförelseoperatorn < även fungerar för strängar?

```
if 'abba' < 'beef':
    print('Sant!')
```

Att stängar stödjer jämförelseoperatorn är en förutsättning för att strängar ska kunna sorteras med **sorted** men vad är det egentligen som jämförs?

> *Uttrycket* ***a*** $<$ ***b*** *är sant om* ***a*** *i alfabetisk ordning kommer före* ***b*** *givet att* ***a*** *och* ***b*** *är strängar.*

Funktionen **sorted** kommer därför sortera listor med strängar i alfabetisk ordning:

```
l = ['c', 'beef', 'abba', 'b', 'a', 'cafe']
s = sorted(l)
print(s)
# OUTPUT: ['a', 'abba', 'b', 'beef', 'c', 'cafe']
```

Sätter du argumentet **reverse** till *True* sorteras listan istället i omvänd alfabetisk ordning.

```
l = ['c', 'beef', 'abba', 'b', 'a', 'cafe']
s = sorted(l, reverse=True)
print(s)
# OUTPUT: ['cafe', 'c', 'beef', 'b', 'abba', 'a']
```

15.1.3 Listor av dictionaries

Följande lista innehåller dictionaries som representerar resultatet från Grupp B under världsmästerskapet i fotboll 1994:

```
group_b = [
    {
        'country': 'Russia',
        'points': 3
    },
    {
        'country': 'Brazil',
        'points': 7
    },
    {
        'country': 'Cameroon',
        'points': 1
    },
    {
        'country': 'Sweden',
        'points': 5
    }
]
```

Vi vill nu sortera listan i poäng-ordning (nyckeln *'points'*). Laget med högsta poäng ska placeras först i listan, laget med näst högst poäng placeras näst först, och så vidare. Avslutningsvis ska vi printa ut en snygg tabell.

Dictionaries stödjer inte jämförelseoperatorn <. Det är därför inte möjligt att utan anpassning sortera listor bestående av dictionaries med **sorted**.

```
standings = sorted(group_b)
# Traceback (most recent call last):
#   File "main.py", line 20, in <module>
#     standings = sorted(group_b)
# TypeError: '<' not supported between instances of 'dict' and 'dict'
```

Funktionen **sorted** kommer inte självmant börja jämföra objektens element och sortera utifrån dessa. Det är ju inte heller möjligt för funktionen att veta exakt vad du har som avsikt att sortera utifrån. Vill du sortera länderna i bokstavsordning (nyckeln *'country'*) eller poäng-ordning (nyckeln *'points'*)?

Argumentet key

För att sortera elementen utifrån nyckeln *'points'* behöver vi först konstruera en funktion som för elementen i listan **group_b** returnerar elementets poäng.

```
def get_points(element):
    return element['points']
```

Funktionen **get_points** kan nu användas för att hämta poängen för varje lag i listan **group_b** enligt följande:

```
print('Brazil␣has', get_points(group_b[1]), 'points')
# OUTPUT: Brazil has 7 points
```

Vi kan också få **sorted** att använda funktionen för att sortera **group_b** i poängordning genom att referera till funktionen i argumentet **key**:

```
standings = sorted(group_b, key=get_points)
```

Om man inte sätter argumentet **key** kommer funktionen **sorted** jämföra (och därav sortera) samtliga element likt följande minimala exempel:

```
russia = group_b[0]
brazil = group_b[1]
if russia < brazil: # OTILLÅTET: dict < dict
    print('Ryssland␣har␣lägre␣poäng␣än␣Brasilien')
```

När vi sätter argumentet **key** till **get_points** kommer **sorted** istället använda funktionen för elementen under jämförelsen likt följande minimala exempel:

```
russia = group_b[0]
brazil = group_b[1]
if get_points(russia) < get_points(brazil):
    print('Ryssland␣har␣lägre␣poäng␣än␣Brasilien')
```

Eftersom **get_points** returnerar heltal är jämförelsen tillåten och **sorted** kan därmed sortera samtliga element enligt heltalen funktionen returnerar.

Argumentet reverse

Vi har hittills lyckats sortera lagen i poäng-ordning:

```
standings = sorted(group_b, key=get_points)
```

Funktionen **sorted** sorterar heltal i stigande ordning. Vill vi iterera listan för att skriva ut en tabell behöver laget med högst poäng placeras först i listan. Vi sorterar i fallande ordning genom att sätta argumentet **reverse** till *True*:

```
standings = sorted(group_b, key=get_points, reverse=True)
```

Utskrift

Nu innehåller variabeln **standings** en lista över samtliga lag som sorterats enligt lagens poäng i fallande ordning. Genom att iterera listan kan vi skriva ut den slutgiltiga tabellen:

```
print('Country␣␣␣␣P')
print('------------')
for team in standings:
    print(team['country'].ljust(10), team['points'])
```

Utskriften blir följande:

```
Country     P
------------
Brazil      7
Sweden      5
Russia      3
Cameroon    1
```

Resultat

Nedan följer det fullständiga scriptet:

```
group_b = [
    {
        'country': 'Russia',
        'points': 3
    },
    {
        'country': 'Brazil',
        'points': 7
    },
    {
        'country': 'Cameroon',
        'points': 1
    },
    {
        'country': 'Sweden',
        'points': 5
    }
]

def get_points(element):
    return element['points']

standings = sorted(group_b, key=get_points, reverse=True)

print('Country␣␣␣␣P')
print('------------')
for team in standings:
    print(team['country'].ljust(10), team['points'])
```

15.2 Övningsuppgifter

Övningsuppgift 15.1. Variabeln **students** innehåller en lista över elever som ska delta vid en skolutflykt:

```
persons = [
    'Alice', 'Lucas', 'Olivia',
    'Liam', 'Astrid', 'William'
]
```

Din uppgift är att formatera listan så den passar för utskrifter. Detta ska du göra genom att först sortera listan i bokstavsordning. Skriv därefter ut varje elev på en ny rad med tillhörande kryssruta så att läraren kan rapportera närvaro för eleverna:

```
[ ] Alice
[ ] Astrid
[ ] Liam
[ ] Lucas
[ ] Olivia
[ ] William
```

Övningsuppgift 15.2. I scriptet nedan fylls listan i **numbers** med tio slumpmässiga heltal, inom intervallet 0 till och med 20:

```
import random
numbers = []
for x in range(10):
    numbers.append(random.randint(0,20))
```

Din uppgift är att sortera **numbers** från lägst till högst. Gör en snygg utskrift av listan innan och efter sorteringen likt exemplet nedan:

```
BEFORE SORTING:
- 9
- 2
- 20
- 13
- 8
- 8
- 15
- 12
- 17
- 6
AFTER SORTING:
- 2
- 6
- 8
- 8
- 9
- 12
- 13
- 15
- 17
- 20
```

Övningsuppgift 15.3. *Fortsättning på Övningsuppgift 14.6.*

I övningsuppgifterna för förra kapitlet konstruerade du en samling funktioner för att läsa, tolka och presentera fotbollsresultat:

add_game Läser in matchresultat. Lagrar resultat i dictionary.

make_list Omvandlar dictionary med matchresultat till lista.

print_table Skriver ut tabell genom att tolka lista med matchresultat.

Mellan funktionerna **make_list** och **print_table** valde vi att medvetet hoppa över en funktion som sorterar resultatet innan utskrift. Men den ska du implementera nu när du läst om hur man sorterar listor med dictionaries.

Konstruera en funktion **sort_list(*list*)** som kan sortera listor med matchresultat. Alltså samma typ av listor som returneras av funktionen **make_list**. Listornas element ska sorteras i poäng-ordning. Funktionen ska returnera den sorterade listan.

Efter att du implementerat funktionen bör du kunna sammanställa en helheltslösning enligt följande grund:

```
teams = # HÄMTA FRÅN ÖVNINGSUPPGIFT 14.4.

def add_game(home_team, home_score, away_team, away_score):
    # DIN LÖSNING HÄR

def make_list(result_dict):
    # DIN LÖSNING HÄR

def sort_list(result_list):
    # DIN LÖSNING HÄR

def print_table(result_list):
    # DIN LÖSNING HÄR

add_game('Costa␣Rica', 0, 'Serbia', 1)
add_game('Brazil', 1, 'Switzerland', 1)
add_game('Brazil', 2, 'Costa␣Rica', 0)
add_game('Serbia', 1, 'Switzerland', 2)
add_game('Serbia', 0, 'Brazil', 2)
add_game('Switzerland', 2, 'Costa␣Rica', 2)

teams_list = make_list(teams)
teams_list_sorted = sort_list(teams_list)
print_table(teams_list_sorted)
```

För att spara rader valde jag att i exemplet ovan inte skriva ut hela definitionen av variabeln **teams**. Hämta istället dictionaryt från Övningsuppgift 14.4.

Notera att helheltsgrunden som presenteras i exemplet ovan endast är ett förslag. Har du exempelvis byggt ut funktionerna som moduler eller i överlag strukturerat din kod på ett annat sätt så är detta okej!

Men hur du än väljer att strukturera din kod är det viktiga att inläsningen av matchdata, sorteringen och utskriften görs på ett korrekt sätt:

```
-------------------------------------------------
| #  | Nation       | W | D | L | GF | GA | GD | P |
-------------------------------------------------
| 1  | Brazil       | 2 | 1 | 0 |  5 |  1 |  4 | 7 |
| 2  | Switzerland  | 1 | 2 | 0 |  5 |  4 |  1 | 5 |
| 3  | Serbia       | 1 | 0 | 2 |  2 |  4 | -2 | 3 |
| 4  | Costa Rica   | 0 | 1 | 2 |  2 |  5 | -3 | 1 |
-------------------------------------------------
```