

---

# LÄR DIG PYTHON FRÅN GRUNDEN

---

Linus Rundberg Streuli  
Antonio Prgomet

Första upplagan

# Innehållsförteckning

<b>Introduktion</b>	<b>7</b>
Bokens målgrupp . . . . .	7
Bokens upplägg . . . . .	7
Språk . . . . .	8
Bokens GitHub . . . . .	8
 <b>1 Introduktion till Python</b>	 <b>9</b>
1.1 Vad är Python? . . . . .	9
1.1.1 Kort historik . . . . .	9
1.1.2 Pythons popularitet . . . . .	10
1.1.3 Olika versioner av Python . . . . .	10
1.2 Pythoninstallation . . . . .	11
1.3 Terminalen . . . . .	12
1.4 Virtuella miljöer . . . . .	12
1.4.1 Skapande av en virtuell miljö . . . . .	12
1.5 Installation av tredjepartsbibliotek . . . . .	13
1.5.1 Installation via Anaconda . . . . .	13
1.5.2 Installation via pip . . . . .	13
1.6 Begrepp och koncept . . . . .	13
1.6.1 Att exekvera Python-kod . . . . .	13
1.6.2 Konventioner . . . . .	14
1.6.3 Objekt, datatyp, värde, identitet . . . . .	14
1.6.4 Namnrymd och omfång . . . . .	16
1.7 Översikt av kommande kapitel . . . . .	16

1.8	Sammanfattning . . . . .	17
1.9	Uppgifter . . . . .	18
<b>2</b>	<b>Datatyper</b>	<b>19</b>
2.1	Numeriska datatyper . . . . .	19
2.1.1	Beräkningar med numeriska datatyper . . . . .	20
2.2	Booleska datatyper . . . . .	22
2.3	Textsträngar . . . . .	23
2.3.1	Indexering av <code>str</code> . . . . .	27
2.3.2	Längden på en sträng - <code>len()</code> . . . . .	28
2.3.3	Dela upp en sträng - <code>split()</code> . . . . .	29
2.3.4	Radbyte . . . . .	29
2.3.5	Formatering av <code>str</code> . . . . .	30
2.4	Variabler . . . . .	31
2.4.1	Identifierare och identitet . . . . .	33
2.5	Datastrukturer . . . . .	35
2.5.1	<code>tuple</code> . . . . .	36
2.5.2	<code>list</code> . . . . .	38
2.5.3	<code>dict</code> . . . . .	46
2.5.4	<code>set</code> . . . . .	51
2.5.5	<code>range</code> . . . . .	57
2.6	Sammanfattning . . . . .	60
2.7	Uppgifter . . . . .	63
<b>3</b>	<b>Logik</b>	<b>69</b>
3.1	Jämförelser . . . . .	69
3.1.1	Tillbaka till <code>bool</code> . . . . .	70
3.1.2	Sanningsvärden för objekt . . . . .	72
3.1.3	Logiska operatorer . . . . .	73
3.2	Villkorlig logik . . . . .	74
3.2.1	<code>if</code> -satser med <code>bool</code> . . . . .	78
3.2.2	Skillnaden mellan flera <code>if</code> -satser och <code>if ... elif</code> . . . . .	79
3.3	<code>if ... else</code> som en <i>expression</i> . . . . .	81
3.4	<code>if</code> -satser med logiska operatorer . . . . .	82
3.5	Sammanfattning . . . . .	84
3.6	Uppgifter . . . . .	85
<b>4</b>	<b>Loopar</b>	<b>87</b>

4.1	<code>for</code> -loopar . . . . .	87
4.1.1	<code>break</code> och <code>continue</code> . . . . .	89
4.2	<code>while</code> -loopar . . . . .	90
4.3	<code>else</code> i loopar . . . . .	92
4.4	Sammanfattning . . . . .	94
4.5	Uppgifter . . . . .	95
<b>5</b>	<b>Funktioner</b>	<b>97</b>
5.1	Definiera och exekvera funktioner . . . . .	97
5.2	<code>return</code> . . . . .	98
5.3	Argument och parametrar . . . . .	100
5.3.1	Argument . . . . .	101
5.3.2	Parametrar . . . . .	104
5.4	Omfång ( <i>scope</i> ) . . . . .	109
5.5	Inbyggda funktioner . . . . .	110
5.6	Enkla funktioner - <code>lambda</code> . . . . .	117
5.7	Dokumentation med <code>docstring</code> . . . . .	119
5.8	Decorators . . . . .	120
5.8.1	Funktioner i funktioner . . . . .	120
5.8.2	Funktioner är objekt . . . . .	121
5.8.3	Funktioner som argument . . . . .	122
5.8.4	Definiera en <i>decorator</i> . . . . .	124
5.8.5	Dekorera med <code>@</code> . . . . .	126
5.9	Type hints . . . . .	127
5.9.1	Analysera kod med <code>mypy</code> . . . . .	129
5.9.2	Analysera kod direkt i en IDE . . . . .	131
5.10	Sammanfattning . . . . .	132
5.11	Uppgifter . . . . .	133
<b>6</b>	<b>Klasser</b>	<b>135</b>
6.1	Datatyp och klass . . . . .	136
6.2	Exempel 1: Biblioteksbok . . . . .	137
6.3	<code>self</code> . . . . .	142
6.4	Exempel 2: Student . . . . .	144
6.4.1	Klassattribut och instansattribut . . . . .	146
6.5	Klassmetoder och statiska metoder . . . . .	148
6.5.1	Klassmetoder . . . . .	148

6.5.2	Statiska metoder . . . . .	148
6.6	Omvandling av objekt . . . . .	149
6.7	<code>isinstance()</code> . . . . .	149
6.8	Underklasser . . . . .	150
6.8.1	Exempel: Olika typer av anställda . . . . .	150
6.9	Docstrings i klasser . . . . .	154
6.10	<code>__dunder__</code> -metoder . . . . .	155
6.10.1	Tydligare representation av klasser med <code>__str__()</code> . . . . .	155
6.10.2	Exempel: En överdrivande <code>list</code> . . . . .	157
6.11	Objektorienterad programmering . . . . .	158
6.11.1	Inkapsling - <i>encapsulation</i> . . . . .	158
6.11.2	Arv - <i>inheritance</i> . . . . .	161
6.11.3	Polymorfism - <i>polymorphism</i> . . . . .	161
6.11.4	Abstraktion - <i>abstraction</i> . . . . .	163
6.12	Exempelkod för klasser . . . . .	164
6.13	Sammanfattning . . . . .	167
6.14	Uppgifter . . . . .	168
<b>7</b>	<b>Konventioner och standarder</b>	<b>173</b>
7.1	Vad är PEP för något? . . . . .	173
7.2	PEP 8 - Style Guide for Python Code . . . . .	174
7.3	PEP 20 - The Zen of Python . . . . .	175
7.4	Sammanfattning . . . . .	177
7.5	Uppgifter . . . . .	178
<b>8</b>	<b>Felhantering</b>	<b>179</b>
8.1	Syntaxfel . . . . .	179
8.2	Exceptions . . . . .	180
8.2.1	Att fånga <i>exceptions</i> . . . . .	182
8.2.2	Att lyfta <i>exceptions</i> . . . . .	192
8.2.3	Mer om <i>exceptions</i> . . . . .	193
8.3	Sammanfattning . . . . .	195
8.4	Uppgifter . . . . .	196
<b>9</b>	<b>Test</b>	<b>199</b>
9.1	<code>doctest</code> . . . . .	199
9.2	<code>pytest</code> . . . . .	202

9.2.1	Exempel . . . . .	205
9.3	Sammanfattning . . . . .	207
9.4	Uppgifter . . . . .	208
<b>10</b>	<b>Moduler och paket</b>	<b>209</b>
10.1	Moduler . . . . .	209
10.1.1	Huvudmodulen - <code>__main__</code> . . . . .	210
10.1.2	Import av moduler . . . . .	211
10.1.3	Namnrymd . . . . .	213
10.1.4	Importera egna moduler . . . . .	219
10.2	Paket . . . . .	222
10.3	Sammanfattning . . . . .	224
10.4	Uppgifter . . . . .	225
<b>11</b>	<b>Loggning</b>	<b>227</b>
11.1	Nivåer för logg-meddelanden . . . . .	228
11.2	Formatering av logg-meddelanden . . . . .	229
11.2.1	Formatera tidsfältet . . . . .	230
11.3	Logg-meddelanden med variabler . . . . .	231
11.4	Logg-filer . . . . .	232
11.5	Goda logg-vanor . . . . .	234
11.6	Bryter logging mot kodstandarden? . . . . .	234
11.7	Ytterligare ett exempel . . . . .	235
11.8	Sammanfattning . . . . .	236
11.9	Uppgifter . . . . .	237
<b>12</b>	<b>Några vanligt förekommande tredjepartsbibliotek</b>	<b>239</b>
12.1	NumPy . . . . .	239
12.2	Pandas . . . . .	242
12.2.1	Pandas och SQL . . . . .	244
12.3	Matplotlib . . . . .	251
12.3.1	Explicit gränssnitt - Figure och Axes . . . . .	251
12.3.2	Implicit gränssnitt . . . . .	254
12.3.3	Andra typer av diagram . . . . .	256
12.4	Fler vanligt förekommande tredjepartsbibliotek . . . . .	258
12.5	Sammanfattning . . . . .	260
12.6	Uppgifter . . . . .	261

<b>13 Exempelprojekt</b>	<b>265</b>
13.1 Dataanalys . . . . .	265
13.1.1 EDA . . . . .	266
13.2 Streamlit . . . . .	270
13.3 Sammanfattning . . . . .	274
13.4 Uppgifter . . . . .	275

# Introduktion

## Bokens målgrupp

Boken är avsedd för kurser av olika slag och används bland annat inom yrkeshögskolan. Boken går också att använda inom andra läroinstitut, företag och organisationer, eller för självstudier.

## Bokens upplägg

Boken använder sig av några element som är värda att notera.

Ibland har vi lagt nyckelord i marginalen. Det är för att underlätta att hitta ett visst begrepp igen vid bläddrande i boken.

nyckelord

Vi använder oss också av två sorters inforutor, som ser ut som följer.

**i** Så här ser en allmän inforuta ut. I de här rutorna skriver vi korta fördjupande förklaringar till vissa begrepp och koncept.

**!** Det här är en viktig-ruta. De här rutorna använder vi för att sätta fokus på viktiga distinktioner eller detaljer som kan vara svåra att upptäcka men som har stor betydelse.



Ofta är bokens kodexempel annoterade. I högra kanten förekommer siffror som motsvarar en förklarande text under kodexemplet. Efter den förklarande texten följer oftast resultatet av själva koden. Vi ser ett exempel här nedanför.

```
print('An annotated code example.')
```

①

- ① Siffran (1) till höger i kodexemplet motsvaras av en förklarande text direkt under kodexemplet.

An annotated code example.

Efter den förklarande texten ser vi resultatet av koden, i det här fallet den utskrivna texten 'An annotated code example.'.

Varje kapitel har en sammanfattning av centrala begrepp och eventuella konventioner kopplade till kapitlets innehåll, och avslutas med uppgifter som vi rekommenderar läsaren att göra. Uppgifterna har som syfte att läsaren ska börja vänja sig vid att skriva kod. Därför är uppgifterna i regel korta och baserade på innehållet från respektive kapitel.

## Språk

Den här boken är skriven på svenska. Bokens kodexempel är däremot på engelska eftersom det är en stark praxis världen över som vi vill uppmuntra läsare till att anamma redan från början för att skapa goda vanor.

## Bokens GitHub

Boken har en tillhörande GitHub-sida där material kopplat till boken finns uppladdat.

[https://github.com/AntonioPrgomet/laer\\_dig\\_python\\_fraan\\_grunden\\_1upplagan](https://github.com/AntonioPrgomet/laer_dig_python_fraan_grunden_1upplagan)

# Kapitel 1

## Introduktion till Python

I det här inledande kapitlet går vi igenom en kort historik om Python, nämner hur populärt språket är bland programmerare, och hur vi kan förstå Pythons system för att numrera olika versioner.

Vi går också igenom vad en läsare behöver installera på sin dator för att få ut så mycket som möjligt av boken.

Avslutningsvis går vi i det här kapitlet kort igenom några centrala begrepp och koncept inom Pythonprogrammering.

### 1.1 Vad är Python?

Python är ett fritt tillgängligt programmeringspråk med tillämpning inom flertalet olika områden, till exempel AI, mjukvaruutveckling, webbutveckling, data science, automation och spelutveckling.

#### 1.1.1 Kort historik

Den första versionen av Python lanserades 1991 och utvecklades av nederländaren Guido van Rossum. Sedan dess har han under lång tid varit ledande i utvecklingen av Python.

Namnet Python kommer från den brittiska humorgruppen Monty Python. Guido van Rossum ville att hans programmeringsspråk skulle ha ett kort, unikt och lite mystiskt namn.

Det är vanligt att man ser referenser till Monty Pythons tv-program och filmer i Pythons officiella dokumentation. I arbetet med den här boken har vi inte velat förutsätta att läsaren är bekant med Monty Python, och har därför valt att inte följa den traditionen.

### 1.1.2 Pythons popularitet

Det är svårt att objektivt jämföra popularitet hos olika programmeringsspråk. Det finns ett antal olika *index* som använder olika metoder, däribland Tiobe-indexet och *PopularitY of Programming Languages*, och enligt dem är Python ett av de mest populära programmeringsspråken i världen, och har varit så de senaste åren. När boken skrevs innehade Python förstaplatsen på båda listorna.

### 1.1.3 Olika versioner av Python

Python är under ständig utveckling och uppdateras kontinuerligt. En del funktioner från senare versioner är inte tillgängliga i tidigare versioner.

Python använder följande system för att numrera de olika versionerna:

Python version A.B.C där

- A endast ökar när riktigt stora förändringar i hur man använder Python sker,
- B ökar när ändringar sker som inte är fullt så stora, och
- C ökar när problem som upptäckts i en B-version fixats.

Senaste gången A ökades var år 2008 när Python 3.0 släpptes. Kod skriven för att fungera med Python 2 löpte stor risk att inte fungera med Python 3, och alla som arbetade med att skriva Python-kod uppmanades att uppdatera sin kod så att den skulle fungera med den nya versionen. 1 januari 2020 slutade Python 2 helt att uppdateras.

Senaste gången B ökades var i oktober 2023 när Python 3.12 släpptes. Nästa ökning av B är planerad till oktober 2024.

Den senaste Python-versionen när boken skrevs var 3.12.4, som släpptes i början av juni 2024. 4 innebär att det var fjärde gången mindre ändringar av Python 3.12 gjorts.

## 1.2 Pythoninstallation

För att tillägna sig den här boken fullt ut behöver läsaren en fungerande Pythoninstallation att programmera i.

När vi har installerat Python har vi tillgång till Python-konsolen, som är det mest grundläggande sättet att programmera i. Python-konsolen är helt textbaserad och består av ett fönster där vi kan skriva Pythonkod. Raden där vi skriver börjar med tre ”större än”-tecken (`>>>`). Därför ser man ofta kodexempel online som använder sig av det formatet. I praktiken använder programmerare mestadels en *integrated development enviroment* (IDE). Detta är ett mer flexibelt sätt att skriva kod på eftersom en IDE har mer funktionalitet såsom att kunna spara kod och *syntax highlighting* där kod färgmarkeras för att underlätta läsning av koden. Ett sätt att få tillgång till olika IDE:er är att installera Anaconda-distributionen. I den kan vi programmera i populära IDE:er såsom VS Code, Spyder eller PyCharm. Det går också att programmera i Jupyter Notebooks, som är interaktiva skript. Uppgifterna i den här boken är tillgängliga i Jupyter Notebooks på bokens GitHub, för den läsare som önskar det. För att se hur Anaconda installeras hänvisar vi läsaren till den officiella dokumentationen: <https://docs.anaconda.com/anaconda/install/>.

Den läsare som har programmerat tidigare och kanske har en annan föredragen Pythonmiljö istället för Anaconda-distributionen kan med gott samvete använda den.

## 1.3 Terminalen

I Avsnitt 5.9.1, Avsnitt 9.2 och Avsnitt 13.2 används terminalen för att exekvera Pythonkod. I Windows kallas terminalen “PowerShell”, och i Linux och macOS kallas den helt enkelt “Terminal”. I dessa avsnitt antar vi att läsaren har grundläggande kunskaper om hur terminalen används.

Den läsare som inte tidigare har arbetat i terminalen kan enkelt lära sig grunderna genom att exempelvis kolla på guider som finns tillgängliga på internet.

## 1.4 Virtuella miljöer

Oftast rekommenderas det att man inte installerar alla bibliotek man ska använda direkt i den Python-installation man har på datorn utan att man skapar en *virtuell miljö* (virtual environment) som är kopplad till just det projekt man arbetar med. En virtuell miljö är en kopia av en Python-installation som är isolerad från den centrala Python-installationen. Det innebär att miljön har en egen Python-tolkare, och att eventuella tredjepartsbibliotek som installeras, installeras i den miljön.

Fördelar med det tillvägagångssättet är bland annat att vi kan skapa en miljö som andra utvecklare kan återskapa och på så sätt vara säkra på att alla bibliotek som installeras har samma version. Det minskar risken för att koden vi skrivit bara fungerar lokalt på vår egen dator.

### 1.4.1 Skapande av en virtuell miljö

Virtuella miljöer kan skapas med hjälp av modulen `venv`, eller i Anaconda. Det finns gott om information online för det som vill veta hur det går till.

## 1.5 Installation av tredjepartsbibliotek

Viss funktionalitet finns inte i Pythons standardbibliotek utan tillhandahålls av tredjepartsbibliotek, som är Pythonkod skriven av andra personer och organisationer än de som utvecklar Python. Vi kommer att gå igenom ett antal vanligt förekommande tredjepartsbibliotek i Kapitel 12.

Tredjepartsbibliotek följer alltså inte med när vi installerar Python. De behöver installeras separat. Här nedan går vi i korthet igenom två sätt att installera tredjepartsbibliotek.

### 1.5.1 Installation via Anaconda

Tredjepartsbibliotek kan installeras via Anaconda Navigator. Läs Anacondas dokumentation för mer information.

### 1.5.2 Installation via pip

Pythons egna installationsverktyg heter `pip`. Det kan också användas för att installera tredjepartsbibliotek. Läsare som vill veta mer kan börja med att läsa `pip`:s dokumentation på <https://pip.pypa.io/en/stable/>.

## 1.6 Begrepp och koncept

Här går vi igenom en del koncept som är bra att ha stött på när vi går igenom dem senare i boken.

### 1.6.1 Att exekvera Python-kod

Python skiljer sig från en del andra programmeringsspråk som C och Java genom att koden inte behöver *kompileras* innan den exekveras (körs). Pythonkod *tolkas* istället av en Python-tolkare (*interpreter*) under tiden koden exekveras. Man säger därför att Python är *interpreted*, till skillnad från exempelvis Java och C, som är *compiled*.

## 1.6.2 Konventioner

Det finns många olika sätt att skriva Python-kod. När koden exekveras finns det många små detaljer som Python-tolkaren inte bryr sig om. Det spelar till exempel ingen roll för Python-tolkaren om vi deklarerar en variabel med koden `a=42` eller `a = 42`. (Notera blankstegen runt `=` i det andra exemplet.)

Även om det inte spelar någon roll för Python-tolkaren, kan det dock spela roll för en människa som ska läsa och förstå vad koden gör.

I dokumentet PEP 8 - Style Guide for Python Code som finns tillgängligt på <https://peps.python.org/pep-0008/>, samlas en mängd *konventioner* med syftet att Pythonkod som skrivs ska vara lättare för människor att läsa. Det handlar alltså inte om att skriva *fungerande* kod, utan *läsbar* kod.

I slutet av varje kapitel kommer vi, i de fall det är aktuellt, att lista konventioner som hör till kapitlets innehåll. Kapitel 7 handlar mer allmänt om konventioner.

## 1.6.3 Objekt, datatyp, värde, identitet

Följande avsnitt tar upp en del abstrakta koncept som inte är nödvändiga att förstå helt och hållet från början, men som är bra att ha stött på och för att börja skapa en intuition för hur Python hanterar data och kod.

### Objekt

I den officiella Python-dokumentationen kan vi läsa följande:

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. ... Every object has an identity, a type and a value.

Fritt översatt kan vi säga att all data i Python representeras av objekt, eller relationer mellan objekt. Alla objekt har en *identitet*, en *datatyp*, och ett *värde*.

**i** Här översätter vi engelskans mer allmänna *type* med det mer specifika *datatyp*.

## Identitet

När ett objekt skapas, får det en unik *identitet*, som representeras av ett heltal och är objektets adress i datorns minne. Den inbyggda funktionen `id()` returnerar ett objekts identitet.

Ett objekts identitet ändras inte efter att objektet skapats.

## Datatyp

Ett objekts datatyp avgör vilka operationer vi kan utföra på objektet, samt vilka värden som är möjliga för objektet att ha. Precis som objektets identitet, kan inte objektets datatyp ändras efter att objektet skapats.

Den inbyggda funktionen `type()` returnerar ett objekts datatyp.

Se Kapitel 2 för exempel på olika datatyper, vilka värden de kan ha, och vilka operationer vi kan utföra på dem.

**i** I Python är begreppen *typ*, *datatyp* och *klass* samma sak. I den här boken kommer vi använda ordet *datatyp* istället för *typ*, och klasser går vi närmare in på i Kapitel 6. Vi kan här snabbt nämna att klasser är sättet att skapa nya sorters datatyper i Python.

## Värde

Ett objekts värde är datan det representerar. Det kan vara ett numeriskt värde som talet 42, eller en textsträng som 'hello'.

Vissa objekts värde kan ändras efter att de skapats. Sådana objekt kallas *mutable*. Andra objekt kan vi inte ändra värdet på efter att de skapats. Dessa objekt kallas *immutable*.



**i** Det finns vissa subtila undantag till uppdelningen i *mutable* och *immutable* men de är utanför den här bokens omfattning.

### 1.6.4 Namnrymd och omfång

Begreppen namnrymd (*namespace*) och omfång (*scope*) beskriver hur och var olika objekt är tillgängliga.

En namnrymd är en samling namn, och de objekt som namnen representerar.

Omfång handlar om var i koden objekt är tillgängliga. Vissa objekt är bara tillgängliga i ett lokalt omfång (till exempel inuti en loop eller en funktion), medan andra finns tillgängliga i det globala omfånget. Vi kommer att se exempel på omfång i Avsnitt 5.4, och namnrymder i Avsnitt 10.1.3.

## 1.7 Översikt av kommande kapitel

Vi kommer i nästa kapitel att gå igenom Pythons olika datatyper. Därifrån kommer vi röra oss, via kapitel om logik och loopar, vidare till något mer komplexa koncept som funktioner och klasser. Sedan kommer ett kapitel där vi tittar lite närmare på konventioner. Därefter kommer kapitel om felhantering, test, moduler och loggning, där vi bygger vidare på de koncept vi gått igenom i de tidigare kapitlen. Boken avslutas med ett kapitel om några centrala tredjepartsbibliotek, samt ett exempelpprojekt.

## 1.8 Sammanfattning

I detta kapitel har vi kollat på vad Python är, förutsättningar som krävs för att få största möjliga behållning av boken såsom att läsaren förväntas ha en installerad Pythonmiljö, samt vissa grundläggande begrepp och koncept. Vi avslutade med en översikt av kommande kapitel.

## 1.9 Uppgifter

1. Vem skapades Python av?
2. Pythonversioner skrivs enligt notationen A.B.C. Vad står A, B respektive C för?
3. Vad är en virtuell miljö?

# Kapitel 2

## Datatyper

I Avsnitt 1.6.3 nämnde vi att all data i Python representeras av objekt, som har olika datatyper, som i sin tur kan ha olika egenskaper. I det här kapitlet ska vi titta på några av de vanligaste datatyperna.

### 2.1 Numeriska datatyper

De flesta programmeringsspråk, Python bland dem, skiljer på heltal (1, 7, 42) av datatypen `int` och decimaltal (5.2, 21.45) av datatypen `float`.

`int, float`

#### **i** Decimalavgränsare

I svenskan använd kommatecken (,) som decimalavgränsare. I engelskan, som Python är baserat på, används punkt (.) istället. För att vara konsekventa kommer vi använda punkt (.) som decimalavgränsare i den här boken.

Vi kan använda den inbyggda funktionen `type()` för att ta reda på vilken datatyp ett objekt har.

```
type(1)
```

①

- ① Den inbyggda funktionen `type()` returnerar ett objekts datatyp. Talet 1 är ett objekt av datatypen `int`.

`int`

```
type(2.72)
```

①

- ① Den inbyggda funktionen `type()` returnerar ett objekts datatyp. Talet 2.72 är ett objekt av datatypen `float`.

`float`

### 2.1.1 Beräkningar med numeriska datatyper

operatorer

Vi kan använda ett antal olika *operatorer* för att genomföra beräkningar med numeriska datatyper.

Operatorer i programmeringsspråk är symboler som fungerar som enkla funktioner. Tabell 2.1 visar Pythons matematiska operatorer.

Tabell 2.1: Matematiska operatorer i Python

Operator	Funktion
+	Addition
-	Subtraktion
*	Multiplikation
/	Division (returnerar <code>float</code> )
//	Division (returnerar <code>int</code> )
%	Modulo (rest av division)

Nedan följer ett antal exempel på enkla matematiska beräkningar med operatorer.

```
1 + 2.72
```

3.72

```
5 - 2
```

3

```
3.14 * 7
```

21.98

```
25 / 5
```

①

- ① Division med `/` i Python resulterar alltid i ett objekt av datatypen `float`.

5.0

```
25 // 5
```

①

- ① För att få en int istället används `//`, som är operatoren för heltalsdivision.

5

```
26 // 6
```

①

- ① Om resultatet av en heltalsdivision inte resulterar i ett heltal, avrundas resultatet nedåt till närmaste heltal.

4

Modulo-operatoren `%` används för att få resten av en division.

```
9 % 3
```

①

- ① 9 är delbart på 3 så resten är 0. Det kan vi se genom att  $9 = 3 * 3 + 0$ .

0

```
10 % 3
```

①

- ① 10 är inte delbart med 3 utan lämnar resten 1. Det kan vi se genom att  $10 = 3 * 3 + 1$ .

1

## 2.2 Booleska datatyper

`bool`

Booleska datatyper, döpta efter den engelske matematikern och logikern George Boole, representerar värdena sant (`True`) eller falskt (`False`). Vi kommer att gå in närmare på hur de används i Kapitel 3.

```
type(True)
```

①

- ① Använd den inbyggda funktionen `type()` för att skriva ut värdet `True`:s datatyp.

`bool`

`True` är ett objekt av datatypen `bool`, en förkortning av *boolean*.

Nedan ser vi ett par kodexempel på hur vi kan använda jämförelseoperatorer i Python, som returnerar booleska datatyper.

```
4 > 5
```

`False`

```
5 == 5
```

`True`

! Notera att `True` och `False` skrivs med stor begynnelsebokstav.

## 2.3 Textsträngar

Text i Python tillhör datatypen `str` som står för “string”, eller sträng på svenska. I den här boken kommer vi benämna objekt av den här datatypen som “textsträng” eller `str`.

```
type('python')
```

①

① Text i Python representeras av objekt som är av datatypen `str`.

`str`

En textsträng kan skrivas inom enkla (') eller dubbla (") citationstecken. Python gör ingen skillnad på enkla eller dubbla citationstecken, men det är viktigt att vara konsekvent. En sträng som inleds med dubbla citationstecken, ", måste avslutas med dubbla citationstecken, ".

```
print('single quotes')
```

single quotes

```
print("double quotes")
```

double quotes

Hur ska man välja när man ska använda enkla eller dubbla citationstecken? Konventionen enligt Pythons officiella kodstandard PEP8 (se Avsnitt 1.6.2 samt Kapitel 7) är följande:

In Python, single-quoted strings and double-quoted strings are the same. This PEP does not make a recommendation for this. Pick a rule and stick to it. When



a string contains single or double quote characters, however, use the other one to avoid backslashes in the string. It improves readability.

Låt oss ta några exempel. I engelskan används enkla citationstecken som apostrofer, till exempel i uttryck som förkortats (It's, I'm, you're). Det blir dock problem om vi försöker använda enkla citationstecken runt en `str` med apostrof i.

```
print('It's raining again.')
```

SyntaxError: unterminated string literal (detected at line 1)

I kodexemplet ovan avslutar apostrofen i `It's` textsträngen för tidigt, och vi får ett `SyntaxError`.

Ett sätt att komma runt det är att sätta ett bakstreck (`\`) framför. Det kallas på engelska för *escaping*.

```
print('It\'s raining again.')
```

①

- ① Med ett bakstreck framför apostrofen går det bra. Lösningen går dock emot rekommendationen i den officiella kodstandarden PEP8 eftersom den gör koden svårare att läsa.

It's raining again.

```
print("It's raining again.")
```

①

- ① Om en sträng innehåller ett enkelt citationstecken, använd dubbla citationstecken runt den.

It's raining again.

```
print("Welcome!", she said.)
```

①

- ① Om en sträng innehåller dubbla citationstecken, använd enkla citationstecken runt den.

"Welcome!", she said.

Numeriska tal inom citationstecken ses också som textsträngar i Python.

```
type('42')
```

①

- ① Använd den inbyggda funktionen `type()` för att skriva ut datatypen av objektet `'42'`.

`str`

När ett numeriskt tal skrivs inom citationstecken blir det ett objekt av datatypen `str`.

Operatorm + kan användas även på textsträngar.

```
"1" + "8"
```

①

- ① När vi använder operatorm + på två eller flera objekt av datatypen `str` skapas en ny `str` som består av tecknen i objekten.

`'18'`

Här ovan ser vi att operatorm + betyder olika saker beroende på vilken datatyp objektet den används på är av. Vi kan inte använda den på två objekt av olika datatyper. I kodexemplet nedan försöker vi använda +-operatorm på ett objekt av datatypen `str` och ett objekt av datatypen `int`, vilket resulterar i ett felmeddelande.

```
"1" + 8
```

`TypeError: can only concatenate str (not "int") to str`

En operator som kan användas på objekt av olika datatyper är multiplikationsoperatorm, `*`.

```
"1" * 8
```

①

- ① Multiplikation mellan en `str` och en `int` med värde `n` upprepar textsträngen `n` gånger.

```
'11111111'
```

I kodexemplet ovan ser vi att textsträngen `'1'` multiplicerad med `8` resulterar i en textsträng som består av åtta ettor.

En textsträng som består av siffror kan omvandlas till en `int` genom den inbyggda funktionen `int()`.

```
int("1") + 8
```

9

En numerisk datatyp kan omvandlas till en textsträng genom den inbyggda funktionen `str()`.

```
"1" + str(8)
```

'18'

### **i** Fundera

En `str` kan alltså multipliceras med en `int`, men går det att multiplicera en `str` med en `float`?  
Fundera själv på vad du tror - lösningen kommer här nedanför.

Att multiplicera en `str` med en `float` går inte.

```
"1" * 8.1
```

`TypeError: can't multiply sequence by non-int of type 'float'`

Felmeddelandet förklarar vad som gick fel: en `str` kan bara multipliceras med en `int`, alltså ett heltal. Det är ju förstås rimligt - vad är

till exempel ordet `äpple` multiplicerat med en `float`, som till exempel `1.652`?

Vi återkommer till felmeddelanden i Kapitel 8, men det är en bra vana att läsa dem och försöka förstå vad de säger.

### 2.3.1 Indexering av `str`

Vi kan komma åt ett eller flera tecken i en `str` genom *indexering*. Vi kommer att återkomma till indexering i Avsnitt 2.5.

```
my_str = 'python'  
my_str[0]
```

①

- ① Index `0` är det första tecknet i strängen. Det gäller generellt i Python att indexering börjar på `0`.

`'p'`

*slicing*

```
my_str[2:4]
```

①

- ① Index `2:4` är det andra till och med det tredje tecknet i strängen. Det kallas *slicing* och resultatet kallas för en *slice*.

`'th'`

När man utför slicing ska man tänka på att det vänstra indexet är inkluderat i slicen, men det högra är inte inkluderat.

**i** Slicing kan upplevas som klurigt i början. Det kan underlätta om man tänker på index som att det pekar *mellan* värdena.

```
+---+---+---+---+---+---+
| p | y | t | h | o | n |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
```

Här ser vi hur slicen `[2:4]` innehåller tecknen *mellan* index 2 och 4.

### 2.3.2 Längden på en sträng - `len()`

Den inbyggda funktionen `len()` talar om hur många tecken en sträng innehåller.

```
len(my_str)
```

6

Mellanslag räknas också som tecken.

```
len("a b")
```

3

**i** Här ser vi en skillnad mellan `str` och datatyper som `int` och `float`: objekt av datatypen `int` har inte stöd för den inbyggda funktionen `len()`.

```
len(1138)
```

`TypeError: object of type 'int' has no len()`

### 2.3.3 Dela upp en sträng - split()

Objekt av datatypen `str` har en metod som heter `split()`. Med den kan vi dela upp en textsträng i flera textsträngar. Vi anger vilket tecken vi vill använda för att dela upp strängen. Metoden returnerar en `list` med de nya delarna. `list` är en datatyp vi återkommer till längre fram i kapitlet.

```
full_name = 'Erik Andersson' ①  
full_name.split(' ')          ②
```

- ① Deklarera variabeln `full_name` som en textsträng med värdet `'Erik Andersson'`.
- ② Exekvera metoden `.split()` på textsträngen `full_name` och ange blanksteg (`' '`) som tecknet vi vill dela strängen med avseende på.

```
['Erik', 'Andersson']
```

Exemplet ovan returnerar en `list` med två textsträngar: `'Erik'` och `'Andersson'`.

### 2.3.4 Radbyte

Om vi vill få vår textsträng att hoppa ned och fortsätta på en ny rad kan vi stöta på problem.

```
str_with_line_break = 'tyvärr kan vi inte byta  
rad hur som helst'  
str_with_line_break
```

SyntaxError: unterminated string literal (detected at line 1)

Python förväntar sig att en textsträng börjar och slutar på samma rad.

En lösning kan vara att använda sig av trippla citationstecken `"""..."""`.

```
str_with_line_break = """inom tre citationstecken kan
    ↪ vi byta
rad precis som du önskar"""
str_with_line_break
```

'inom tre citationstecken kan vi byta\nrad precis som du önskar'

Men vänta, den bytte ju inte alls rad! Vi behöver använda den inbyggda funktionen `print()` för att skriva ut vår textsträng med radbrytningar.

```
print(str_with_line_break)
```

inom tre citationstecken kan vi byta  
rad precis som du önskar

Går vi tillbaka till det förra kodexemplet hittar vi ett annat alternativ för att byta rad i en sträng: specialtecknet `\n`. Det betyder helt enkelt "fortsätt på nästa rad".

```
str_with_line_break = 'såhär kan vi också\nbyta rad'
print(str_with_line_break)
```

såhär kan vi också  
byta rad

### 2.3.5 Formatering av str

f-string

Ibland vet vi inte på förhand exakt vad det ska stå i en sträng. Det kan vara ett användarnamn, eller ett värde som inte är uträknat ännu. Då kan vi använda en *formatted string*, eller f-string som de ofta benämns.

Om vi skriver ett `f` framför en textsträng kan vi ersätta variabler inom klammerparenteser `{}` med deras värden.

```
name = 'Arthur'
print(f'Hello {name}!')
```

①

②

- ① Definiera variabeln `name` med värdet `Arthur`.
- ② Skriv ut en `f-string` som ersätter `{name}` med värdet i variabeln.

Hello Arthur!

`f-string`-metoden är väldigt användbar.

## 2.4 Variabler

Oftast vill vi skapa ett objekt och behålla det i datorns minne för att kunna använda det senare. Det gör vi genom att deklarera en *variabel*.

*variabel*  
*identifierare*

Ofta används begreppen variabel, värde och objekt lite slarvigt och man säger att man exempelvis “sparar värdet 42 i variabeln `my_int`”. Det är dock inte helt korrekt. I Avsnitt 2.3.3 deklarerar vi till exempel bland annat en variabel som vi kallar `full_name`. Den variabeln fungerar som en identifierare (*identifier*) till ett objekt av datatypen `str`. När vi vill komma åt objektet för att läsa eller ändra objektets värde använder vi identifieraren `full_name`. Det viktiga här är att `full_name` inte innehåller ett värde, utan representerar ett objekt, och det är objektet som har värdet. I Avsnitt 2.4.1 ska vi se varför den skillnaden kan vara viktig.

Samtidigt som det är bra att vara medveten om att vi egentligen inte sparar ett värde i en variabel, finns det en poäng med ett något förenklat språk. Vi kommer därför framöver ibland att använda oss av det lite slarviga sättet att använda begreppen variabel och värde.



**i** När vi deklarerar en variabel avgör ofta Python själv vilken datatyp objektet i variabeln ska få. Detta skiljer Python från många andra programmeringsspråk där vi själva behöver ange datatyp när vi deklarerar en variabel.

I Avsnitt 5.9 står det mer om för- och nackdelar med Pythons sätt att hantera datatyper.

En variabel deklarerar med likhetstecknet, =.

```
a = 2 ①  
b = 5.3 ②  
a * b ③
```

- ① Deklarera variabeln `a` som en identifierare till ett objekt av datatypen `int` som har värdet 2.
- ② Deklarera variabeln `b` som en identifierare till ett objekt av datatypen `float` som har värdet 5.3.
- ③ Använd operatoren `*` för att beräkna `a` multiplicerat med `b`.

10.6

Variabler kan också innehålla resultat av beräkningar.

```
a = 2  
b = 5.3  
c = a * b ①  
print(c) ②
```

- ① Deklarera variabeln `c`. Den är en identifierare till ett objekt med produkten av `a` multiplicerat med `b` som värde. Vilken datatyp objektet är av beror på resultatet av operationen.
- ② Skriv ut värdet i objektet som identifieras av variabeln `c` med hjälp av den inbyggda funktionen `print()`.

10.6

### 2.4.1 Identifierare och identitet

*identitet*

En variabel är alltså en identifierare som representerar ett visst objekt. Ett objekt kan dock identifieras av flera variabler samtidigt! Däremot har alla objekt som skapas i Python en unik *identitet* (se Avsnitt 1.6.3), som inte ska förväxlas med dess identifierare.

Den inbyggda funktionen `id()` returnerar ett objekts identitet.

```
a = 42 ①  
id(a) ②
```

- ① Deklarera variabeln `a`.
- ② Använd den inbyggda funktionen `id()` för att skriva ut `a`:s identitet.

140638183671312

Flera variabler kan alltså identifiera samma objekt. Se kodexemplet nedan:

```
a = ['a'] ①  
b = a ②  
id(a) == id(b) ③
```

- ① Deklarera variabeln `a` som en identifierare till ett objekt av datatypen `list` med ett element, textsträngen `'a'`.
- ② Deklarera variabeln `b` som en identifierare till variabeln `a`.
- ③ Jämför variabelernas identiteter med jämförelseoperatoren `==`.

True

`a` och `b` har samma identitet, och representerar samma objekt.

Det kan vara lätt att tro att vi gör ett nytt objekt när vi deklarerar `b = a`, men det gör vi inte. Vi fortsätter med kodexemplet:

```
b[0] = 'b'
print('a:', a)
```

①

②

- ① Uppdatera b och ändra det första elementets värde till textsträngen 'b'.
- ② Skriv ut a.

a: ['b']

Värdet i a har ändrats, fast vi i koden skrev att vi ville ändra b. Det är för att a och b båda identifierar samma objekt.

För att skapa ett nytt objekt kan vi använda metoden `copy()`.

```
c = b.copy()
id(b) == id(c)
```

①

②

- ① Använd metoden `copy()` på b för att skapa en kopia. Variabeln c är en identifierare till det nya objektet.
- ② Jämför variabelernas identiteter.

False

b och c representerar olika objekt.

Vi kan uppdatera värdet i c.

```
c[0] = 'c'
print('b:', b)
print('c:', c)
```

①

②

- ① Uppdatera c och ändra det första elementets värde till textsträngen 'c'.
- ② Skriv ut b och c.

b: ['b']

c: ['c']

Endast c har ändrats.

Här har vi sett att variabler i Python inte själva innehåller värden, utan är identifierare till objekt, och att ett och samma objekt kan ha flera variabler som identifierar det samtidigt.

## 2.5 Datastrukturer

Tidigare i kapitlet har vi stött på ett antal objekt av olika datatyper: `int`, `float`, `bool` och `str`. I det här avsnittet ska vi titta närmare på objekt som har en lite annan sorts datatyper - datastrukturer. De används för att samla ihop och bearbeta data på olika sätt. Olika datastrukturer har olika egenskaper och används vid olika tillfällen. Vi kommer i det här avsnittet kolla på `tuple`, `list`, `dict`, `set`, `frozenset` och `range`.

Faktum är att `str` också räknas som en datastruktur. Anledningen till det är att objekt av datatypen `str` har tillgång till många av de metoder som andra datastrukturer har, bland annat indexering och stöd för den inbyggda funktionen `len()`.

! I de följande avsnitten återkommer vi till begreppet *metod*. Det är ett begrepp vi kommer att gå in på mer i detalj i Kapitel 6, men i nuläget kan vi nöja oss med att det är funktioner som hör till objekt, och att objekt av olika datatyper har tillgång till olika metoder.

Tabell 2.2 visar de vanligaste datastrukturerna och deras särskiljande egenskaper.

Tabell 2.2: Olika typer av datastrukturer i Python

Typ	Särskiljande egenskaper
<code>tuple</code>	En samling av noll eller flera element. <i>immutable</i>
<code>list</code>	En samling av noll eller flera element. <i>mutable</i>
<code>dict</code>	En samling av nyckel-värde-par. <i>mutable</i>
<code>set</code>	En samling av unika värden, oordnad. <i>mutable</i>
<code>frozenset</code>	En samling av unika värden, oordnad. <i>immutable</i>

Typ	Särskiljande egenskaper
<code>range</code>	En talföljd. <i>immutable</i>
<code>str</code>	En textsträng. <i>immutable</i>

*element*

Datastrukturer består vanligtvis av ett eller flera värden. Dessa värden kallas även *element*.

### 2.5.1 tuple

*tuple*

En `tuple` är en datastruktur som definieras genom att skriva ett eller flera värden separerade med komma (,) eller genom den inbyggda funktionen `tuple()`.

```
my_tuple = 4, 8, 15, 16, 23, 42
my_tuple
```

`(4, 8, 15, 16, 23, 42)`

För tydlighet är det vanligt att värdena i en `tuple` också skrivs inom parenteser `()`.

```
my_tuple = (4, 8, 15, 16, 23, 42) ①
my_tuple
```

① Här har vi definierat samma `tuple` som i kodexemplet ovan, men med värdena inom parenteser `()`. Resultatet är detsamma.

`(4, 8, 15, 16, 23, 42)`

Precis som med `str` kan vi komma åt specifika element i en `tuple` genom indexering.

```
my_tuple[0]
```

**i** Python, tillsammans med flera andra programmeringsspråk så som C och Java, använder *nollindexering*. Det betyder att det första elementet i en datastruktur har index 0, det andra elementet har index 1, och så vidare.

För att komma åt det sista elementet i en samling kan vi använda index -1.

```
my_tuple[-1]
```

42

För att komma åt det tredje elementet från slutet skriver vi

```
my_tuple[-3]
```

16

Vi kan komma åt fler element samtidigt genom att använda *slicing*.

```
my_tuple[2:5]
```

①

① Använd *slicing* för att skriva ut en del av `my_tuple`.

(15, 16, 23)

När en `tuple` väl är definierad går det inte att ändra värdena i den. På engelska kallas det att en `tuple` är *immutable*.

```
my_tuple[1] = 5
```

①

① Försök att ändra ett värde i en `tuple` resulterar i ett felmeddelande som talar om att det inte går att ändra ett värde i en `tuple`.

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

Andra typer av datastrukturer kan konverteras till en `tuple` genom `tuple()`-funktionen.

```
my_str = 'python'
my_tuple = tuple(my_str)
my_tuple
```

①

① Omvandla `my_str` till en tuple.

```
('p', 'y', 't', 'h', 'o', 'n')
```

Antalet värden i en tuple får vi genom funktionen `len()`, precis som med en `str`.

```
my_tuple = (1, 2, 3)
len(my_tuple)
```

3

## 2.5.2 list

`list`

En `list`, eller lista på svenska, är en annan typ av datastruktur. Vi kommer härnäst omväxlande kalla dem för `list` eller “lista”. Värden definieras inom hakparenteser `[]` eller genom den inbyggda funktionen `list()`.

```
my_list = [42, 8, 16, 23, 15, 4]
my_list
```

```
[42, 8, 16, 23, 15, 4]
```

`len()` funkar även på en `list`.

```
len(my_list)
```

6

Även listor stöder indexering och *slicing*.

```
my_list[1]
```

8

```
my_list[3:5]
```

[23, 15]

Till skillnad från en tuple är en list *mutable* - du kan ändra värden inuti en list efter att den definierats.

```
my_list[1] = 5  
my_list
```

①

① Byt ut värdet vid index 1 mot 5.

[42, 5, 16, 23, 15, 4]

```
my_list[2:4] = [12, 39]  
my_list
```

①

① Vi kan ändra flera värden samtidigt med *slicing*.

[42, 5, 12, 39, 15, 4]

Att en list är *mutable* innebär att den har en mängd metoder som inte finns tillgängliga på datatyper som är *immutable*. Tabell 2.3 visar de metoder som finns tillgängliga på objekt av datatypen list.

Tabell 2.3: Metoder på objekt av datatypen list

Metod	Beskrivning
append(x)	Lägger till ett element x sist i en lista.
extend(iterable)	Lägger till alla element i *iterable* sist i en lista.
insert(i, x)	Lägger till ett element x vid index i.



Metod	Beskrivning
<code>remove(x)</code>	Tar bort den första förekomsten av ett element med värdet <code>x</code> . Lyfter ett <code>ValueError</code> om inget sådant element finns i listan.
<code>pop([i])</code>	Tar bort elementet vid index <code>i</code> och returnera det. <code>pop()</code> utan argument tar bort och returnerar det sista värdet i listan.
<code>clear()</code>	Tar bort alla element i listan.
<code>index(i[, start[, end]])</code>	Returnerar index för det första elementet i listan som har värdet <code>x</code> . Lyfter ett <code>ValueError</code> om inget sådant element finns i listan. De frivilliga argumenten <code>start</code> och <code>end</code> tolkas som en <i>slice</i> och kan användas för att söka i en del av listan.
<code>count(x)</code>	Returnerar antalet element med värdet <code>x</code> som förekommer i listan.
<code>sort()</code>	Sorterar en lista.
<code>reverse()</code>	Vänder en lista bak- och fram.
<code>copy()</code>	Returnerar en kopia av listan.

Eftersom en `list` är *mutable* går det även att lägga till nya värden i den. Det göra vi genom `.append()`-metoden.

`append()`  
`insert()`

```
my_list.append(75)
my_list
```

① Lägg till värdet 75 sist i `my_list`.

`[42, 5, 12, 39, 15, 4, 75]`

`append()` lägger till värdet sist i vår `list`. Vill vi lägga till ett värde på en annan plats kan vi använda `insert()`-metoden och ange önskat index.

```
my_list.insert(3, 12)
my_list
```

- ① Lägg till värdet 12 vid index 3.

[42, 5, 12, 12, 39, 15, 4, 75]

För att ta bort värden i en lista används antingen `remove()`- eller `pop()`-metoderna.

`pop()`,  
`remove()`

`pop()` tar bort värdet vid angivet index, men funktionen returnerar också det borttagna värdet.

```
removed = my_list.pop(3)  ①  
print(removed)           ②  
my_list                  ③
```

- ① Definiera variabeln `removed` och ge den värdet vid index 3 i `my_list`.  
② Skriv ut värdet i `removed`.  
③ `my_list` har inte längre värdet 12 vid index 3.

12

[42, 5, 12, 39, 15, 4, 75]

`remove()` letar upp första förekomsten av det angivna värdet och tar bort det.

```
my_list.remove(5)  ①  
my_list
```

- ① Ta bort den första förekomsten av värdet 5 i `my_list`.

[42, 12, 39, 15, 4, 75]

Försöker vi att ta bort ett värde som inte finns i en `list` resulterar det i ett felmeddelande.

```
my_list.remove(6)
```

`ValueError: list.remove(x): x not in list`

Vi kan ordna innehållet i en `list` med metoden `sort()`.

```
my_list.sort()
my_list
```

```
[4, 12, 15, 39, 42, 75]
```

```
sort()
```

**i** Metoden `sort()` fungerar bara på objekt av datatypen `list`.

Om vi har en `list` och vill ha ett objekt som är *immutable* istället kan vi omvandla den till en `tuple` med funktionen `tuple()`.

```
my_tuple = tuple(my_list)
my_tuple
```

```
(4, 12, 15, 39, 42, 75)
```

```
my_tuple.insert(1, 8)
```

①

- ① Försök att lägga till ett värde till en `tuple` ger ett felmeddelande som talar om att `insert()`-metoden inte fungerar. En `tuple` är ju *immutable* och kan inte ändras.

```
AttributeError: 'tuple' object has no attribute 'insert'
```

- i** Som vi såg i Avsnitt 1.6.3 kan inte ett objekts datatyp ändras efter att den skapats. När vi “omvandlar” en list till en tuple skapar vi egentligen ett nytt objekt av datatypen tuple och ger den samma innehåll som fanns i listan.

Vi kan se ett kort exempel:

```
a = [2, 4, 5]                                ①
print('a identity as a list:', id(a))        ②
a = tuple(a)                                 ④
print('a identity as a tuple:', id(a))       ③
```

- ① Deklarera variabeln `a` som en list.
- ② Skriv ut variabeln `a`:s identitet med den inbyggda funktionen `id()`.
- ③ Omvandla variabeln `a` till en tuple.
- ④ Skriv ut variabeln `a`:s identitet en gång till. Den skiljer sig från den tidigare identiteten, eftersom vi inte har gjort om `a` till en tuple, utan har skapat ett nytt objekt av datatypen tuple, och gjort så att `a` refererar till det objektet istället.

```
a identity as a list: 140638104383808
```

```
a identity as a tuple: 140638104384128
```

Vi ser att vi egentligen inte har omvandlat `a` från en list till en tuple, utan har skapat ett nytt objekt och ändrat så att variabeln `a` identifierar det nya objektet istället.

Det gamla objektet, alltså listan, kommer att tas bort ur datorns minne eftersom det inte längre refereras till av en identifierare.

På motsvarande sätt kan vi omvandla en tuple till en list.

```
my_tuple = (2.72, 3.13, 42)
my_list = list(my_tuple)
my_list
```

[2.72, 3.13, 42]

Datastrukturer kan innehålla varandra. I kodexemplet nedan skapar vi en `tuple` som bland annat innehåller en `list`.

```
my_tuple = (12, [1, 4, 7], 'apple')
my_tuple
```

①

① Definiera en `tuple` som innehåller en `int`, en `list` och en `str`.

(12, [1, 4, 7], 'apple')

Även om en `tuple` är *immutable* går det att ändra värden i en `list` inuti en `tuple`.

```
my_tuple[1][0] = 3
my_tuple
```

①

① Ändra värdet vid index 0 i listan som finns vid index 1 i `my_tuple`.

(12, [3, 4, 7], 'apple')

Det här är ett exempel på nästlad indexering (*nested indexing*). [1] är det andra värdet i `my_tuple`, alltså listan [1, 4, 7], och [0] är det första värdet i listan.

in

För att leta efter värden i datastrukturer som `tuple` och `list` används nyckelordet `in`. Det resulterar i `True` om det sökta värdet finns i listan.

```
fruits = ['apple', 'banana', 'orange']
'banana' in fruits
```

True

Annars resulterar det i False.

```
'lemon' in fruits
```

False

in kan negeras genom att man sätter nyckelordet not framför.

```
'pineapple' not in fruits
```

True

Såväl tuple som list kan kombineras med +-operatoren.

```
vegetables = ['tomato', 'cucumber', 'onion']  
vegetables + fruits
```

```
['tomato', 'cucumber', 'onion', 'apple', 'banana', 'orange']
```

Om det handlar om stora mängder data kan extend()-metoden vara ett bra alternativ till +-operatoren, då extend()-metoden är snabbare.

```
vegetables.extend(fruits)  
vegetables
```

```
['tomato', 'cucumber', 'onion', 'apple', 'banana', 'orange']
```

Ofta vill vi göra något med värdena i en datastruktur - utföra en beräkning eller liknande. Ett smidigt sätt att göra det i Python kallas för *list comprehensions*.

```
my_list = [1, 2, 3, 4, 5] ①  
squares = [n ** 2 for n in my_list] ②  
squares
```

① Definiera en lista med olika värden.

- ② Definiera variabeln `squares` som innehåller kvadraterna av varje värde i `my_list`.

```
[1, 4, 9, 16, 25]
```

I rad (2) i kodexemplet ovan ser vi ett exempel på en *list comprehension*. Inom hakparenteser (`[]`) anger vi att vi vill räkna ut kvadraten (`n ** 2`) för varje element `n` i `my_list`. Resultaten sparas i variabeln `squares` som är en ny list.

*List comprehensions* är ett bra komplement till `for`-loopar som vi går igenom i Kapitel 4.

`join()`

Om vi vill göra om innehållet i en datastruktur till en textsträng och skriva ut den kan vi använda metoden `.join()`. Se kodexemplet nedan.

```
', '.join(fruits)
```

①

- ① Exekvera metoden `.join()` på textsträngen `', '` och ge en datastruktur, till exempel en list, som argument. Alla element i datastrukturen måste vara objekt av datatypen `str`. Python skriver ut alla värden i listan, avgränsade med `', '`.

```
'apple, banana, orange'
```

Avgränsaren kan vara vilken textsträng som helst.

```
' or '.join(fruits)
```

```
'apple or banana or orange'
```

### 2.5.3 dict

`dict`

`dict` (kort för *dictionary*) är en datastruktur som används för att kombinera nycklar (`key`) och värden (`value`). En `dict` definieras inom klammerparenteser, `{key: value}` eller den inbyggda funktionen `dict(key=value)`.

```
shopping_list = {'tomatoes': 4, 'beans': 7, 'onions':  
    ↪ 3} ①  
shopping_list
```

① Definiera en dict, `shopping_list`, med klammerparenteser.

```
{'tomatoes': 4, 'beans': 7, 'onions': 3}
```

```
shopping_list = dict(tomatoes=4, beans=7, onions=3) ①  
shopping_list
```

① Definiera en dict, `shopping_list`, med den inbyggda `dict()`-funktionen. Notera att vi inte ska definiera nycklarna (`tomatoes`, `beans`, `onions`) inom citationstecken när vi använder `dict()`-funktionen. Resultatet är detsamma som i det föregående kodexemplet.

```
{'tomatoes': 4, 'beans': 7, 'onions': 3}
```

Vi kan komma åt ett visst värde (`value`) i en dict genom att indexera på nyckeln (`key`).

```
shopping_list['onions']
```

3

Om nyckeln saknas får vi ett felmeddelande i form av ett `KeyError`.

```
shopping_list['apples']
```

`KeyError: 'apples'`

För att undvika att koden stannar kan vi använda `get()`-metoden, som ger tillbaka värdet `None` om nyckeln saknas.

```
print(shopping_list.get('apples')) ①
```



- ① Använd `print()` för att skriva ut resultatet `None`.

`None`

Vi kan också definiera ett standardvärde (*default value*) för `get()` att returnera om nyckeln saknas.

```
shopping_list.get('apples', 0) ①
```

- ① Ange att metoden `get()` ska returnera värdet `0` om nyckeln saknas.

`0`

Vi kan lägga till nya nyckel-värde-par i en dict.

```
shopping_list['potatoes'] = 2 ①  
shopping_list
```

- ① Uppdatera `shopping_list` med den nya nyckeln `potatoes` och värdet `2`.

```
{'tomatoes': 4, 'beans': 7, 'onions': 3, 'potatoes': 2}
```

Vi kan också uppdatera värdena genom att indexera på nyckeln.

```
shopping_list['tomatoes'] = 1 ①  
shopping_list
```

- ① Ändra värdet som är kopplat till nyckeln `tomatoes` till `1`.

```
{'tomatoes': 1, 'beans': 7, 'onions': 3, 'potatoes': 2}
```

Om vi vill uppdatera baserat på det befintliga värdet kan vi använda operatoren `+=`. Den betyder “lägg till värdet till höger om operatoren till värdet som objektet till vänster om operatoren redan har”.

```
shopping_list['tomatoes'] += 1 ①  
shopping_list
```

- ① Ändra värdet som är kopplat till nyckeln `tomatoes` till 1 med `++`-operatoren.

```
{'tomatoes': 2, 'beans': 7, 'onions': 3, 'potatoes': 2}
```

`++`-operatoren är ett alternativ till att skriva följande kod:

```
shopping_list['tomatoes'] = shopping_list['tomatoes'] +  
↪ 1
```

De gör samma sak.

Vi kan se alla nycklar i en `dict` genom `.keys()`-metoden.

```
shopping_list.keys()
```

`keys()`  
`values()`  
`items()`

```
dict_keys(['tomatoes', 'beans', 'onions', 'potatoes'])
```

Alla värden i en `dict` får vi genom `.values()`-metoden.

```
shopping_list.values()
```

```
dict_values([3, 7, 3, 2])
```

Metoden `.items()` skapar ett objekt av datatypen `dict_items`, som innehåller en lista av `tuple`s med nyckel-värde-paren i vår `dict`.

```
shopping_list.items()
```

```
dict_items([('tomatoes', 3), ('beans', 7), ('onions', 3),  
('potatoes', 2)])
```

Vi kan använda *list comprehension* även på en `dict` - då kallas det en *dict comprehension*. I exemplet nedan använder vi `items()`-metoden på vår `dict` `shopping_list` för att skriva ut både nycklar och värden.

```
[(key, val) for key, val in shopping_list.items()] ①
```

- ① `.items()`-metoden resulterar i `tuple`s som består av nycklarna och värdena: `(key, val)`. Vi behöver alltså deklarerar två variabler i vår *list comprehension*: `key` och `val`.

```
[('tomatoes', 3), ('beans', 7), ('onions', 3), ('potatoes', 2)]
```

Nu kan vi använda funktionalitet vi gått igenom tidigare för att skriva ut innehållet i `shopping_list` på ett fint sätt.

```
print('\n'.join([f'{key.title()}: {val}' for key, val
↪ in shopping_list.items()]))
```

```
Tomatoes: 3
Beans: 7
Onions: 3
Potatoes: 2
```

Här använder vi den inbyggda funktionen `print()` för att skriva ut en f-string med nycklar och värden från en *dict comprehension* av `shopping_list`. `'\n'.join()` gör att det blir en ny rad efter varje nyckel-värde-par och metoden `.title()` gör att varje ord i en `str` får stor begynnelsebokstav.

## Skapa en dict med `zip()`

Med hjälp av den inbyggda funktionen `zip()` kan vi skapa en `dict` från två datastrukturer.

```
fruits = ['apples', 'oranges', 'bananas'] ①
amounts = (5, 3, 7) ②
```

- ① Definiera en lista, `fruits`, som innehåller våra nycklar.
- ② Definiera en `tuple`, `amounts`, som innehåller våra värden. Notera att `amounts` är en `tuple` för att demonstrera att den här metoden är möjlig med olika typer av datastrukturer. Både `fruits` och `amounts` kan vara en `list`, en `tuple`, ett `set`, en `str`, eller av olika typer av datastrukturer.

Den inbyggda funktionen `zip()` tar ett valfritt antal datastrukturer som argument och kombinerar dem till `tuple`s.

```
[(fruit, amount) for fruit, amount in zip(fruits,
↪ amounts)] ①
```

- ① Definiera en *list comprehension* och skriv ut resultatet av den inbyggda funktionen `zip()`.

```
[('apples', 5), ('oranges', 3), ('bananas', 7)]
```

Vi ser att `zip()` kombinerar värdena i `fruits` och `amounts` i `tuple`s.

Det här kan vi utnyttja och skapa en `dict` genom en *dict comprehension* i kombination med `zip()`.

```
fruits_dict = {fruit: amount for (fruit, amount) in
↪ zip(fruits, amounts)} ①
fruits_dict
```

- ① Definiera en `dict`, `fruits_dict` med hjälp av en *dict comprehension* och den inbyggda funktionen `zip()`. För varje kombination av `fruit` och `amount` skapas ett nytt nyckel-värde-par med den aktuella kombinationen.

```
{'apples': 5, 'oranges': 3, 'bananas': 7}
```

## 2.5.4 set

`set` är en *oordnad* datastruktur som innehåller *unika värden*. Att `set` kallas *oordnad* betyder att den inte håller koll på i vilken ordning värdena ligger i när den definieras. Det går därför inte att komma åt värden i ett `set` med indexering.

`set`

Att ett `set` innehåller unika värden innebär att ett visst värde bara kan förekomma en gång i ett `set`.

`set` definieras inom klammerparenteser `{}` eller genom den inbyggda funktionen `set()`.

**i** Om vi vill definiera den tomma mängden måste vi använda `set()`, eftersom `{}` skapar en tom `dict`.

Ett `set` innehåller bara unika värden. Att omvandla en `list` som innehåller flera likadana värden till ett `set` är ett enkelt sätt att göra sig av med dubblett-värden.

```
my_list = [1, 75, 1, 3, 23, 5, 8, 8, 8] ①  
my_set = set(my_list) ②  
my_set
```

- ① En `list` med ett antal värden som förekommer flera gånger: det finns två ettor och tre åttor.
- ② Omvandla `my_list` till ett `set` med den inbyggda funktionen `set()`.

`{1, 3, 5, 8, 23, 75}`

Här ser vi både hur `set()` bara behåller ett element av varje värde, och att ett `set` inte håller koll på i vilken ordning värdena ligger i från början.

Detsamma gäller när vi omvandlar en `str` till ett `set`.

```
set('monty python') ①
```

- ① Omvandlar en `str` till ett `set`.

`{' ', 'h', 'm', 'n', 'o', 'p', 't', 'y'}`

När vi omvandlar en textsträng till ett `set`, behålls endast ett av varje tecken i textsträngen. Ordningen som tecknen stod i behålls inte.

**i** När vi skriver ut eller loopar över ett `set` kommer ju elementen i *någon* ordning. Internt använder Python den inbyggda funktionen `hash()` för att avgöra ordningen. Funktionen `hash()` är utanför den här bokens omfång. Vi nöjer oss här med att konstatera att det finns ett visst mått av slumpmässighet involverat när ett `set` skapas och att vi aldrig ska skriva kod som förutsätter att elementen i ett `set` är sorterade i en viss ordning.

Datatypen `set` är *mutable*. Vi kan lägga till element i ett `set` med metoden `add()`.

```
my_set = {2, 4, 6, 8}
my_set.add(9)
my_set
```

```
{2, 4, 6, 8, 9}
```

För att ta bort element kan vi använda metoderna `remove()`, `discard()`, `pop()` eller `clear()`.

```
my_set.remove(4)
my_set
```

```
{2, 6, 8, 9}
```

Om vi försöker exekvera `remove()` med ett värde som inte finns i vårt `set` får vi ett felmeddelande.

```
my_set.remove(42)
my_set
```

```
KeyError: 42
```

Metoden `discard()` säger ingenting om värdet inte finns.

```
my_set.discard(42)
my_set
```

{2, 6, 8, 9}

Om värdet finns tar `discard()` bort det.

```
my_set.discard(2)
my_set
```

{6, 8, 9}

**i** Det kan verka lockande att bara använda `discard()` - då slipper vi ju felmeddelanden! Ofta kan det dock vara bättre att hantera fel som uppstår snarare än att tysta dem. Vi skriver mer om detta i Kapitel 8.

Metoden `pop()` returnerar ett slumpmässigt valt element ur ett `set` och tar bort elementet.

```
element = my_set.pop()
print(element)
my_set
```

6

{8, 9}

Metoden `clear()` tar bort alla element i ett `set`.

```
my_set.clear()
my_set
```

`set()`

`set` kan användas med mängdoperationer som union (`union()`, `|`), snitt (`intersection()`, `&`), differens (`difference()`, `-`) och symmetrisk

differens (`symmetric_difference()`, `^`).

Operationerna kan exekveras antingen som en metod på ett av objekten, eller med en operator mellan objekten som ska jämföras.

union  
intersection  
difference

```
A = {1, 2, 3, 7, 8, 9}
B = {2, 4, 6, 8}
```

**Union:** mängden av alla element i A och/eller B.

```
A.union(B)
A | B
```

①

{1, 2, 3, 4, 6, 7, 8, 9}

**Snitt:** mängden av alla element i A och B.

```
A.intersection(B)
A & B
```

{2, 8}

**Differens:** mängden av alla element i A men inte i B.

```
A.difference(B)
A - B
```

{1, 3, 7, 9}

**Symmetrisk differens:** mängden av element i A eller B men inte i båda.

```
A.symmetric_difference(B)
A ^ B
```

{1, 3, 4, 6, 7, 9}



**i** Notera att om operationen utförs som en metod, kan argumentet till metoden vara en *iterable* av vilken datatyp som helst.

```
A.intersection([5, 6, 7, 8])
```

```
{7, 8}
```

Det fungerar dock inte med operatorerna, som kräver att samtliga objekt är av datatypen `set`.

```
A | [4, 5, 6, 7]
```

```
TypeError: unsupported operand type(s) for |: 'set'
and 'list'
```

`set` har också en `update()`-metod, som fungerar som `union()`, men lägger till elementen från `B` i `A` istället för att returnera ett nytt `set`.

```
A.update(B)
A
```

```
{1, 2, 3, 4, 6, 7, 8, 9}
```

De andra mängdoperationerna har motsvarande metoder. De heter som operationen, följt av `_update`.

```
A.difference_update(B)
A
```

```
{1, 3, 7, 9}
```

Operatorerna kan användas på liknande sätt genom att vi skriver ett likhetstecken efter operatören.

```
A ^= B
A
```

```
{1, 2, 3, 4, 6, 7, 8, 9}
```

### frozenset

`frozenset` är en *immutable* version av `set`. Det innebär att alla metoder ovan som ändrar innehållet i ett `set` inte är tillgängliga på ett `frozenset`.

```
C = frozenset(A)
C.add(42)
```

AttributeError: 'frozenset' object has no attribute 'add'

## 2.5.5 range

En `range` är en talföljd, som oftast används för att köra en loop ett bestämt antal gånger. Det enklaste sättet att skapa en `range` ser vi här nedanför.

`range`

```
my_range = range(5)
print(my_range)
```

```
range(0, 5)
```

Vi kan använda vår `range` för att exekvera en loop, det vill säga exekvera viss kod ett visst antal gånger:

```
for i in my_range:           ①
    print(i)                 ②
```

- ① Definiera en `for`-loop över `my_range`.
- ② Skriv ut värdet i variabeln `i`.

```
0
1
2
3
4
```

Vi kommer att gå in mycket närmare på loopar i Kapitel 4.

På många sätt beter sig en `range` som en `list` bestående av talen i talföljden, men som vi ser här ovanför är det en egen datatyp. En `range` skapar inte själva listan, vilket sparar minne.

För att skriva ut värdena i en `range` kan vi använda en *list comprehension*.

```
[n for n in my_range]
```

```
[0, 1, 2, 3, 4]
```

`range()` behöver minst ett argument: `stop`. I kodexemplet ovan där vi deklarerar variabeln `my_range` är det 5, vilket resulterar i en `range` från 0 till talet innan argumentet `stop`, alltså 4.

`range()` kan ta två till argument: `start` och `step`. `start` anger talet att starta på och `step` anger antalet steg mellan varje tal.

```
my_range = range(4, 10)
[n for n in my_range]
```

①

① `start = 4, stop = 10`

```
[4, 5, 6, 7, 8, 9]
```

```
my_range = range(4, 20, 3)
[n for n in my_range]
```

①

① `start = 4, stop = 10, step = 3`

```
[4, 7, 10, 13, 16, 19]
```

Ett negativt värde på `step` räknar baklänges.

```
my_range = range(10, 0, -1)
[n for n in my_range]
```

①

① `start = 10, stop = 0, step = -1`

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

Vi kan förstås passera 0.

```
my_range = range(-10, 10, 2)
[n for n in my_range]
```

①

① `start = -10, stop = 10, step = 2`

`[-10, -8, -6, -4, -2, 0, 2, 4, 6, 8]`

## 2.6 Sammanfattning

I det här kapitlet har vi gått igenom de vanligaste datatyperna i Python och några av deras användningsområden.

*Tabell 2.4: Begrepp i Kapitel 2*

Begrepp	Förklaring
datatyp	Ett objekts datatyp avgör vilka värden objektet kan ha och vilka operationer som är möjliga att utföra på objektet.
variabel	En identifierare som representerar ett objekt.
element	Ett av objekten i en datastruktur.
indexering	Att komma åt ett eller flera element i en datastruktur genom att ange index inom [].
slicing	En variant av indexering där ett eller flera element väljs genom att ange flera index inom [], separerade med :.
<i>list comprehension</i>	Ett sätt att utföra en operation på varje element i en datastruktur och spara resultatet i en ny lista.

Tabell 2.5 visar några vanliga datatyper, Tabell 2.6 visar de vanligast förekommande datastrukturerna, Tabell 2.7 visar de vanligaste operatorerna och Tabell 2.8 visar mängdoperationerna.

Tabell 2.5: Datatyper

Datatyp	Representerar	Exempel
int	Heltal	7
float	Decimaltal	2.72
bool	Sant eller Falskt	True, False
str	Textsträng	'hello'

Tabell 2.6: Datastrukturer

Typ	Särskiljande egenskaper
tuple	En samling av noll eller flera element. <i>immutable</i>
list	En samling av noll eller flera element. <i>mutable</i>
dict	En samling av nyckel-värde-par. <i>mutable</i>
set	En samling av unika värden, oordnad. <i>mutable</i>
frozenset	En samling av unika värden, oordnad. <i>immutable</i>
range	En talföljd. <i>immutable</i>
str	En textsträng. <i>immutable</i>

Tabell 2.7: Operatorer

Operator	Funktion
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo (rest av division)

Tabell 2.8: Mängdoperationer

Operation	Metod	Operator
Union	union()	
Snitt	intersection()	&
Differens	difference()	-
Symmetrisk differens	symmetric_difference()	^

## **i** Konventioner

Operatorer skrivs med blanksteg före och efter.  
Exempel:

```
my_int = 3  
i = i + 1
```

När man genomför operationer som sker enligt olika prioritetsordningar används blanksteg för att särskilja vilka operationer som har lägre prioritet.

```
a = x*2 - (y**4+3) - 1  
b = (x+y) * (x-y)
```

Namn på variabler ska bestå av endast små bokstäver. Om namnet består av flera ord ska de sammanbindas med understreck (\_). Denna namngivningskonvention kallas *snake case*.

Exempel: `my_variable`.

## 2.7 Uppgifter

1. Addera de två talen nedan och spara resultatet i en variabel som heter `my_result`. Printa ut variabeln `my_result`.

```
num_1 = 5
num_2 = 3
```

2. Multiplicera de två talen nedan och spara resultatet i en variabel som heter `my_mult_result`. Printa ut variabeln `my_mult_result`.

```
number_1 = 10
number_2 = 7
```

3. Förklara vad nedanstående kod gör.

```
10 % 3
```

4. Vi vet att  $27/6 = 4.5$ . Förklara vad nedanstående kod gör.

```
27//6
```

5. Vad är fel i nedanstående kod? Rätta till koden så det fungerar.

```
print('it's fun to learn Python!')
```

6. Vad gör nedanstående kod?

```
print("ha"*3)
```

7. Extrahera namnet “Anna” från strängen nedan genom att göra en *slice*.



```
full_name = "Anna Andersson"
```

8. Hur många tecken har strängen nedan? Använd `len()`-funktionen.

```
full_name = "Anna Andersson"
```

- 9.
- a) Genom att använda en f-string med de två variablerna `founder` och `language`, printa ut texten “Hello Guido, Python is really fun to learn!”
  - b) Du vill nu skriva ut hela namnet “Guido van Rossum” så texten blir “Hello Guido van Rossum, Python is really fun to learn!”. Gör det genom att ändra namnet i variabeln `founder` och exekvera din f-string på nytt.

```
founder = "Guido"  
language = "Python"
```

10. Varför blir koden nedan fel?

```
my_first_tuple = (10, 5, 'hi', 3)
```

```
my_first_tuple[1] = 10
```

11. Vad gör koden nedan?

```
my_tuple = ('languages', ['Python', 'Java', 'C', 'R'],  
            ↪ 'apples')
```

```
print(my_tuple[0])
print(my_tuple[-1])
print(my_tuple[1][2])
```

12. Kolla på vilken datatyp variabeln `my_tuple` har genom att använda `type()`-funktionen.

```
my_tuple = ('languages', ['Python', 'Java', 'C', 'R'],
            ↪ 'apples')
```

13. Vad gör koden nedan?

```
my_first_list = [10, 5, 'hi', 3]
```

```
my_first_list[1] = 7
```

```
my_first_list
```

14. Se vilken datatyp variabeln `my_first_list` har genom att använda `type()`-funktionen.

```
my_first_list = [10, 5, 'hi', 3]
```

15. Beräkna antalet element i variabeln `shopping_list` genom att använda `len()`-funktionen.

```
shopping_list = ['apple', 'banana', 'grapes', 'eggs',
                 ↪ 'milk']
```

16. Vi glömde att lägga till bröd (*bread* på engelska) i vår shoppinglista. Gör det genom att använda `append()`-metoden och printa ut den nya shoppinglistan.

```
shopping_list = ['apple', 'banana', 'grapes', 'eggs',  
    ↪ 'milk']
```

17. Beräkna antalet gånger siffran 7 finns i listan `my_numbers`. Använd `count()`-metoden.

```
my_numbers = [1, 7, 2, 7, 10, 7]
```

18. Konvertera `one_tuple` till en lista och spara det i en variabel som heter `one_list`.

```
one_tuple = (10, 5, 3)
```

19. Vad gör nedanstående kod?

```
my_first_dict = {'Anna': 38, 'Goran': 19, 'Lennart':  
    ↪ 59, 'Halimah': 28}
```

```
my_first_dict['Goran']
```

20. Kolla på vilken datatyp variabeln `my_first_dict` har genom att använda `type()`-funktionen.

21. Varför printas elementet 5 endast ut en gång i koden nedan?

```
my_set = {10, 5, 2, 5, 5}  
print(my_set)
```

{10, 2, 5}

22. Vi har definierat de två mängderna A och B nedan.

- a) Hitta alla element som är i både A och B. Använd `intersection()`-metoden.

- b) Hitta alla element som är i A och/eller B. Använd `union()`-metoden.
- c) Hitta alla element som är i A men inte i B. Använd `difference()`-metoden.

```
A = {1, 2, 3, 4, 5}
B = {3, 10, 5, 7}
```

23. Vi har definierat de två listorna `list_a` och `list_b` nedan. Nedanstående uppgifter kan enkelt lösas genom att konvertera de definierade listorna till mängder först.

```
list_a = [1, 2, 3, 4, 5]
list_b = [5, 6, 10, 21]
```

- a) Hitta alla element som är i både `list_a` och `list_b`.
  - b) Hitta alla element som är i `list_a` och/eller `list_b`.
24. Hitta alla unika element i variabeln `duplicate_values` genom att konvertera `tuple:n` till ett set.

```
duplicate_values = (10, 2, 2, 10, 1)
```

