Ormviskare
En introduktion till Python

Rickard Löfberg

August 9, 2017

1 Introduktion

Syntax

Så – du har bestämt dig för att bli en ormviskare, lära dig pytoniska. Då är detta boken för dig men om du hade förhoppningar på att lära dig kommunicera med ormar efter att du läst denna boken måste jag tyvärr göra dig besviken. Du kommer att lära dig är pytoniska, men inte den betydelsen att lära sig prata med pytonormar utan det handlar om att lära sig programeringsspråket *Python*. Om du dessutom bor i Sverige skulle jag rekommendera att du lär dig huggormianska istället (mycket mer användbart), tyvärr känner jag för tillfället inte till någon bok som täcker just det ämnet. Bristen på litteratur inom det här området, sorligt som det är, lämnar jag till dig som en framtida idé att vidareutveckla. Nu tänker du kanske lite besviket att *en bok som lär mig att kommunicera med ormar hade varit mycket mer användbart, varför ens läsa en bok om programering?*

Jo, antagligen vill du lära dig programmering och förhoppningsvis i programeringsspråket Python. Om du är helt ny till programering eller så har du redan lite erfarenhet av det kan detta vara boken för dig. Utformningen i boken är att lägga stor fokus på de delar som oftast förbises eller inte får mycket tyngd i annan litteratur om programering. Oftast är det mycket fokus på att få personen att börja skriva kod snabbt och få igång dem att börja koda för att koda är det bästa sättet att lära sig, sedan lämnas den som lär sig ofta till att söka och ta till sig av information själv om hur man bäst löser ett problem eller en uppgift. Jag har själv erfarat detta vilket ofta lätt till att när jag väl hittat svart tänkt tanken:

– Varför kunde inte bara någon sagt detta till mig?

Visserligen är det viktigt att lära sig söka upp information självständigt och kunna hitta lösningar till problem på men om man skulle lägga lite mer

tid på att lära sig grunderna först kan man undvika en hel del sökande. Om man inte riktigt heller har lärt sig begreppen eller vad det är för problem man vill lösa blir det bara svårare att söka. Därför kan det vara viktigt att spendera tid på att förstå grunderna både för att veta vad som är möjligt och också för hur man kan hitta mer information om problemet. Eller som det gammla talesättet går "weeks of coding can save you hours of planning".

Upplägget i denna boken är att först kommer vi gå igenom grunderna och prata om dem. Fokusen kommer ligga på en del detaljer om varför vissa saker fungerar som de gör för att du ska få en bra förståelse vad som händer när man gör vad. Det kommer inte vara väldigt tekniska detaljer utan snarare konceptuella, vilket förhoppningsvis ger dig en förståelse och grund att stå på. Vidare kommer andra halvan av bkoen fokusera på mer exempel och mer "meta programering" som abstrakta datatyper, pseudokod samt lite mer praktiska applikationer.

Relevant är också att det inte alltid är lätt att få datorer att göra som man vill, de kan ofta anses vara väldigt bråkiga och som att de bara vill sätta sig på tvären, nästan vara elaka mot en. Genom att programera har man mer möjlighet att få datorn att göra som man vill, även om den fortfarande kommer att bråka. Det kan också ge dig insikter i hur en dator fungerar eller varför de ibland bråkar med en.

Efter att ha läst denna boken kommer du ha lite mer förstålese för programering och hur program fungerar, kunna skriva egna enkla program och förhoppningsvis ha verktygen som kan hjälpa dig att forsätta utvecklas på egen hand eller hjälp av hemsidor och/eller andra böcker.

1.1 Tillgänglighet och licens

Detta är en bok som är skriven för att öka tillgängligheten av läromedel och litteratur på svenska för programering. Boken är skriven i Emacs med org-mode, tillgänglig under [VÄLJ LICENS] och finns att hämta på Github.

1.2 Vem är boken för?

Boken är skriven för dig som antingen inte har någon erfarenhet i programmering men har en viss teknisk kompetens eller för dig som redan använt Python lite men känner att du saknar en del förståelse för vad som försigårs ibland.

Det görs ingan antaganden om tidigare programeringserfarenhet men ett visst antagande i datorvana görs, du bör t.ex. veta vad en *filtyp* är, hur man använder sig av *kommandotolken* eller *terminalen* samt på egenhand kunna

installera *Python*. Vi kommer gå igenom hur du kan ladda ner Anaconda vilket är en platform vilket har många av de verktyg som är bra att ha när man använder Python. För att sedan skriva kod krävs det också en textredigerare. Alla operativsystem har inbyggda textredigerare och nedan följer de som medföljer olika operativsystem.

1. Windows - Notepad
2. Mac - TextEdit
3. Linux - Gedit

Alla dessa tre fungerar utmärkt när du börjar skriva kod och en fördel i början är att de inte "hjäper" sin använder med `Syntaxen` vilket gör att du måste vara mer noggrann med den kod du skriver, dock finns det andra alternativ som är mer kraftfulla och gör det lättare att skriva kod eftersom de har inbyggada funktioner för att hjälpa användaren, några av dessa är.

1. vim
2. emacs
3. atom
4. sublime

De två första av dessa är gamla och beprövade men de har dock en inlärningskurva vilket gör att de kan vara svåra att börja med, för den som vill ha något som fungerar snabbt och bra "direkt ur lådan" rekommenderas att använda någon av de två sista.

1.3 Varför lära sig programmera?

Varför ska jag lära mig programmera? Här är tre anledningar att lära sig, först och främst är det ett bra sätt att **öva problemlösning**, utöver det hjälper det en att **hantera information** och slutligen ökar det ens **förståelse för teknologi**.

Låt oss utveckla dessa svaren lite. Om du är en person som gillar problemlösning eller ofta finner att du är irriterar dig på att *något görs på ett dumt sätt* kommer du att gilla programering. Du kan programera bara för ett sätt att öva på problemlösning, precis att lösa suduko. En hemsida för att öva på problem är t.ex. hackerrank, på denna hemsidan kan du gå in och försöka lösa olika problem. Men du kanske är en person som ofta finner

att något görs på ett dumt sätt, varför gör vi dessa dumma rutiner, eller varför kan man inte göra detta på ett bättre sätt? Då är programering för dig eftersom programering ger dig verktygen att själv lösa problemet. Du behöver inte ens alltid skriva ett jättestort program, ibland kan det räcka med att visa att din lösning fungerar med ett konceptuellt program för att sedan implementera lösningen. Det finns också tillfällen då din kunskap inom programering gör att du förstår varför något fungerar på ett visst sätt p.g.a begränsingarna i teknologin.

Utöver den mentala gympan av problemlösning är programering användbart när man ska hantera stora mängder information. Varför finns det, t.ex. rutiner där man måste leta upp mycket data manuellt eller lägga in data manuellt? Borde man inte kunna automatisera det? Jo, det borde man kunna och det kan man. Det krävs inte alltid mycket kod för att kunna söka, extrahera och spara text. Många företag och organisationer har ofta mycket data, men utan struktur kan det vara ett hinder snarare än en fördel. Med lite programmering blir datan ens vän istället för fiende.

Slutligen och tidigare nämnt är förståelsen av teknologi. När man förstår hur något fungerar vet man dess begränsningar och får också en känsla för vad man kan och inte kan göra med det. Kanske tycker man att något fungerar på ett konstigt sätt och tänker att detta kan man förbättra. Med kunskaper om programering kan du då utvärdera om det är något man kan göra och också hur man skulle göra det. Det kan vara så att man kommer fram till att det faktiskt inte är möjligt eller i vissa fall inte värt att lägga ner all den tid det skulle ta på att ordna det i förhållande till värdet som lösningen sparar in (tid, pengar, irritation).

1.4 Vad är Python?

En pyton är en orm, *något kan också vara pyton* men *Python* är inte pyton. Python är ett programmeringsspråk vilket används mycket inom undervisning, forskning och också av många företag. Fördelarna som många ser med Python är att det är ett språk som är lätt att lära sig, ger snabba resultat och således leder till produktivitet.

En av de anledningar till att det är lättare att lära sig och att man snabbt kan komma igång med att analysera data är att Python är väldigt förlåtande då man kan skriva över datatyper, detta till skillnad från många andra språk där man måste *deklarera* sina datatyper. Vad detta innebär är att man ibland måste hålla tungan rätt i munnen eftersom det kan vara så att man ändrat på något man inte täntke ändra på. Man behöver inte heller *kompilera* det för att kunna "köra" ett program vilket gör att man snabbt

kan skriva och testa kod.

Utöver det finns det en fokus på läslighet i Python, det ska alltså vara lätt att läsa och skriva det man vill göra. Det betyder inte att andra språk inte har en fokus på läsbarhet, alla språk förespråkar att man ska skriva läsbar kod men det finns alltid en begränsning i läsbarheten beroende på hur språket är designat. Python är designat för att vara lätt att skriva och läsa (så länge det skrivs väl) vilket gör att det är ett utmärkt val. Om du är intresserad kan du läsa pythons designprinciper själv.

2 Installera Python

Låt oss installera Anaconda genom att först gå till hemsidan `https://www.continuum.io/downloads`, scrollar sedan ner en bit tills du ser *Download for Your Prefered Platform*. Under det väljer du att ladda ner `Python 3.6` till det operativsystem du använder. Efter det installerar du Anaonda som ett vanligt program till ditt operativsystem, om du vill ha mer detaljerade instruktioner för hur du installerar det kan du hitta det på dokumentationen för Anaconda. Gå till länken[1]. Välj sedan ditt operativsystem för mer information.

För att se att det har blivit installerat kan vi testa om vi kan köra python i terminalen eller kommandotolken. Om du kör windows kan du göra detta genom att trycka på *windows-knappen* och *r* samtidigt, efter att fönstret kommer upp skriver du `cmd` och trycker sedan *enter*. Nu borde du se ett svart fönster, testa att skriva `python3` i det och sedan *enter*. Om du nu information om den version av python du kör och sedan tre stycken pilar »> betyder det att python körs och väntar på dig.

I de flesta linuxdistributioner brukar det gå att trycka på *Ctrl*, *Alt* och *t* samtidigt för att öppna terminalen. Efter att du öppnat den skriver du `python3` precis som ovan och du borde då få samma resultat.

För Mac ....

.

2.1 Skript och Terminalen

Efter att vi har installerat Python (Anaconda) finns det två huvudsakliga sätt att köra det på, vi kan antigen köra ett *skript* (en pythonfil) eller göra det *interaktivt* genom *terminalen* [2]. Men vad är då skillnaden? I enkla drag

[1] `https://docs.continuum.io/anaconda/install/#detailed-installation-information`

[2] Terminalen kommer att användas som ett samlingsnamn för *kommandotolken* och *bash* i denna boken.

kan vi säga att den stora skillnaden mellan skript och interaktivt är att när vi använder oss av skript skriver vi först koden och kör[3] igenom all kod på en gång när vi kör programmet, men när vi gör det interaktivt skriver vi och kör koden på samma gång. Efter att vi har kört ett skript finns vår kod kvar i den filen vi skrev den i och vi kan köra den igen eller ändra koden medans när vi gör det interkativt finns koden bara kvar så länge vi befinner oss i *pythonmiljön* [4].

Ett vanligt mistag som många nya ormviskare gör är att skriva all sin kod interaktivt, det är inte fel att skriva interaktivt i sig men det bästa är att använda styrkorna och svagheterna hos de olika metoderna. Generellt rekommenderar jag att du skriver din kod i en *pythonfil*, anledningen till detta är att du sparar all din kod så om du gör ett mistag kan du gå tillbaka i koden och ändra det utan att behöva skriva om allt från början. En annan anledning är att det gör koden återanvändbar, du kan återanvända gammal kod på ett smidigt sätt. Du kan alltså gå in i dina gamla filer och kopiera kod som du redan har skrivit för att återanvända det i ny kod. Den koden som du skriver interkativt försvinner efter att du slutar använda det interkativa läget.

Men varför har vi ens det interaktiva om vi ändå ska skriva all kod som skript? Jo, det finns en del riktigt smidiga fördelar med att kunna skriva kod interaktivt. En kan t.ex. klippa och klistra in delar av sin kod i den interaktiva miljön och testa den så att man vet att man har tänkt rätt. Du kan också använda det för testa en idé eller kolla upp hur en viss *datatyp* eller *datastruktur* beter sig eller något man har glömt bort. Om det är så att man har glömt något är det också väldigt användbart för att "slå upp" saker i den interaktiva miljön. Generellt behöver man inte skriva mer än några rader kod i det interaktiva, när du börjar komma upp i över 20 rader kod i det interaktiva läget skulle jag säga att du borde överväga om inte detta borde vara ett skript? Låt oss ta en liten närmare titt på vad ett skript är och hur man använder sig av terminalen.

2.1.1 Interaktiva Python

Öppna terminalen och starta python som vi gjorde ovan. Du vet att det har startat eftersom texten i terminalen kommer att skriva ut vilken version av Python du kör och sedan tre stycken pilar `>>>` som visar att Python nu väntar på instruktioner, ett exempel finns nedan. Vad dessa tre pilar egentigen

[3] Kör betyder att vi ....

[4] En smart förklaring till *pythonmiljön*

betyder är att Python väntar euntisiastiskt på att du ska börja skriva kod och säga till Python vad du vill att hen ska göra.

```
Python 3.5.2 |Anaconda 2.5.0 (64-bit)| (default, Jul  2 2016, 17:53:06)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Vi kan redan nu utan några egentliga svårigheter börja använda Python som en enkel miniräknare genom att skriva lite olika uträkningar. Oroa dig inte för att detta är det enda vi kan använda Python till, Python är ett kraftfullt språk som vi kan göra väldigt avancerade uträkningar i men låt oss börja med det bekanta först. Nedan kan vi se några olika exempel på uträkningarna, försök lite själv, testa olika uträkningar och oroa dig inte om du får ett felmeddelande. Några operatorer som du kan testa är `+`, `-`, `/`, `*` och några lite mer ovanliga är `%`, `//` och `**`.

```
>>> 100 * 0.25
25.0
>>> 100 * (0.25 + 1)
125.0
>>> 6 / 7
0.8571428571428571
>>> 1 + 2 * 2
5
```

Som vi kan se är det ganska enkelt att göra lite enkla mattematiska uträkningar i Python och om vi använder oss av **parenteser fungerar det precis som i mattematiken, det inom parenteserna beräknas först och sedan beräknas resten** samt att multiplikation beräknas före addition. I det interaktiva läget skriver Python direkt ut svaret på "frågan" och vi kan se det på nästa rad i terminalen.

Det går också bra att använda Python som ett litet uppslagsverk när man glömmer bort hur något specifikt fungerar, detta gör man genom att skriva `help()` och sedan kan man söka efter det man vill ha info om och när man är klar skriver man `quit` och trycker enter för att komma ut.

```
Welcome to Python 3.5's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.5/tutorial/.
```

```
Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> quit

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.
```

Något att ha i åtanke är dock att informationen i den interaktiva dokumentationen kan vara lite kortfattad ibland och om man vill ha mer information är internet en bra källa och det kan vara en bra ide att kolla den fullständiga dokumentationen för det du letar efter.

2.1.2 Skript

Ett skript är kort och gott en textfil som innehåller pythonkod, något viktigt för att datorn ska veta vad det handlar om en pythonfil är att vi använder oss av *filtypen* `py`, eller med andra ord den fil vi arbetar i måste sluta på *.py*. Detta är precis som när vi arbetar med olika filtyper, till exempel slutar Microsoft Word filer på *.docx* och textfiler slutar på *.txt*. Allt detta gör är att det låter datorn veta vad det handlar om för *filtyp* så att datorn då vet hur den ska hantera den. När datorn ser att något slutar på *.docx* tänker den "Ahhha, jag vet att detta ska öppnas i Microsoft Office" eller "Åhhh, en .txt fil, nu måste jag öppna detta med en textredigerare". Om datorn ser en .py fil utbrister den såklart "Woohoo, nu ska vi köra lite pythonkod!". Alla de pythonfiler vi skapar kan alltså ha ett namn som börjar med ett informativt namn och slutar på *.py*, men en sak som är bra att tänka på är att om man har mellanrum i *filnamnet* kan det leda till lite problem när vi vill köra ett skript ifrån *terminalen*. En vedertagen standard är att använda understreck istället för mellanrum när man skapar sina filer. Som ett exempel är det generellt bättre att en fil heter `min_kod.py` istället för `min kod.py`. Det

är också vanligt att använda orm-skrift och börja varje nytt ord (utom det första) med stor bokstav men också skrippa understrecket, filen skulle i så fall heta `minKod.py`.

Men hur ser det då ut i en pytonfil? Det är tyvärr inte så fruktansvärt exalterande, utan det är bara en massa kod (text) som är strukturerat på ett visst sätt (enligt *syntax* [LÄNK HÄR]). Om vi vill göra samma sak som vi gjorde i det interaktiva läget skulle vi alltså skriva det i en fil och sedan göra alla dessa beräkningar på en gång, i detta fallet skulle det se ut på följande sätt.

```
100 * 0.25
100 * (0.25 + 1)
6 / 7
```

De uträkningar vi vill göra lägger vi alltså bara in i vår pythonfil. För att köra detta får vi navigera oss till mappen där filen ligger med hjälp av vår terminal[5]. När vi hittat dit skriver vi helt enkelt in `python3` följt av ett mellanrum och `namnet_på_filen.py`, vi skriver alltså inte exakt detta utan in det namn på vår pythonfil som vi har sparat, i mitt fall heter den exempel1.py vilket gör att det fullständiga kommandot är `python exempel1.py`. Med konfigurationen på mitt system så skrivs inte hela vägen till vart i systemet jag befinner mig ut, utan det skrivs bara `$`, om du använder ett Unix-baserat system kan det se annorlunda ut för dig.

```
$ python3 exempel1.py
$
```

Men, detta ser ju lite lustigt ut, vi får inte ut några resultat alls, vad beror det på? Jo, det finns en väldigt viktig skillnad mellan det interaktiva läget och att skriva skript, vilket är att när du skriver direkt i det interaktiva kommer den direkt att ge dig resultatet av det du har skrivit men om du vill att skriptet skriver ut svaret måste du säga det genom att använda *funktionen* `print()`. Vad som händer när vi kör skriptet nu är att den gör alla uträkningar (jag lovar) även om vi inte ser det eftersom den inte skriver svaret. Låt oss då lägga till en `print()` runt alla våra beräkningar i skriptet och se vad som händer, skriv till `print()` i ditt skript så att det ser ut som det nedan, glöm inte att spara filen när du har gjort det. Tänk på att parenteserna i `print` ska omsluta det som du vill ska skrivas ut.

[5]Hur du navigerar dig i terminalen beror på om du använder windows eller unix/max.

```
print(100 * 0.25)
print(100 * (0.25 + 1))
print(6 / 7)
```

Om vi nu testar att köra det igen så kommer du förhoppningsvis få något som liknar det nedan.

```
$ python exempel1.py
25.0
125.0
0.8571428571428571
$
```

Taadaaa, vi fick samma svar som när vi använda terminalen vilket är bra, det betyder ju att båda sätten att göra det på ger samma svar. Dessutom kan vi köra skriptet hur många gånger vi vill utan att behöva skriva om all kod samt kopiera den, förändra den eller lägga till mer kod. Visst, nu börjar det kanske låta lite tjatigt men efter att ha sett personer som fortfarande använder sig av det interaktiva läget efter att ha programmerat ett tag är det viktigt att ponjektera detta.

För att sammanfatta det vi precis har gått igenom, majoriteten av din kod ska skrivas som skript och sparas som pythonfiler. När du skriver pythonkod skriver du det i en fil och om det är någon kort bit kod du vill testa eller om det är något du har glömt hur man gör är det interaktiva läget perfekt för att snabbt kolla upp det. Båda fyller en funktion men det gäller att använda dem till rätt sak, var sak har sin plats.

2.1.3 Syntax med exempel

En väldigt viktig aspekt i Python är syntax men för att förstå syntax måste vi anväda oss av lite mer avancerade exempel, du förväntas inte förstå allting i koden. Det kan vara en bra idé att komma tillbaka hit och se om du förstår det när du har kommit längre i boken. Med allt detta sagt, låt oss först ta en titt på koden och sedan gå in i lite mer detalj på vad *syntax* är och varför det är viktigt.

I koden finns det inga kommentarer, detta är inget jag rekommenderar utan de har blivit utelämnande för att göra det lite mer klurigt, vi kommer att diskutera kommentarer i senare kapitel.

```
import math
```

```
def is_prime(nummer):
    if not isinstance(nummer, int):
        return False
    elif nummer == 1 or nummer == 3:
        return True
    elif nummer % 2 == 0:
        return False
    else:
        for i in range(1, int(math.sqrt(nummer))):
            if nummer % i == 0:
                return True
    return False

print(is_prime(3))
print(is_prime(4))
print(is_prime(5))
print(is_prime(6))
print(is_prime(7))
```

Det vi ser här är ett enkelt primtest som kommer att retunera *sant* ifall numret vi testar är ett primnummer och *falskt* ifall det inte är det. I koden finns det också lite tester där vi redan vet svaret för att testa att koden fungerar på dessa nummer. De test vi använder är inte fullständiga men de är tillräckliga för vårt ändamål. Om vi kör skriptet kommer vi att få följande resulat, du kan läsa mer om vad *sant* (`True`) och *falskt* (`False`) i KAPITEL [LÄNK TILL KAPITEL].

```
True
False
True
False
True
```

Alltså att 3 är ett primtal, 4 är inte det, fem är ett primtal, 6 är inte det och att 7 är ett primtal. Testa gärna att kopiera koden till det egna skript och kör den. Testa också att ändra numren men ha i åtanke att om numret är för högt kan det ta en stund att köra koden.

Men låt oss nu ta en titt på hela det här konceptet med syntax, vad är det engentligen och hur relaterar det till vårat exempel?

När det kommer till olika *språk* har de en *syntax* för svenska och andra naturliga språk kallar vi denna syntaxen för *grammatik*. Det är "reglerna"

för hur vi använder språket, i mänskliga språk handlar det oftast att om vill bli förstånd följer man konventioner för hur språket ska användas, i vilken ordning orden kommer. När det kommer till *datorspråk* är dessa regler lite hårdare, en människa kan förstå en annan människa även om de inte har alla orden i "rätt" ordning. Tyvärr är detta något som datorer har svårare för och därför måste vi vara extra noggranna när vi pratar med datorn för att inte förvirra den.

Exempel på olika språk är t.ex. HTML (Hypertext Markup Language), Java, XML (eXtendable Markup Language), osv.. Notera att av dessa är bara Java ett *programeringsspråk*.

I Python finns det viss syntax som måste följas för att Python inte ska bli förvirrad, den viktigaste delen (där misstag oftast sker) är när det kommer till *indrag* av text. Detta är något som Python är väldigt petig med och det är därför något man måste vara extra nogrann med själv. Man kan tänka sig att koden sker i lager där den kod med minst indrag (längst till vänster) är den som körs först och sedan går man till nästa lager och utför den koden först, men ingen kod med mindre indrag kommer heller att köras innan det i det undre lagret har körts klart. Det kanske låter lite förvirrande men i enkelhet kommer man vid olika tillfällen att avsluta rader i python med ett kolon `:` och varje gång man ser det vet man att nästa rad kommer vara indragen. Detta används t.ex. för if-satser och loopar men mer om detta senare, nedan följer ett par exempel som du kan testa själv om du vill.

```
if 1 < 2:
    print("Det är sant")
print("Klar")
```

I detta fallet kommer programmet först att skriva ut `Det är sant` och sedan `Klar`, den kollar om 1 är mindre än 2 och eftersom det är det skriver den ut koden "i nästa lager"; när den sedan är färdig med det "hoppar den upp" ett lager och skriver "klar". Ett annat exempel är loopar som fungerar likadant.

```
for nummer in range(10):
    print(nummer)
print("Klar")
```

Något annat som oftast kan orsaka problem är att man inte matchar sina parenteser, det är viktigt att för varje parentes som man börjar, måste man också avsluta den.

En del av syntaxen som väldigt ofta orsakar ett problem för de som är nya till python är det tidigare nämnda

3 Var är min variabel?

En av de viktigaste funktionerna som program har är att kunna spara data temporärt för att manipulera och sedan spara eller skriva ut resultatet. För att spara data temporärt används *variablar*, temporärt betyder mer specifikt att datan sparas i RAM-minnet vilket oftast jämförs med datorns närminne. Efter scriptet (programet) har kört klart och manipulerat datan/informationen på det sättet man vill försvinner datan ur minnet, om man inte har valt att spara datan på ett eller annat sätt på datorns hårdisk vilket oftast jämförs med datorns långtidsminne. Det är inte alltid lätt att riktigt förstå hur allt detta fungerar eftersom allting sker elektroniskt, därför kan det vara användbart att använda metaforer för att bygga upp en mental bild av hur programmet fungerar.

När vi pratar om RAM-minnet i en Pythonkontext kan vi föreställa oss det som en byrå med väldigt många lådor som alla representerar olika delar av RAM-minnet, varje låda är också numrerad. Utöver det har vi också en referensbok och när vi vill spara någon data lägger vi den i lådan och skriver i referensboken vad det var vi sparade (variablenamnet) och i vilken låda den ligger. Vi kan alltså säga att en *variabel* refererar till den data vi har i minnet, de fungerar som *referenser*. En viktig faktor när det kommer till datorer är att de – precis som mäniskor – har en begränsing i sitt närminne, nu för tiden utrycks det i Gigabyte (GB) och vanligtvis har en dator ungefär 4, 6 eller 8 GB närminne (RAM). Det viktigaste att komma ihåg är att det finns en begränsing på hur mycket datorn kan hålla i minnet samtidigt.

Låt oss nu ta en titt på ett exempel och se vad som händer när vi *tillskriver data till en variabel*, refererar en ny variabel till samma data och slutligen vad som händer när vi ändrar *referenser*.

```
>>> x = 5
>>> y = x
>>> x = 10
>>> print(x)
>>> 10
>>> print(y)
>>> 5
```

Om vi tittar på [REF TILL NÅGOT] ser vi att den första raden `x = 5` gör att vi skapar vårt värde fem, vi skriver att vår referens till detta värdet är `x`, om vi tänker det i samma termer som metaforen ovan så är skriver vi i vår loggbok att vi har variabelnamnet `x` som refererar till en del av

minnet som håller värdet 5. Det som Python gör automatiskt åt oss är att den tar hand om var i minnet det ska förvaras och vilket "nummer" vi använder för att referera till det. När vi på rad 2 skriver `y = x` betyder det att vi säger att `y` ska referera till samma värde som `x`, programmet kommer alltså att kolla i vår loggbok efter vilket värde `x` pekar till för att sedan använda samma referens åt `y`. Python är väldigt snäll och gör det återigen automatiskt åt oss, vi behöver inte alls tänka på vad som hamnar i loggboken utan bara att både `x` och `y` refererar till samma värde i minnet. Det som händer på rad 3 är det som ibland kan vara förvirrande och ibland vara väldigt intuitivt, tillsammans är jag säker på att vi kan reda ut det ifall det skulle vara förvirrande. Nu har vi ångrat oss och tycker att `x` istället borde referera till värdet 10 istället för fem så vi skriver: `x = 10`. Det som i detta fallet kan skapa en del förvirring är att vi vet ju att `y = x` så logiskt borde då värdet för `y` också ändras, men det är inte riktigt hela sanningen och vi ska nu ta en titt på varför det blir så här.

Problematiken är att vi genom alla år i skolan har lärt oss att = betyder *är lika med* men när det kommer till programering är det inte det det betyder, för i programmering har vi ett annat tecken som visar på att något är samma eller inte och det är (bara för att öka förvirringen) två stycken likamedtecken efter varandra, alltså `= betyder /är lika med/. Det vi menar när vi skriver = är snarare något i stilen med /tilldela det värdet till höger om mig ()` till den variabel vänster om mig/. Om vi återgår till `x = 5` betyder det alltså att *tilldela 5 till variabeln x* eller att variabeln `x` refererar till värdet 5. Likså betyder det inte (som vi tidigare sett) inte `y = x` att de är samma utan att vi *tilldelar* `y` *samma värde som* `x` *refererar till*. Efter att vi har sagt att `y` ska ha samma värde som `x` kommer båda dessa värden att referera till samma data (5) men `y` kommer inte att göra det "via" `x`. Det är genom denna långsökta väg och förklarning som vi landar på att när vi sedan tittar på vilka värden `x` och `y` har genom att använda `print()` så kommer vi se att `x` refererar till värdet 10 och `y` refererar till värdet 5.

3.1 Jag döper dig här med till ..

En kort notis om vilka variablenamn är nog i sin ordning i denna delen av boken, eftersom man kan tyvärr inte bara använda sig av variabelnamn hur som helst. Det finns egentligen två stycken begränsingar på detta, den första är en *hård beränsing* som Python har eftersom vissa namn är redan paxxade samt vissa tecken som en variabel inte kan börja med och den andra regeln är en mjuk begränsning som säger att de variabelnamn du använder borde vara informatift för den som läser koden. Det skulle kunna vara du (om

6 månader) eller någon annan person som blir väldigt förvirrad ifall dessa variabelnamn inte är informativa. Ibland händer det att man kollar på kod som man skrivet för ett tag sen och tänker (ibland säger högt) för sig själv "Vilket j***a stolpskott var det som skrev den här sk*t koden" för att sedan komma på att det var en själv som skrev det. En annan anledning till att använda sig av informativa variabelnamn är den eviga skam som medföljer om någon annan kommer på dig. Om vi nu ska ta och titta på vilka de *hårda begränsningarna* är och vilka de *mjuka begränsningarna* är.

De hård begränsningarna är ganska lätta att upptäcka eftersom Python kan bli lite förvirrad när du använder dem och kommer då att meddela dig om att hen är lite förvirrad med ett meddelande som t.ex.:

```
File "<stdin>", line 1
SyntaxError: can't assign to literal
```

Vilket kan tolkas som att Pyton säger "Jag försöker tilldela ett värde till variabeln men det går inte så bra, kan du vänligen hjälpa mig". När Python ned stora hundögon (läs ormögon) så snällt ber oss om hjälp kan man inte hjälpa att man smälter lite i hjärtat och vill göra sitt bästa för att hjälpa till. I detta fallet ka vi göra det genom att dubbelkolla att vi använder oss av ett tillåtet variabelnamn för att se att allt stämmer. Låt oss dock ta och använda oss av en gammal hederlig lista för att lista upp de tecken som ett variabelnamn inte för börja på, men ikke att förglömma är de andra *reserverade nyckelorden* [LÄNK TILL DETTA] som Python har, visst vore det fint om vi hade en till lista för att lista dessa? Sagt och gjort, du kan finna dessa två listor nedan.

Men om vi då ska ta oss en titt på de *mjuka begränsningarna* och hur vi kommer bli dömd av allt och alla i vår omgivning (samt online) om vi inte använder oss av informativa variabelnamn. Det är också värt att nämna lite olika varianter som vanligtvis används för att skriva variablelnamn, vilken av dessa typer man föredrar att använda varierar från person till person men det är bra att känna till dem. Vid de tillfällen då du kan komma att arbeta med någon annans kod eller bygga vidare på ett program är det oftast konvention att följa samma standard som dem för att all kod ska vara enhetlig i *kodbasen*. Python har sina rekomedationer för hur man ska skriva variabelnamn men jag rekommenderar att du använder dig av den typ som du tycker om.

OrmSkrift eller KamelSkrift (SnakeCase/CamelCase) är den standard då du börjar med en stor bokstav och skriver ihop ord OrdSomEnEndaLångText där varje nytt ord markeras med stor bokstav.

mixadSkrift (mixed case) fungerar precis som OrmSkrift men det första ordet börjar med liten bokstav istället.

gemener$_{\text{med understreck}}$ (lowercase$_{\text{with underscore}}$) rekommenderas av Python i PEP8-dokumentet som länkades till ovan med denna konventionen skrivs allt med små bokstäver (gemener) och orden åtskiljs av understreck.

Det går i Python3 också att använda sig av olika språk för variablar, du kan till exempel om du vill använda dig av kinesiska tecken om du vill men det kan leda till problem med kompabilitet och läslighet. Självklart går det bra att använda sig av å, ä och ö men jag skulle rekommendera att använda dessa av samma anledningar jag inte rekommenderar att använda sig av kinesiska. Generelt rekommenderas det att man använder de tecken som ingår i ASCII standarden.

När det kommer till variabelnamn har man inte så mycket val när det kommer till den första bokstaven antigen vara en bokstav eller ett understreck.

4 Data har olika typer?

Tidigare har vi bara använt oss av nummer som vi har tilldelat till olika variablar men det är inte den ända *typen av data* som vi kan tilldela till en variabel utan i Python finns det många olika slags typer av data eller *datatyper*. Några av dessa är *strängar* (strings), *tupplar* (tuples), *hashtabeller* (hashtables/dictionaries), *sets* (mängder) och *listor*. Dessa är alla inbyggda i Python och de används på lite olika sätt, en del som *listor* och *hashtabeller* används för att struktuera datan och andra som *strängar* används för att hantera text. Du kan läsa mer om dessa på: `https://docs.python.org/3.6/library/stdtypes.html`.

Men eftersom vi redan har börjat med att arbeta med nummer kan vi likväl fortsätta med dem. Det finns två typer av nummer i Python och det är *heltal* (integers) och *bråktal* (floating point numbers), det som är skillnaden mellan dem är att heltal inte har några decimaler medans bråktal har decimaler. Några exempel på heltal är 1, 2, 3, 4, 5, 6, 7, osv.. och några exempel på bråktal är 0.5, 1.0, 1.5, 2.0, 2.5, osv.. Vi kan också se att till skillnad för hur man skriver bråktal i svenska med dett kommatecken så använder man sig av en punkt för att markera var decimalerna börjar i Python.

Vi har redan sett att vi kan använda oss av alla de vanliga matematiska verktygen för att addera, subtrahera, multiplicera och dividera. Låt oss utforskar lite mer hur vissa uttryck fungerar i Python kan vi helt enkelt använda oss av terminalen för att se vad som händer när vi manipulerar vår data. Om vi t.ex. skriver:

```
>>> 1 + 1
2
```

Kommer Python att skriva ut beräkningen för vad vi ville beräkna, men om vi dock gör en liten förändring och använder oss av en variabel kommer Python inte att skriva ut svaret åt oss utan spara det i minnet tills vi avslutar interkativa Python eller då ingen variabel refererar till det värdet längre.

```
svar = 1 + 1
```

Men vi kan fortfarande få ut värdet på som Python refererar till genom att skriva antingen `svar` eller `print(svar)`.

```
>>> svar = 1 + 1
>>> svar
2
>>> print(svar)
2
```

Vi ser att vi får ut samma värde genom att använda båda dessa metoder och som vi tidigare sett skulle inte värdet som variablen refererar till skrivas ut i ett skript om vi inte hade `print()` runt det.

det verkar också vara enklare att bara skriva ut variabelnamnet för att se vilket värde vi har, så varför använder du ens `print()`? Det finns en väldigt viktig anledning till att använda `print()` och det är att när man kör Python interaktivt i terminalen går det alldeles utmärkt att bara skriva ut variabelnamnet för att se vilket värde variabeln refererar till och vi kan också göra det samma när vi skriver ett skript men det som händer är att Python kommer inte skriva ut vilket värde det refererar till. Denna skillnaden i hur Pythonskript och den interaktiva pythonmiljön beter sig har lett till otaliga tillfällen av förvirring och det är därför viktigt att pongetera den. Det absolut viktigaste att komma ihåg är att om du vill att Python ska "skriva ut" ditt svar i ett script så måste du använda dig av `print()`. Hädanefter kommer jag också att följa denna standarden i kodexemplen, även när jag ger exempel i den interaktiva miljön. Om vi fortsätter att undersöka hur några av de olika matematiska uttrycken beter sig i Python, kanske division? Om vi i den interaktiva miljön skriver `1 / 2` alltså vi delar ett heltal med ett heltal, vilken datatyp borde vi få då? Jo, då får vi ett bråktal eller närmare bestämt `0.5`.

```
>>> 1 / 2
0.5
```

Okej, det verkar ju ganska logiskt att 1/2 skulle vara 0.5 men vad händer om vi istället försöker dela 1/1? Sanoligt borde vi då få 1, eller hur? Nja, inte direkt, i detta fallet får vi också ut ett bråktal. Det spelar alltså ingen roll om täljaren och nämnaren är exakt samma nummer vi kan generalisera en regel som säger att `heltal` / `heltal` = `bråktal`. Vi vet sedan tidigare också att ett `hetal` + `heltal` = `heltal`. Vi kan använda oss av alla olika kombinationer för detta för att se vad de olika kombinationerna ger och detta är summerat i tabell [REF TILL TABELL]. Jag uppmuntrar dig att testa själv och se att alting verkligen stämmer men innan dess är det värt att ta en titt på de olika *operander* som inte är så vanligt förekommande när man studerar matematik, dessa är *golvdivision* och *modulus* vilka uttrycks med `//` och `%` i Python. Om vi först tittar på golvdivison genom att testa lite olika värden så ska i se om vi kan lista ut vad det är som försegårs.

```
>>> 0 // 3
0
>>> 1 // 3
0
>>> 2 // 3
0
>>> 3 // 3
1
>>> 4 // 3
1
>>> 5 // 3
1
>>> 6 // 3
2
```

Hmmmm, detta ser ju lite mystiskt ut. Vi ser att om vi skriver 0 till och med 2 *golvdividerat* med 3 blir svaret 0 och för 3 till och med 5 är svaret 1 men för 6 är svaret 2. Vad tror du det är som händer? Okej, jag tror att du har listat ut det men låt oss ta en titt på det ändå. Vad Python (och andra programeringsspråk) gör när det kommer till golvdivision är att man räknar bara hur många gånger jämnt som TÄLJAREN? går genom NÄMANRE? om vi får några decimaler slänger vi alltså bort dem. Så för värderna 0 till och med 2 kan vi inte dela jämnt en ända gång och rundar då ner till noll, för värderna 3 till och med 5 kan vi dela jämnt en gång men inte två och rundar då ner till 1 för 6 kan vi dela två gånger och får då värdet två, jag tror att du fattar hur det fungerar nu...

Okej, nästa intresanta operator är då modelus vilket till viss del liknar golvdivision men precis som tidigare tycker jag att vi ska ta en titt på det tillsammans för att se hur den beter sig..

```
>>> 0 % 3
0
>>> 1 % 3
1
>>> 2 % 3
2
>>> 3 % 3
0
>>> 4 % 3
1
>>> 5 % 3
2
>>> 6 % 3
0
```

Nu får vi ett annat mönster istället, hur ska vi tolka det här mönstret då? Det ser ut som att det repeterar sig och mönstret vi ser är 0, 1, 2 vilket från de få exempel vi har verkar fortsätta. Testa med lite egna nummer och se om du kommer på hur denna operanden fungerar. Okej, nu har du haft tillräckligt med tid att leka och testa, här kommer svaret. Det som händer när du använder modolus är att istället för att se hur många gånger vi kan dela täljaren jämnt med nämnaren så tittar vi på vad vi får i rest (det som blir över efter att vi delat jämnt) och skriver sedan ut det värdet. Detta gör att efter att vi har delat 0, 3 och 6 med 3 har vi ingenting över eftersom det går jämnt ut, för 1 och 4 får vi ett i reset eftersom för ett kan vi inte dela någon gång och får då 1 kvar och för fyra kan vi dela en gång och får också 1 kvar, för 2 och 5 får vi på samma sätt två kvar.

Det finns ett vanligt användningsområde för modolus vilket är att kolla om ett tal är jämnt eller inte, försök att lista ut hur du på ett smidigt sätt kan göra det.

TABELL

	Addera	Subtrahera	Dividera	Multiplicera	Golvdividera	Moduera
Heltal	Heltal	Heltal	Bråktal	Heltal	Heltal	Heltal
Bråktal	Bråktal	Bråktal	Bråktal	Bråktal	Bråktal	Bråktal

4.1 Stränga strängar

En sträng (string) är det som hanterar en godtycklig kombination av tecken och är användbara för att hantera till exempel text eller annan data. När man vill använda sig av en sträng markeras datan med antingen enkla (') eller dubbla (") citattecken innan och efter texten (strängen). Det finns en visst risk att man kan mista en sträng med ett heltal eller bråktal eftersom när man skriver ut en sträng då citattecknen tas bort när det skrivs ut.

```
>>> nummer = 2
>>> sträng = '2'
>>> print(nummer)
2
>>> print(sträng)
2
```

Om man är osäker på vilken datatyp det är man arbetar kan det leda till en del problem eftersom när det kommer till nummer (heltal och bråktal) kan man använda olika typer av operander på dem, vissa som också fungerar på strängar, men i detta fallet ger ett annat resultat.

```
>>> nummer = 2 + 2
>>> sträng = '2' + '2'
>>> print(nummer)
4
>>> print(sträng)
22
```

Om det vi ville göra var att räkna ut vad "2 plus 2" får vi precis rätt svar när vi använder oss av nummer men blir ganska förvånade av svaret 22. Det som har hänt när vi får nummer 22 är att vi har *konkatinerat* strängarna till en ny sträng. Vi har alltså slagit ihop de två strägarna till en ny sträng. Ett annat exempel som är lite tydligare i varför det inte fungerar är det nedan då vi försöker subtrahera en sträng från en annan och då får ett felmedelande som säger att "operanden - inte stöds när vi använder oss av två strängar".

```
>>> nummer = 2 - 1
>>> sträng = '2' - '1'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Men.. hur gör vi då för att veta vad för datatyp något är när vi har tappat bort oss i träsket av så många olika variablar så att vi knappt kommer ihåg hur vi ens hamnande i denna situationen? Jo, Python kan hjälpa dig med det med genom att vi använder oss av `type()` vilket talar om vilken *typ* något är. Men nu måste vi använda oss av två stycken parenteser, för vi måste ju också använda `print()` för att skriva ut informationen. Håll tungan rätt i mun nu för det gäller att hänga med och se till att ens parenteser matchar. Regeln är att vi måste ha lika många öppna parentser (som stängda parenteser), de innersta parenteserna hanteras först och sedan arbetar vi oss utåt. Säg att vi har variabeln `min_variabel` och vi vill veta vilken typ det är, då skriver vi alltså `type(min_variabel)`, efter att vi har tagit reda på vilken typ det är vill vi också skriva ut svart, då lägger vi nästa parentes runt den vi redan har och får `print(type(min_variabel))`. Först tar vi alltså reda på vilken typ variabeln är och sedan skriver vi ut den.

```
>>> nummer = 100
>>> sträng = '100'
>>> print(type(nummer))
<class 'int'>
>>> print(type(sträng))
<class 'str'>
```

Vi kan se i exemplet att variabeln `nummer` är av typen `int` vilket är en förkortning av integral eller heltal och att variabeln `sträng` är en `str` vilket är en förkortning av string eller på svenska sträng. För dig som är nyfiken är både `print()` och `type()` två olika *funktioner* vi kommer inte att gå in i detalj på vad de är just här men om du är nyfiken kan du gå till [REF TILL KAPITEL HÄR].

Något som liknar en *funktion* är en *metod* i den mån att de också har ett namn och parenteser efter sig. Skillnaden är att funktioner omsluter den data de arbetar med medans metoder gör något med en *klass*. Allt detta kommer att diskuteras i mer detalj [REF TILL LÄNK HÄR] men den enkla förklaringen är att en funktion ser ut så här `funktion(data)` och en method kan se ut så här `sträng.metod_namn()`. Om du ser en variabel eller något

liknande som har en punkt följt av något som ser ut som en funktion är det en metod.

I Python finns det så kallade *inbyggda methoder* det betyder att för olika datatyper finns det en del saker som så pass många männoskir gillar att göra så dessa är inbyggda i språket. För att lista alla dessa metoder kan vi använda en ny funktion (jag vet, väldigt förvirrande men ha lite tålamod) som heter `dir()` om vi omsluter en datatyp med `dir()` kommer python att skriva ut vad man kan göra med den datatypen. Om du skriver skript är det såklart viktigt att vi också använder `print()` men för att ge ett tips på en av fördelarna av att använda interaktiva Python är just när man är lite förvirrad och/eller har glömt exakt vilken method det var man letade efter för en datatyp. Då kan man snabbt öppna ett interaktivt fönster och omsluta den datatyp man vill ha hjälp med ett `dir()`. Nedan har vi gjort det för en sträng och för att få plats på sidan har vissa delar av datan blivit borttagen.

```
>>> dir('')
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
 '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', 'capitalize', 'casefold', 'center', 'count',
 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map',
 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',
 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper',
 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace',
 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
 'upper', 'zfill']
```

Oj! Vad mycket grejer man kan göra med en sträng, för att släcka din kunskapstörst kommer tar vi en snabbtitt på alla dessa som börjar och slutar med `__`. Dessa "talar om för oss" vad för typer av operander samt funktioner som vi kan använda på dem. Några exempel på detta är att '__add__' vilket betyder att vi kan *konkatinera* strängar, vi kan alltså slå ihop en sträng med en annan (som vi tidigare sett). Vi ser också att vi kan använda `dir()` på strängar eftersom det finns '__dir__'. Något som inte är lika uppenbart är kanske att '__str__' betyder att vi kan använda oss av `print()` funktionen på strängar.

Men alla dessa som inte börjar med två understräck, hur fungerar de? Dessa är våra metoder, alltså de som vi kan lägga efter strängen med en punkt och metodnamnet för att sedan göra något. Vissa av dessa metoder behöver också lite extra information för att de ska fungera, vi tar en titt på några exempel.

```
>>> 'detta är en sträng.'.capitalize()
'Detta är en sträng.'
>>> 'detta är en sträng.'.upper()
'DETTA ÄR EN STRÄNG.'
```

Den första gör att den första bokstaven blir en stor bokstav och den andra gör alla bokstäverna till stora bokstäver. Vi ska nu ta en titt på en metod som inte fungerar utan några *argument*, den behöver alltså mer information för att fungera, vi kan antigen ge den ett eller två argument, låt oss ta en titt på exemplet nedan för att få mer information.

```
>>> 'x'.center()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: center() takes at least 1 argument (0 given)
>>> 'x'.center(5)
'  x  '
>>> 'x'.center(5, '-')
'--x--'
>>> 'zyxyz'.center(5, '-')
'zyxyz'
```

När vi försöker använda oss av `center()` metoden förstår inte riktigt Python vad det är vi vill göra eftersom Python behöver argument, Python vet alltså inte vad vi vill göra om den inte får mer information. Men hur vet vi vilken information som Python behöver för att kunna göra det vi vill. I detta fallet hade vi tur som skrev rätt men vi kommer också ta en titt på hur man kan ta reda på denna informationen när man inte vet.

Metoden `center()` tar två argument och ett av dessa argument (det första) är ett måste, för att vi ska kunna använda metoden måste vi alltså ha detta argumentet. Detta argumentet är i detta fallet en integral som säger hur "bred" (många tecken) vår sträng ska minst vara, om vi har färre tecken än det kommer `center()` att centrera den texten med utfyllnad på båda sidorna. Om vi inte specifierar vad vi vill fylla ut med för tecken kommer

den att fylla ut med blanksteg. Vi ser att `.center(5)` gör precis detta, den centrerar och fyller ut strängen med två blanksteg på båda sidorna, när vi sedan säger att vi vill fylla ut med bindestreck istället gör den samma sak fast med bindestreck. Sannerligen är det ganska svårt att veta vad dessa två argument ska vara om man jobbar med en metod som man aldrig tidigare sett men det finns lite trick för att få mer information om vad man ska ha, vi kan i interaktiva Python skriva `help(str())` för att få upp mer information om strängar (`str()`) och dess metoder. Efter att du har läst klart det du vill läsa kan du gå ut från informationen genom att trucka `q` men först behöver vi gå ner till `center(...)` genom att trycka på mellanslag för att gå ner eller så kan vi använda oss av piltangeterna (upp och ner) för att navigera oss dit. Där kan vi läsa följande:

```
center(...)
    S.center(width[, fillchar]) -> str

    Return S centered in a string of length width. Padding is
    done using the specified fill character (default is a space)
```

Jaha, tänker du nu.. detta var ju lite kryptiskt och inte alls så hjälpsamt. Jag vet, det kan vara lite svårt att förstå vad allt betyder när man precis börjar så låt oss gå igenom denna kryptiska text och se om vi kan förstå den tillsammans. Det som står på första raden (`center(...)`) är metodnamnet, på nästa rad står det `S.center(width[, fillchar]) -> str` där i detta fallet står `S` för sträng som sedan följs av metoden och vilka argument som behövs. Vi ser att det första argumentet är bredden, alltså som vi såg ovan, hur bred ska strängen vara. Efter det står det `[, fillchar]` som har lite olika delar, låt oss börja med hakparenteserna `[` och `]`. Allt som står mellan dessa två är frivilliga argument, de behövs alltså inte och som vi såg ovan går det bra att använda funktionen med bara ett nummer. Kommatecknet används bara för att separera argumenten och `fillchar` är en förkortning (vilket vi sedan kan se i texten) för *fill character* eller det tecken vi vill fylla ut det extra utrymmet med. När det inte finns några fler argument som ska vara med markeras det med en stängd parentes. Slutligen har vi en pil `->` till `str` som kort och gott betyder att den typ av data vi kommer få tillbaka är av typen sträng. Slutligen följer det ett par rader med mer information om metoden. Vi gick precis igenom ganska mycket information men det är värt att gå igenom och förstå hur det fungerar för att på egen hand sedan kunna hitta information när man inte är säker på hur någonting fungerar.

En till sak som inte alls är dumt att veta är att om man ger fel argument till en metod kommer den klaga, vi har redan sett att om vi inte ger något

argument är Python mest förvirrad och vet inte riktigt vad hen ska göra mer än att säga att det borde finnas ett argument men jag vet inte riktigt vad. Om vi skulle chansa med vilket argument vi ger kommer dock Python att säga *bu eller bä*, hen säger *bu* när det är fel och också vad vi borde göra istället och en orm säger ju inte riktigt *bä* men om vu använder oss av rätt kommer det bara att fungera, nedan finns ett par exempel på detta:

```
>>> 'x'.center('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object cannot be interpreted as an integer
>>>
>>> 'x'.center(5.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: integer argument expected, got float
```

Vi ser att när vi vill gen Python en sträng eller ett bråktal säger Python i det första fallet att "Jag förväntar mig ett heltal, jag vet inte hur jag ska tolka en sträng som ett heltal" och i det andra fallet "Allt jag ville ha i julklapp var ett heltal, men allt jag fick var ett bråktal" (fri tolkning av översättningen).

4.2 Stycka datan

Något som är väldigt användbart när det kommer till både strängar och listor är att vi kan *stycka* (slice) dem, vi har inte ännu pratat om listor men principen i detta fallet är densamma, de styckas på samma sätt. Men först måste vi ha något att stycka och till det kan vi använda strängen "stycka inte mig" som vi tilldelar till variabeln `ska_styckas`. När vi styckar en sträng använder vi oss av hakparenterser och kolon efter vår sträng eller den variabeln som refererar till strängen. Formatet vi använder är `[start_index:slut_index:steg_storlek]` där vi kan strunta i att använda oss av alla tre. Om vi skippar startindex kommer vi börja med index 0 om vi skippar slutindex kommer det att vara sista index + 1 och om vi skippar stegstorleken kommer den att vara 1. Jaha, men vad betyder alla dessa nummer och *index* i detta sammanhanget då? Om vi börjar med vad ett *index*. Index är ett nummer som motsvarar ett tecken i strängen (när det kommer till strängar), det börjar med 0 och sedan räknar man uppåt, det som ofta skapar förvirring är att man börjar räkna från noll istället för ett som man

är van vid, om vi ska sätta detta i kontext så kan vi skriva ut våra index för den stränen vi har.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Tecken	s	t	y	c	k	a		i	n	t	e		m	i	g

Om vi börjar med att bara ändra startindex för att ändra strängen till "inte mig" måste vi tänka, på vilken index ska vi börja på. Eftersom *inte* börjar på index 7 verkar det som ett rimligt sälle att börja på. För att uttrycka detta i kod ändrar vi då `start_index` till 7 och vi måste ha med ett kolon för att visa att vi vill ha med resten av strängen från den punkten men vi kan helt hoppa över steget eftersom vi vill ha med alla tecken och bara hoppa ett steg i taget. Den kod som vi slutar med om vi arbetar med variabeln `ska_styckas` är då `ska_styckas[7:]` vilket ger oss precis det resultatet vi ville ha.

```
ska_styckas = "stycka inte mig"
>>> ska_styckas[7:]
'inte mig'
```

Ibland vill vi kanske behålla starten av en sträng men inte ha kvar slutet, då måste vi istället säga vad vårat `slut_index` är, det finns en viktig skillnad mellan start och slut som vi måste tänka på. Håll i dig nu men skillnaden är "och med", alltså när vi går från en start är det "från och med" som gäller men när vi går till är det "till" och inte "till och med" som gäller. Det är kanske bäst att illustrera detta med ett exempel, om vi vill skära bort allting efter index 10 måste vi alltså gå till index 11 för att vi ska få med index 10.

```
>>> ska_styckas[:11]
'stycka inte'
```

Nu när vi vet hur vi får med det från och med ett index till ett index så kan vi testa att försöka att kombinera dem för att få fram ordet *inte*. Vi vill alltså gå från och med index 7 till index 11.

```
>>> ska_styckas[7:11]
'inte'
```

Vi kan också använda stegen för att få fram varanat tecken istället för varje tecken, om vi för en kort stund ignorerar start och slut eftersom vi vill göra det från början till slut. Vi måste dock fortfarande ha med kolon så att Python vet att vi har gjort valet att strunta i att använda dessa värden.

```
>>> ska_styckas[::2]
'syk nemg'
```

Vi ser att vi lämnar de utrymmen där startindex och slutindex skulle ha varit blanka, det Python då vet är att den kommer gå från det första värdet till det sista värdet. Om vi inte hade haft dessa kolon hade python kunnat missförstå stegstorleken med start- eller slutindex.

Vi kan också använda oss av negativa index, om vi t.ex. vill ta bort de tre sista kan vi då skriva `[:-3]` och om vi vill ha med allting förutom de tre sista kan använda `[-3:]`, om vi vill gå igenom hela strängen men börja från slutet kan vi använda negativa steg och skriva `[::-1]` och om vi vill göra samma sak men bara få med varannat tecken kan vi skriva `[::-2]`, resultatet av dessa kan du se nedan.

```
>>> ska_styckas[:-3]
'stycka inte '
>>> ska_styckas[-3:]
'mig'
>>> ska_styckas[::-1]
'gim etni akcyts'
>>> ska_styckas[::-2]
'gmen kys'
```

4.3 Ändra, glöm

Något som vi inte har nämnt innan är vad som faktiskt händer när vi styckar och har oss, vi använder ju samma variabel helatiden men den verkar inte förändras. Anledningen till detta är att de metoder vi har använt inte tilldelar den data vi får ut till någon plats i minnet, utan de läser datan, förändrar den och slänger iväg den. Orginaldatan finns dock kvar, vår variabel `ska_styckas` (i fallet ovan) refererar fortfarande till samma data i minnet som vi inte har förändrat. Om man vill behålla den nya datan i minnet eller spara den är det därför viktigt att ha det i åtanke, om du förändrar data och vill ha kvar resultatet måste du tilldela det till en variabel, du kan desutom tilldela det till samma variabel. Att styckningen inte påverkar den datan som variabeln refererar till i minnet har dock sina fördelar, det gör t.ex. att vi kan stycka den flera gånger utan att något förändras. Kanske vill vi få ut ordet, *stycka* och *mig*, då kan vi göra det genom följande.

```
>>> ska_styckas[:6] + ska_styckas[12:]
'styckamig'
```

Att kombinera dem är alltså inte en så dum grej, vi kan här stycka samma variabel två gånger och sedan *konkatinera* dem till en ny sträng. Om vi vill kan vi se till att få med ett mellanrum från strängen men vi kan också konkatinera in den genom att "plusa" till ett mellanrum, det kan se ut på detta viset.

```
>>> ska_styckas[:6] + ' ' + ska_styckas[12:]
'stycka mig'
```

5 Sant, falskt, eller?

Att veta om något är sant eller falskt är inte alltid så lätt och även i programmeringsspråk är det inte alltid så lätt eftersom det gäller att hålla tungan rätt i mun. En sak som är säker är dock att när det kommer till programering är definitionen mer strikt för det än när det kommer till mänskliga påståenden. Att avgöra om något är sant eller falskt är också en av de mest *kraftfulla* egenskaperan när det kommer till programering. Det är detta som gör att vi kan ta beslut om vilken väg vi ska ta genom koden, men låt oss först undersöka lite vad som är sant eller falskt. En av de mest grundläggande jämförelserna man kan göra är att jämnföra om något är "samma sak" eller om t.ex. ett nummer är större eller mindre än ett annat nummer. När vi pratar om sant eller falskt i Python skrivs sant som `True` och falskt som `False`, det är viktigt att de börjar med stor bokstav, om vi inte använder det kommer Python inte förstå att vi menar sant och falskt.

Det enklaste är kanske för oss att öppna Python i det interaktiva läget och börjar göra lite tester för att se vad som händer. För pedagogikens skull kan det vara bra att börja med nummer och tal som vi alla vet svaren på, bara för att kontrollera att Python faktiskt beter sig som vi förväntar oss att det ska bete sig. Låt oss då börja med att se om 1 är mindre än 2, vilket borde vara sant. I terminalen skriver vi in `1 < 2` och i mitt system säger Python `True` så än så länge håller vi med varandra. Låt oss då testa om 1 är större än 2 genom att skriva `1 > 2`, återigen får jag rätt svar och Python säger `False`. Har du testat på ditt system? Får du samma svar?

Tidigare nämnde vi att = inte riktigt beter sig som man förväntar sig att det borde göra när man är van att använda det i mattematik. Om vi vill ha ett likamedtecken dubblar vi det istället, för mer är väl alltid bättre, för att jämnföra om något är "lika med något annat" använder vi oss alltså av `==`. Vi kan se om Python tycker att "1 är lika med 1" genom att skriva `1 == 1`. Om vi gör det borde Python säga till oss att det är `True`, vilken tur, Python verkar stämma överens med det vi har lärt oss i skolan. Men om vi

vill se om något inte är samma sak då, tänk om vi behöver försäkra oss att "1 inte är 2" hur gör vi då? Vi kan såklart använda samma som tidigare men då förvänta oss att Python säger `False` istället för `True` men vi kan också säga att "1 inte är lika med 2" genom att använda `!=`, nedan följer ett par exempel på dessa.

```
>>> 1 == 2
False
>>> 1 != 2
True
>>> 1 == 1
True
>>> 1 != 1
False
```

Notera hur `1 == 2` och `1 != 1` ger oss samma svar och att `1 != 2` och `1 == 1` också ger samma svar. Det gör det eftersom det är samma sätta tt uttrycka något på men vi ställer oss olika frågor. Det första exemplet ställer frågan "Är 1 och 2 samma sak?", det andra frågar om "1 och 1 inte är samma sak?" medans fråga 3 är "Visst är 1 inte samma sak som 2?" och slutligen ställer vi frågan, "Hur var det nu igen, är 1 samma sak som 1?". I alla de fallen ovan stämmer svaren överens med vår intuition men det gäller att hålla tungan rätt i mun, speciellt när man negerar frågan. När vi ställer frågan om "1 är 1" finns det bara en möjlighet för att det är *sant* men när vi ställer den omvända negativa frågan, t.ex. "1 är inte x" och x kan ha olika värden så är svaret *sant* för alla möjliga x förutom när x är 1. Fyra till vanliga operander är större än, mindre än, större eller lika med och mindre än eller lika med. Dessa finns i nedan tabell med exempel men efter att ha läst detta tror jag att du kan lista ut hur de fungerar.

	Operator	Exempel
Mindre än	$<$	
Större än	$>$	
Lika med	$==$	
Mindre eller lika med	$<=$	
Större eller lika med	$>=$	
Inte lika med	$!=$	

Men nu när vi vet om något är *sant* eller *falskt* vill vi kanske ta lite beslut utifrån vår nyfunna kunskap. Det finns många olika exempel man kan använda och jag uppmuntrar dig att skriva något liknande men låt oss

börja med att bygga en rutin eller struktur från när vi måste vattna våra växter. För att förenkla detta något och inte ta med hur soligt/regning eller andra variablar i vårt program så kommer vi endast att fokusera på tid. Hur vi vet om vi behöver vattna vår växt eller inte är genom att räkna dagar, om vi inte har vattnat på en vattnar vi, annars väntar vi till nästa dag. Vårat program är en simulation vilket gör att vi kommer snabba på processen och istället för att vänta kommer vi gå direkt till nästa dag. Vi gör detta för ett år (365 dagar), låt oss först ta en titt på koden och sedan analysera vad den gör.

```
# Vi säger att vi vattnat på dag 0
dagar_sen_vatten = 0
# Vi börjar på dag 1
dag_nummer = 1

# Begränsa till 365 dagar
while dag_nummer <= 365:
    # Skriv ut vilken dag det är idag
    print('Dag nummer {}'.format(dag_nummer))
    # Om vi inte vattnat, på en dag
    if dagar_sen_vatten == 1:
# "Vattna"
print('Inte vattnat på en dag, vattnar växter')
# Återställ dagar sedan vatten räknaren
dagar_sen_vatten = 0
    else:
# Vattna inte öka vår räknare
print('Idag behöver vi inte vattna')
dagar_sen_vatten += 1
    # Ny dag
    dag_nummer += 1
    # Skriv ut en tom rad för att göra det mer lättläst
    print()
```

Det finns kommentarer i koden men också en del koncept som vi inte har pratat om ännu, låt oss börja med `while` eller på svenska *medans*. Vilket vi använder för att uttrycka att vi vill göra något medans något är sant, rent konkret skriver vi `while` + "ett sanningstest". I fallet ovan är det vi testar om variabeln `dag_nummer` är mindre än eller lika med `365`, så länge detta är sant kommer vi att forsätta köra den biten av programmet som är

indenterad, varje gång vi har kört igenom den biten kod kommer vi att göra ett nytt test om detta stämmer eller inte. I vårat fall slutar koden köra när `dag_nummer` får värdet 366 (det är därför det inte skrivs ut).

Vi har redan tagit ett *beslut* i koden, att vi ska göra något *medans* vi klarar testet. De andra beslutet vi tar är att *om* (`if`) något är sant, gör A (vattna) annars (`else`) vänta. Precis som `while` behöver vi ge `if` ett test, testet i vår kod är *om* variabeln som håller koll på hur länge sedan vi vattna (`dagar_sen_vatten`) är `1` så vattnar vi, *annars* (`else`) väntar vi. Besultet att vänta behöver inget test eftersom detta är "slasken" eller det ikke-beslut vi gör om vi inte möter vårt första kriteritum.

En annan sak som är ny är att vi använder oss av `+=` vilket i vår kod betyder att vi skriver `dag_nummer = dag_nummer + 1`. Det är alltså ett enklare sätt att skriva det på.

6 Loopiga Loopar

När vi tittade på Primtalsexemplet i *syntax delen* [länk här] såg vi en typ av loop, nämligen *for-loopen* eller på svenska kan man kanske kalla den *för-loopen*. Den kallas just detta eftersom den går igenom ett värde i taget **för** varje definierat värde.

Det finns också en annan typ av loop, den så kallade *while-loopen* eller *medans-loopen*

7 Data har struktur

Data kan också ha struktur och det kan hanteras på olika sätt. I Python kan det vara lite difust om något är en struktur eller en typ och det är både en för- och nackdel. Andra språk kan vara mycket striktare med vad de anser vara vad men i Python används de på ett väldigt lika sätt. En väldigt vanlig, tidigare nämnd och användbar *datatyp* eller struktur är *listor* eller `list()` vilket används för att strukturera upp annan data. Du kan i princip strukturera upp vilken annan data som helst i en lista, till och med en annan lista. Listor implementerar en datorstruktur som kallas för `LIFO` eller *Last In First Out*, du kan dock få listor att bete sig på andra sätt och de är så pass flexibla att du kan använda dem på många olika sätt. Dock är LIFO-strukturen den som är effektivast implementerad i Python. Vi kommer att ta en mer djupgående titt på detta senare under *vanliga datastrukturer* [LÄNK HÄR], just nu är vi mest intresserade av vad vi kan göra med listor.

Tidigare har vi tittat på hur man sparar data till en variabel och detta är väldigt smidigt i vissa tillfällen, men tänk om vi vill jobba med massa data samtidigt. Det blir lite omständigt att ha en variabel för varje värde, speciellt om de är av liknande värde. Då kan det vara dags att använda en lista för att spara mycket data på ett och samma ställe med bara en variabel som *refererar* till datan.

8 Funktioner

Ibland vill man göra samma sak många gånger, och även om datorer har den fantastiska funktionen att *kopiera* och *klistra in* (Ctrl-c, ctrl-v) är det kanske lite dumt om vår kod är flera hundra rader lång bara för att vi vill göra något så många gånger i rad. Ibland vet vi inte ens hur många gånger vi vill göra något och då blir det svårt att klippa och klistra. Men oroa dig inte, det är just därför vi har funktioner, de är användbara delar av kod som vi kan använda hur många gånger vi vill. Säg till exempel att vi vill ha en funktion som översätter svenska till rövarspråk, om vi vill göra det på flera ställen i koden är det smidigt att vi har en funktion som vi smidigt kan använda det för, nedan följer ett exempel på hur en sådan kan se ut.

```
def rovarSprak( strang ):
    # En sträng som håller översättnigen
    oversattning = ''
    # En lista med alla vokaler
    vokaler = ['a', 'e', 'i', 'o', 'u', 'y', 'å', 'ä', 'ö']
    # En lista med tecken vi vill skippa
    skippa = [' ', ',', '.', '!', '?', '1', '2', '3', '4',
    '5', '6', '7', '8', '9']

    # Gå igenom varje tecken
    for tecken in strang:
# Om det inte är en vokal och inte ett tecken vi vill skilla
if tecken not in vokaler and tecken not in skippa:
    # Lägg till tecken + o + tecken i översättning
    oversattning += tecken + 'o' + tecken.lower()
else:
    # Lägg bara till tecken
    oversattning += tecken

    # Retunera översättnigen
```

```
    return oversattning

print(rovarSprak('Hej, vad heter du?'))
print(rovarSprak('Jag har inget namn.'))

test = 'rövarspråk'
test = rovarSprak(test)
print(test)
test = rovarSprak(test)
print(test)
```

```
Hohejoj, vovadod hohetoteror dodu?
Jojagog hoharor inongogetot nonamomnon.
rorövovarorsospoproråkok
rorororövovovovarorororsosososspopopoprorororåkokokok
```

Precis som utlovat kan vi använda samma *funktion* flera gånger för att göra samma sak om och om igen... I detta fallet hela fyra gånger. De två första gångerna översätter vi bara en sträng text och skriver ut resultatet men nästa gång skickar vi in en bit text, ändrar så att vår variabel refererar till det och skickar in det genom funktionen igen. Vi rövifierar språket två gånger, vilket är ganska häftigt men betydligt svårare att både säga och förstå. Om man vill hoppa över mellansteget att uppdatera variabeln kan man också skriva `rovarSprak(rovarSprak('text'))`, vi kan alltså skicka in resultatet vi får från första gången funktionen körs in i funktionen igen.

Kopiera koden koden och lägga in mer text som du vill översätta. Det finns många saker som koden inte hanterar, försök att hitta dem. Om du hittar något den inte hanterar, kan du lösa det?

Det finns kommentarer i koden som jag hoppsa är tydliga nog, om det är något som är oklart är det alltid bra att testa de olika delarna inviduellt för att förstå vad de gör.

8.1 Omfång eller sikte

Omfång, eller sikte, på engelska kallas det för *scope*. En viktig del av kodning är omfånget av koden, eller med andra ord, vilken del av koden påverkar andra delar. Den viktigaste delen av omfånget är att ibland är det bara lokalt, en del av koden påverkar bara sig själv men inte andra delar av koden. Detta återkopplar till hur variablar refererar till olika delar kod men

lått oss titta på ett exempel på varför det är viktigt.

```
x = 5

def ganger_tva( x ):
    x = x * 2
    return x

print(x)
y = ganger_tva(x)
print(y)
print(x)
```

Det vårat program skriver ut är följande:

```
5
10
5
```

Men vänta lite nu, `x` skrivs ut som 5 både innan vi använder funktionen som har koden `x = x * 2`, borde inte `x` då bli 10 eftersom vi säger att `x = 10` i början av koden? Det är just detta som är det som händer med ett lokalt omfång, den delen som utförs i *funktionen* är inte den samma som det utanför funktionen. Vi kan se det som att "`x` utanför funktionen är inte samma som `x` inne i funktionen", när vi använder `x` i funktionen är det alltså inte samma `x`. Utan vi har snarare en variabel som refererar till samma data och precis som när vi tidigare sa att `y = x` och sedan ändrade värdet på `y` utan att ändra värdet `x` när vi uppdaterade `y`. I funktionen har vi alltså ett *lokalt omfång*, det vi gör i funktionen påverkar alltså bara de *lokalavariablarna* eftersom de är separata variablar med samma namn. Men, detta verkar ju jättedumt? Är det inte bara förvirrande att variablar med samma namn pekar till olika saker? Ja, till en början är det helt klart förvirrande och det är därför det nämns men det finns också lite fördelar till att koden delas upp på detta sättet. Tänk dig till exempel att ni är flera stycken som arbetar med samma kod och ni av slumpen använder samma *variabelnamn* vilket kan leda till en massa problem. Det kan bli svårt att hitta felen i koden eftersom man inteförstår varför en variabel får ett visst värde man inte trodde det skulle ha. Om vi är säkra på att de variabelnamn vi har är *lokala* och inte kommer att ha oönskade konsekvenser på resten av koden är det lättare att felsöka och se till att man inte sitter och felsöker efter problem.

8.2 Rekursiva funktioner

Hänger du fortfarande med? Vilken tur, låt oss se om vi kan förvirra dig nu med ett av de delarna som många finner väldigt förvirrande *rekursiva funktioner*. Detta är funktioner som kallar sig själva för att göra en uträkning eller manipulerar data genom att skicka in data i "sig själv" eller en annan *instans* av funktionen. Låt oss först titta på en funktion, försök förstå vad som händer själv och sen går vi igenom det steg för steg.

```
def summan_av( nummer ):
    minus_ett = nummer-1
    if nummer == 0:
return 0
    else:
return nummer + summan_av(minus_ett)

print(summan_av(5))
print(summan_av(10))
```

Koden ovan skriver ut följande svar, om du inte riktigt hänger med i vad den gör rekommenderar jag att du testar att ändra *indatan* lite och se vad skillnaden i *utdatan* blir.

```
15
55
```

Okej, men vad gör koden då egentligen? Jo, den ger oss summan av alla nummer från noll till det nummer vi skickar in i funktionen så för nummret 5 gör den uträkningen $5 + 4 + 3 + 2 + 1 + 0 = 15$ och för 10 gör dem samma sak fast från 10 ner till 0.

För att bygga rekursiva funktioner finns det en viktig sak som man måste ha kolla på och en princip i hur python fungerar för att förstå vad som händer. Det viktiga är att man måste sätta ett *basfall* (basecase) för sin funktion när man designar den, detta är det som gör att den avslutas. Man kan säga att det är *kriteriet* för när ska vi sluta "gå in i ossjälva" och retunera svaret. En av de mer komplicerade sakerna är ofta att komma på precis vad detta kriteruimet är men det är viktigt att ha det eftersom annars kan det leda till att de aldrig kommer slutas köra. Om vi först tänker, vad vill vi att vår kod gör kan vi sedan definiera basfallet efter det? Det vi vill göra är att räkna ut summan för alla talen från 0 till och med det nummer vi skickar in. Vi vill inte ha med negativa nummer eftersom då skulle programmet aldrig sluta,

därför är det rimligt att vi slutar när nummret är noll. Mer formellt i koden ovan är vårat basfall `if nummer == 0`, när vi kommer till 0 retunerar vi bara det nummret. Okej, än så länge hänger vi med men vad gör vi fram tills vi har kommit till 0 då? Det första vi borde göra är att ta fram nummret som är det nästa att lägga till i summan eller det nummer vi nu vill addera minus 1. Vi sparar detta nummret i variabeln `minus_ett`. Om vi inte har kommit till noll skickar *retunerar* vi det nummret plus nummret minus ett som vi skickar in i samma funktion. **Det är viktigt att vi retunerar både basfallet och det som annars händer** eftersom om vi inte gör det kommer vi antigen inte att rettunera ett svar eller så kommer koden aldrig att sluta köra.

Nu är det en princip i hur programmet fungerar som man måste hålla reda på och det är något som kan vara väldigt förvirrande för många. De uträkningar som står till höger är det som sker innan vi *retunerar* svaret, vi retunerar alltså inte först. Detta innebär att vi först kommer räkna ut `nummer + summan_av(minus_ett)` innan vi retunerar och `summan_av(minus_ett)` innan vi adderar. Först, kommer vi alltså att skicka in `minus_ett` i funktionen `summan_av()` och det är viktigt att veta att varje gång vi gör detta skapas en *ny instans* av funktionen som *börjar om från början*. När vi gör det kommer vi alltså inte börja på samma ställe eftersom det är en ny *instans* av funktionen. Efter att den funktionen har kört klart kommer vi sedan att *addera* ihop `nummer` och vad som retunerades av funktionen `summan_av()` och slutligen *retunera* det vi har adderat. Jag vet, det är väldigt mycket information och jag hoppas att du fortfarande hänger med men för att vara på den säkra sidan kan vi ta en titt på bilden nedan ... beskriv bild och sedan titta på ett till exempel.

8.3 Ett klasiskt exempel, psudo kod

Fibinacisekvensen är en klassisk sekvensk och också ett klassiskt exempel för att visa på rekursion, i denna delen kommer vi titta på ett *naivt* exempel av hur man kan använda det. För även om rekursiva funktioner är imponerande är de inte alltid effektiva om man inte implementerar dem rätt. Efter att vi har tittat på vårat naiva exempel kommer vi ta en titt på varför det – imponerande som det är – inte är väldigt effektivt och hur vi kan förbättra. En bra utgångspunkt när man vill skriva ett skript är oftast att förstå eller kunna uttrycka i ord vad det är man vill göra. För att kunna uttrycka i ord vad vi vill göra måste vi först förstå problemet, så låt oss ta en titt på vad det är vi vill göra.

Fibinacisekvensen är en sekvens där vi får det nästföljande numret genom

att addera de två tidigare nummren i sekvensen. Men om vi inte har några nummer att starta med kommer vi aldrig att komma någonstans eftersom vi inte har några nummer att addera, därför brukar man ofta *initiera* eller starta sekvensen med att säga att de två första nummre är *noll* och *ett* eller ibland *ett* och *ett*. Eftersom det är en *sekvenks* kan vi tänka oss att varje del av sekvensen har ett *index*, och således att index 0 och 1 antingen har värderna *noll* och *ett* eller *ett* och *ett*. Vidare skulle du det *tredje* numret som har index 2 antingen bli *ett* eller *två* beroende på vad vi har för två startnummer. Vi kan säga att det tredje (index 2) fibinacinumret är 1 eller 2. För att göra det lättare för oss sätter vi nu en standard vilket är att i vårat fall börjar sekvensen med 0 och sedan 1. Detta gör att index 0 är 0, index 1 är 1 och index 2 är 1, osv..

Det vi vill göra är att räkna ut är ett nummer för det index vi ger funktionen, eller *fibi(index) -> Nummret på index i fibinacisekvensen.* Om vi ger funktionen ett index vill vi att vi retunerar motsvarande nummer för det indexet i sekvensen. Det vi redan vet är vad svaret för index 0 och 1 är vilket gör att detta är en bra utgångspunkt för att sätta vårat basfall. För att undvika negativa nummer kan det även vara bra att säga att alla nummer under noll kommer att räknas som noll. Innan vi uttrycker det i kod kan det vara bra att skriva det som *pseudokod* eller en abstrakt representation av kod. Pseudokod är något som ligger närmare mänskligt språk och programeringskod, tanken är att det beskriver en metod eller ett sätt för att lösa något utan att vara bundet till ett språk. Du kan till viss del se det som ett sätt att uttrycka ett koncept och hur det implementeras är upp till den inviduella programmeraren eftersom olika språk skjiler sig åt. Nedan följer ett väldigt basalt exempel på pseudokod vilket beskriver det vi kommit fram till än så länge i vår kod.

```
fibinummer_för_index( fibiindex )
    Om index är mindre än noll, retunera noll
    Om index är lika med ett, retunera ett
```

Ofta kan pseudokod vara mer formell än just det vi uttrycker i exemplet ovan då detta ligger lite mer mänskligt språk än vad man ofta kan stöta på. Men i vårat fall använder vi det mest för att uttrycka det vi vill göra och i vilka steg vi vill göra det, vi använder en väldigt hög abstraktion ifrån programerningsspråk för att göra detta.

Okej, så genom att titta på hur vår "kod" ser ut än så länge borde det fungera för index 0 och index 1, sakta men säkert går det frammåt. Men vad gör vi då om vi får ett index som är större än 1? Jo, det är ju bara att kolla

på de två tidigare index för att se vad svaret blir.. men hmmmmmmm... för index 2 går detta alldeles utmärkt men vad händer med index 3? Då behöver vi summera index 1 och index 2, index 1 går bra men vi vet inte vad index 2 är men om vi hade skickat in index två i funktionen kunde vi få svaret för index 2, om vi sedan skulle kolla index 4 behöver vi skicka in båda dessa i funktionen igen. För att det ska fungera med index högre än 3 måste vi alltså räkna ut svaret från vår funktion igen för att det ska fungera. Om vi lägger till det i koden kommer den då att se ut som följande.

```
fibinummer_för_index( fibiindex )
    Om index är mindre än noll, retunera noll
    Om index är lika med ett, retunera ett
    Annars, räkna ut fibinicinummret för index-2 och addera med fibinicinummret för in
```

Okej, jag tror att det verkar rätt, men för att vara säker kan vi gå igenom koden i huvudet eller på papper för några nummer bara för att vara säkra på att det gör det vi vill. Om vi börjar med index 0 kommer vi få 0, för index 1 kommer vi få 1. Vad händer då om vi skickar in 2? Då kommer vi räkna ut fibinaci för index 0 (2-2) och index 1 (2-1), vad svaret till dessa är det vet vi ju redan, det är 0 och 1, vad bra, då behöver vi bara addera dem vilket ger oss 1. Det verkar stämma överens med vårat antagande men för att vara på den säkra sidan kan det vara bra att ta en titt på ett till exempel, vad händer om vi vill veta nummret på index 3? Precis som innan måste vi då vet svaret för index 1 (3-1) och index 2 (3-1), vilket vi återigen vet att vi kan få genom tidigare exempel, om vi då adderar de svaren vilket är 1 och ett får vi 2. För att få lite översikt kan vi sätta det i en tabell som den nedan.

Index	0	1	2	3
Nummer	0	1	1	2

Vi vet att ett nummer borde vara lika med summan av de två tidigare nummren, stämmer det för denna tabellen? Ah men visst ser det ut att stämma, vad bra. Efter att vi nu har formulerat frågan, funderat ut ett svar för att sedan uttrycka det i psudokod som vi sedan testat och fungerar för de tester vi körde är det dags att implementera vår lösning. Gör ett försök på att implementera det själv först innan du tittar på koden nedan som innehåller en lösning. Om du räknar ut det nummer som du borde få för index 10 är svaret 55, får du det?

```
def fibi( index_nr ):
    if index_nr < 0:
```

```
return 0
    elif index_nr == 1:
return 1
    return fibi(index_nr-2) + fibi(index_nr-1)
```

Det finns dock lite problematik med denna lösningen och det är i den koden vi har skrivit görs många saker flera gånger, desto högre index vi vill räkna ut desto fler beräkningar görs flera gånger och detta är ganska suboptimalt, helst vill man ju bara göra varje uträkning en gång. Fast, vadå saker görs flera gånger? Om vi tänker att vi vill räkna ut index 4 då kommer vi behöva att köra funktionen två gånger, för index 2 och index 3. Låt oss följa dem en i taget och se när de tar slut.

Funktion	**Beräkning 1**	**Beräkning 2**
0-fibi(4)	1-fibi(2)	2-fibi(3)
1-fibi(2)	1-fibi(0)	1-fibi(1)
2-fibi(3)	2-fibi(1)	2-fibi(2)
2-fibi(2)	2-fibi(0)	2-fibi(1)

Om vi tar en titt på tabellen med de beräkningar som vi behöver göra är det kanske inte först uppenbart att vi gör samma sak två gånger, så låt mig förklara hur den fungerar. Tanken är att vi kan följa de två olika uträkningarna osm görs genom att titta på prefixen vilket är 0-, 1- eller 2- där 0- står för det första vi skickar in 1- för vad som kommer fårn den första beräkningen (index-2) och 2- för den andra beräkningen (index-1). Den första raden ger oss två nya beräkningar vi behöver göra och på den andra raden ser vi hur den första av de beräkningar ser ut, eftersom det vi beräknar där är 0 och 1 är vi klara. På rad 3 är den andra uträkningen vi behöver göra vilket är fibi för 1 och 2, vi vet redan att den för 1 blir ett men vi behöver då gå vidare med den för 2, men vänta lite, denna har vi ju redan gjort. Det är just detta som är problematiken eftersom med den koden vi nu har behöver vi göra den igen, antalet beräkningar som vi kommer behöva göra mer än en gång kommer att öka desto högre index vi har.

! Testa att räkna hur många beräkningar vi behöver göra två gånger om vi börjar med index 5.

I det exemplet vi precis tittat på ser det ju inte ut att vara så farligt, vi räknar faktiskt bara ut en sak två gånger, inte optimalt men inte förödande heller. Helt korrekt, när det kommer till små nummer spelar det oftast inte så stor roll men beroende på hur snabbt antalet beräkningar ökar kan det leda till problem. Testa att räkna ut `fibi(100)`, ett tips är att du kan avsluta

något från att köra genom att trycka `Ctrl-z` (Control tangenten + z) eller `Ctrl-c` (Control tangenten + c). ## SKRIV VIDARE MED ANTALET BERÄKNINGAR ##

Det som händer är att det kommer ta väldigt lång tid att göra alla beräkningar, antagligen kommer du inte kunna räkna ut det med den datorn du använder förän om XX antal år. Det känns som att vi behöver en bättre metod för att göra detta, en teknik som vanligen används för detta är *dynamisk programering*. Kort och gott kan man söga att det dynamiska är att vi använder oss av ett minne för att hålla reda på tidigare steg och inte behöva beräkna saker två gånger.

När det kommer till programmering är detta ett väldigt vanligt koncept, att vi använder oss av *datorminne* för att minska antalet processer. Vanligtvis tänker man i *utrymme* mot *körtid* och utifrån vad man vill göra måste man alltid göra en avvägning. Om vi kanske tänker oss att vi gör en internetsökning är det kanske viktigare att detta sker snabbt snarare än att alla beräkningar görs i realtid, detta leder till att sökmotorer oftast redan har *indexerat* hemsidor sen tidigare för att det ska gå snabbare. Om vi spelar ett spel är det nog också viktigt att beräkningarna går ganska snabbt så att det inte är jättesegt att spela det. Nu förtiden är minne oftast inte ett problem men tidigare när minnet var mer begränsat var det viktiga avvägningar att göra. Ett sätt att exemplifiera detta är genom att titta på ett par olika sorteringsmetoder för att sortera data. Här kan vi använda oss av mer minne eller i princip inget minnet alls. Okej, men om vi inte vill använda oss av mer minne än vad vi behöver för att spara datan, hur gör vi då?

Bubblesort är en välkänd algoritm som illustrerar hur vi inte behöver använda oss av något extra minne för att sortera en lista. Den fungerar på ett av de enklaste sätten man kan tänka sig, den byter ut värden i listan, två i taget i "en bubbla" tills vi är säkra på att listan är sorterad. Låt oss utgå ifrån att vi vill sortera en lista i stigande ordning (lägsta numret till högsta), får lista är N element lång. Vi börjar från index 0 och jämför det med index 1, om index 0 är större än index 1, byter vi värde. Vi forsätter göra detta för index 1 och 2, 2 och 3, osv.. tills vi nått index N. Nu vet vi att på index N kommer vi att har det största värdet eftersom vi har flyttat med det största värdet till den positionen (även om det låg på index 0 tidigare). Vi forsätter att göra detta ## INTE KORREKT ## N-1 gånger, eftersom vi nu vet att index N har det största värdet behöver vi bara gå upp till index N-1 nästa gång och gången efter det N-2, osv.. Det kanske är bäst att uttrycka detta i psuedokod för att förstå hur det fungerar.

```
sortera_lista( lista )
```

```
Gör N gånger
    slut_N <- N-1 till 1
    Gör slut_N gånger
bubbel_index_vänster <- 0 till slut_N-1
bubbel_index_höger   <- 1 till slut_N
Om värdet på bubbel_index_vänster är större än höger,
Byt värde på höger och vänster index
```

```
def bubble_sort( lista ):
    for i in range(len(lista), 0, -1):
for j in range(1, i):
    if lista[j-1] > lista[j]:
lista[j-1], lista[j] = lista[j], lista[j-1]
    return lista
```

Vi kan se att när vi använder oss av *bubble sort* använder vi knappt något minne alls om man bortser från att vi håller reda på vilket index i listan vi arbetar med. Det enda vi gör att att byter plats på värden i listan.

8.4 Körtid

8.5 Dynamisk programering

Dynamisk programmering handlar om att man använder sig av minnet för att slippa lösa samma delproblem flera gånger. Detta innebär i princip att vi byter ut minne mot snabbhet, vi använder oss av mer minne för att programmet ska gå snabbaare att köra. Vi har redan sett ett exempel där vi löser samma problem fleera gånger. När vi `rankade fibiniacisekvensen` räknade vi ut samma delproblem flera gånger eftersom vi inte sparade resultaten. Vi ska nu ta en titt på hur vi kan lösa problemet snabbare genom att använda oss av dynamisk programering.

8.5.1 Fibinacci sekvensen

Det vi vill göra är att spara den uträkningen för ett fibinatccinummer varje gång som vi har räknat ut det. Ett bra sätt att spara det på är att använda sig av ett `dict()` eftersom om vi använder det kan vi slå upp det tidigare nummret varje gång vi behöver det. Om vi inte har nummret måste vi dock beräkna det. Låt oss först uttrycka det u pseudokod.

```
minnet <- skapa minnet {0 : 0, 1 : 1}
```

```
def hitta_fibi ( fibi_index ):
Finns svaret i minnet?
  retunera minnet[fibi_index]
Annars om det inte finns i minnet
     minnet[fibi_index] <- hitta_fibi( fibi_index-2 ) + hitta_fibi( fibi_index-1 )
retunera minnet[fibi_index]
```

Några saker att notera som sker är att vi har vårat minne utanför funktionen, detta har med omfånget att göra. Om vi skulle ha vårat minne i funktionen skulle vi skapa ett nytt minne och slänga bort det varje gång vi körde funktionen, för att undvika detta sparar vi allting utanför funktionen. Låt oss då ta en titt hur vi kan formulera detta i python. En annan sak vi gör är att initiera minnet med de två första givna nummren, detta gör att vi kan förenkla koden en del.

```
# Initiera minnet som ett dictionary med de två första nummren
minnet = {0 : 0, 1 : 1}

def hitta_fibi( fibi_index ):
    if minnet.get(fibi_index, -1) != -1:
return minnet[fibi_index]
    minnet[fibi_index] = hitta_fibi( fibi_index-2) + hitta_fibi( fibi_index-1 )
    return minnet[fibi_index]
```

9 Praktiskt Python

Teori i all sin ära men ibland vill man faktiskt göra ett program som fungerar och gör något vettigt. Tanken med denna delen är att introducera ännu några koncept som inte blev täckta i den första delen av boken men på ett mer praktiskt sätt. Ha i åtanke att det finns ingen kod som fungerar universiellt för alla problemen och det är en extrem förenkling att säga att alla exemplen kommer vara användbara för dig eller ens applicerbara. Förhoppningsvis kommer du att få en viss känsla och insikt på hur du kan använda python för att lösa det problemet du har eller få en idé på att det går att lösa något du inte ens trodde var ett problem.

9.1 Inbyggda funktioner

9.2 Räkna ord, leta mönster

Ett ganska intressant problem är att räkna ord och unika ord i en text för det finns vissa problem med det som man inte vanligtvis tänker på. Ett par av dessa är *vad är ett ord* och *hur delar man upp ett ord*. Låt oss börja med att säga att ett ord är en sammanhängade bit text som består av tecken och åtskiljs av mellanrum. Utifrån denna definitionen låter det rimligt att behandla texten som en *sträng* eftersom det är text vi har med att göra. Tidigare har vi sett att vi kan använda oss av `split()` för att dela upp text och vi kan ange vad vi vill dela den med men om vi lämnar den tom kommer den att använda sig av mellanrum. Låt oss då titta på ett par exempel och se hur det går.

```
>>> text = "Detta är en exempel text. Den är skriven år 2017, min emejl är test@test.se
 >>> text.split()
['Detta', 'är', 'en', 'exempel', 'text.', 'Den', 'är', 'skriven', 'år', '2017,', 'min',
```

Det finns ett problem med detta exemplet och ett som man stöter på desto mer variearad och komplicerad text man har, kan du se det? De olika skijetecknen orsakar en hel del problem. Vi vill nog att `2017,= ska vara =2017` och =,= och inte heller ha med den sista punkten i emejlen. Detta kallas att tokanisera texten, eller att dela uppden i dess beståndsdelar. Varje ord eller tecken som punkter och komman ses som en token. Låt oss inte oroa oss över det för tillfället men det är bra att ha i åtanke lite senare. Okej, men om vi ska ta en titt på hur vanligt ett ord är, hur ska vi då gå till väga. En datastruktur som vi inte ännu diskuterat är `uppslagsverk` eller `dictionaries` skrivna som `dict()` i Python.

9.2.1 Slå upp, slå ner

Inom andra programeringsspråk kallas de för *hashtabeller* [LÄNK TILL INFO OM DETTA] och de fungerar som *hashtabeller* i andra språk. Men hur fungerar de då? Jo, precis som ett uppslagsverk (ungefär). Du väljer något du vill använda som referens, det måste vara något som du inte kan ändra, ganska fast i form, som är din referens. Detta kallas oftast för *nykel* eller *key* och det som det refererar till kan vara något förändringsbart vilket kallas för *värdet* eller *value*. Om man ska översätta denna abstrakta beskrivning betyder det i princip att du inte kan använda en lista till nykel men de flesta andra datatyper går bra att använda.

Hur skapar vi då ett uppslagsverk? Jo, vi skapar en variabel som refererar till ett `dict()` för att skapa det först. Det finns två sätt att vanligtvis göra detta på antingen skriver vi `variabel_namn = dict()` eller `variabel_namn = {}` där måsvingarna ({}) betyder att det är ett uppslagsverk. Dessa är dock tomma när vi skapar dem så vi måste fylla dem med data först. En sak vi kan använda dem för är att räkna något eller för att ha något som refererar till något annat. Låt oss börja med att räkna mobiltillverkare, kanske gör vi en undersökning av vilken tillvärkare av mobiler våra kompisar har. Om vi skapar den tom kan vi räkna först och sedan lägga till rätt data, låt oss ta en titt på hur man skulle göra detta.

```
>>> mobil = dict()
>>> mobil['samsung'] = 5
>>> mobil['nokia'] = 1
>>> mobil['iphone'] = 4
mobil
{'nokia': 1, 'iphone': 4, 'samsung': 5}
```

Vi lägger till strängar som våra nycklar vilket refererar till våra värden som är antalet personer med en mobil från den tillverkaren. Men vänta, vi glömde ju att ränka med Klas som också har en iphone, vi måste lägga till ett. Vi kan då uppdatera värdet genom att skriva:

```
mobil['iphone'] = mobil['itelefon'] + 1
>>> mobil['iphone'] = mobil['itelefon'] + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'itelefon'
```

Vänta, nu gick något fel.. den säger att vi får ett `KeyError` för ='itelefon'=, det verkar alltså vara ett problem med vår nyckel.. hmmmmm, ja juste! Vi använde oss ju inte av nyckeln *itelefon* utan *iphone* det är klart den inte kan hitta en itelefon om den inte redan finns. Låt oss försöka igen.

```
>>> mobil
{'nokia': 1, 'iphone': 4, 'samsung': 5}
>>> mobil['iphone'] = mobil['iphone'] + 1
>>> mobil
{'nokia': 1, 'iphone': 5, 'samsung': 5}
```

Nu fungerar det, vi har nu fem stycken som har en iphone istället för fyra. Men vad skedde då ovan. Jo, som vi minns så gör Python det till höger först, den hämtar alltså värdet för hur många som har en iphone (fyra stycken) plussar på ett (totalt fem) och ändrar sedan referensen för iphone till fem istället för fyra.

Nykeln skulle t.ex. kunna vara en sträng och värdet skulle kunna vara ett heltal.

9.3 hitta primtal

9.4 Göm en bok i en bild

9.5 Sno en hemsida

10 Saker att ha med

- En sektion om att ha verktygen för att kunna göra det.. Till exempel hur vi går från en idé till att definiera det i pseudokod och sedan skriva koden. Men för att skriva koden behöver vi rätt verktyg, vi behöver kunna koda för att göra det vi vill.
- Ta upp impotera ocn annat (kanske under andra delen)

10.1 Två delar av boken

En ny tanke om strukturen.. att först introduceeras koncepten så bra som möjligt (kanske 50 sidor?) och jag behöver inte ta upp objektorienterat och sedan spenderas andra halvan åt "real world problems" .. Eller hitta övningar där man kan applicera det man har lärt sig som refererar tillbaka till de grundläggade kapitlen men också introducerar en del nya koncept som hur moduler impoteras och dersa omfång (jämnf med funktioner)..

Låter detta som en bra ide?

10.2 Mål/frågeställningar

11 Länkar

11.1 Interna länar

11.2 Externa Länkar

Snake Case