KODA MATTE MED PYTHON

Taifun Alishenas

TECH pages.

Titel: Koda matte med Python
ISBN: 978-9-197-42045-7

Författare: Taifun Alishenas
taifun@kodamatte.se

Omslag: Simon Bernström

Tel: 08 - 792 36 28
www.techpages.se

Tryckeri: Eprint, Stockholm
September 2018

KODA MATTE MED PYTHON

Programmering i matematik
En handbok för lärare och elever

Med övningar och lösningar

Koda direkt i vår mobila pythonmiljö
Ladda ned appen *Mattekollen* här:

En bok av
Taifun Alishenas

TECH pages.

Innehållsförteckning

Om boken

- *Koda matte med Python* är varken en lärobok i programmering eller i Python, utan en handledning om hur programmering kan integreras i skolmatematiken utan att göra mattekursen till en programmeringskurs.

- Boken presenterar ett koncept om vilka delar av matematiken som kan vara lämpliga för programmering och på vilket sätt kopplingen till läroplanen kan realiseras – ett komplement till den ordinarie matteboken i det dagliga arbetet både för lärare och elever.

- Boken kan användas på både högstadie- och gymnasienivå av lärare som inte haft erfarenhet av programmering eller inte kombinerat programmering med matte. Den kan även användas som kursmaterial av elever. Strukturen följer Skolverkets ämnesplan för matematik.

- *Koda matte med Python* börjar med att behandla första kapitlet av kursen Matematik 1 (a, b, c). Vi bortser från skillnaderna mellan a, b, c-spåren. Ytterligare kapitel och kurser är planerade.

- Bokens programexempel och övningar förutsätter inte någon speciell miljö för programmering. De kan utföras i vilken pythonmiljö som helst. Det är upp till användaren att välja sin favorit utvecklingsmiljö.

- Samtidigt ger boken instruktioner till nybörjare i programmering för nedladdning, installation och användning av Pythons senaste version (sid 14) samt Pythons egna integrerade utvecklingsmiljö (sid 17) – båda är freeware. En annan gratis utvecklingsmiljö presenteras i bokens appendix.

- Som ett digitalt komplement till boken finns i appen *Mattekollen* möjligheten att använda en pythonmiljö. Där kan elever testa bokens programexempel och övningar utan att behöva installera Python (sid 156). Mattekollen finns som webbapp samt till Android och iOS.

Vad boken handlar om

Hösten 2018 ska Skolverkets nya läroplaner för matematik börja gälla i skolan. Enligt dem ska programmering ingå i matematikundervisningen. Men hur ska det praktiskt gå till? Å ena sidan oroar sig många för hur tiden ska räcka till för båda tunga ämnen. En oro som är högst relevant med tanke på att de flesta praktiserande mattelärare varken är utbildade i eller har erfarenhet av programmering. Å andra sidan är det berättigat att låta skolmatematiken äntligen dra nytta av informationsteknologin som matematiker en gång i historien lade grunden till. Det är inte bara dags utan sedan länge angeläget att *algoritmisera* skolmatematiken för att ge eleverna en bättre förståelse för matematiken. Detta kan uppnås genom att strukturera uppgifternas lösning till *algoritmer* – embryon till datorprogram. Att lösa den här svårlösta ekvationen vill denna bok bidra till.

Efter flera års erfarenhet som forskare i numerisk analys, författare av läromedel både i matematik och programmering samt mattelärare på gymnasiet känner jag mig direkt utmanad att ta mig an denna uppgift. Frågan är på vilket sätt programmering kan vävas in i det nuvarande upplägget av mattekurserna, så att inte bara tiden räcker till utan även eleverna i slutet av läsåret når Skolverkets kunskapskrav i sina respektive kurser. Denna bok besvarar frågan inte definitivt utan är snarare ett *koncept* om hur programmering kan integreras i matematikundervisningen utan att göra mattekursen till en programmeringskurs. Den innehåller material på *delar* av skolans mattekurser som med fördel kan behandlas med programmering. Ytterligare delar kommer att kompletteras i bokens framtida upplagor. Vilka delar som i praktiken väljs i undervisningen överlåts den enskilde praktiserande läraren – med hänsyn till klassens aktuella tillstånd vad gäller intresse, inriktning, matematiska förkunskaper, nivå och önskemål.

Som miljö för programmering föreslår många program av typ *Excel*, *GeoGebra, Octave, Matlab* etc. eller något webbspråk. Dessa verktyg är specifika program som andra skrivit och som vi endast kan lära oss att *använda*. Målet borde dock vara att lära eleverna att *programmera från grunden*, att ställa upp *algoritmer* som beskriver metoder för matematisk problemlösning och koda algoritmer. Verktygen ovan är inga optimala lösningar för detta ändamål. Därför går boken ett steg längre och väljer ett *universellt* programmeringsspråk: Python. Även detta är endast ett medel. Målet är att lära sig *tankesättet* och *tekniken* att programmera, oberoende av språk, så att även Python kan ersättas av andra språk.

Boken förutsätter inga förkunskaper i programmering. Den kan användas av lärare som inte haft erfarenhet av programmering eller inte tillämpat programmering på matematiken. Den kan även användas av elever som kursmaterial. Speciellt övningsdelen med lösningar är i första hand avsedd för att användas av elever i klassrum och/eller hemma.

Ett komplement till bokens övningsdel är *Mattekollen*, en app som ständigt utvecklas både som digitalt hjälpmedel till den här boken och som stöd för skolmatematikens andra kurser. Läs mer om detta på sid 156.

Koda matte med Python utvecklas och uppdateras permanent. Jag vill gärna ta del av läsarnas intryck under programmeringshösten 2018 och utveckla nästa upplaga i en pågående dialog med läsarna. Har du lust att tycka till om boken, bolla idéer, föreslå nya upplägg eller medverka på något annat sätt? Hör gärna av dig till taifun@kodamatte.se så kan vi prata mer.

Kan programmering främja matematikens lärande?

Kan programmering verkligen bidra till att elever får en bättre förståelse för matematiken eller är den bara en extra belastning för matematikundervisningen?

För att diskutera frågan borde inledningsvis frågan besvaras: vad egentligen är programmering? Här några kortfattade förslag:

- Programmering är problemlösning med hjälp av datorn.
- Program är algoritm plus data.
  (*Niklaus Wirth, skaparen av Pascal på 60-talet*)
- Program är modeller av verkligheten.

Om målet är problemlösning måste lösningen inte bara hittas utan också formuleras. Ett tillvägagångssätt som exakt och entydigt *beskriver* hur man löser ett problem, kallas för *algoritm*. Väljer man programkod för att formulera algoritmen, har man ett *datorprogram*. Läggs det till vissa *data* (t.ex. input) har man kommit till Niklaus Wirths definition.

Programmering bidrar till matematikens lärande, eftersom den formulerar matteproblemens lösningar som algoritmer. Lösningarna *måste* formuleras som algoritmer för att datorn ska ”förstå” dem. Lyckas man med det har man själv förstått dem bättre. Programmering lyfter en till samma nivå som en lärare då datorn blir ens elev. Dessutom uppmuntrar den eleverna till exakthet och logiskt tänkande. Båda behövs i matematiken, även om inte bara. Inte minst är algoritmer generella och bidrar till att eleverna lär sig att *abstrahera*, vilket är en förutsättning för en djupare förståelse av matematiken.

Abstraktion

Abstraktion är ett grundläggande koncept i allt tänkande, även i matematiken. Hela matematiken består egentligen av en rad abstraktioner på olika nivåer. Därför är den till sin karaktär en abstrakt vetenskap. Detta för att kunna:

- vara generell,
- bortse från de oväsentliga skillnaderna mellan de många konkreta fal len och koncentrera sig på deras gemensamma struktur,
- använda matematiken inte bara på några få enstaka uppgifter utan på så många konkreta problem av liknande typ som möjligt.

I denna bemärkelse är matematiken ganska lik filosofin.

> ”Filosofins komplexitet ligger inte i dess innehåll,
> utan i våra hjärnors förvrängda knutar.”
>
> *Ludwig Wittgenstein: Filosofiska anmärkningar (1930)*

Samma sak skulle man kunna säga om matematiken. Låt oss lösa knutarna med hjälp av programmering!

Om programmering

Programmering är i allra högsta grad ett praktiskt ämne. Ingen kan lära sig programmering genom att bara läsa böcker. För att själv skriva och testa program behöver man ett antal verktyg – som i sin tur är programvaror. Det viktigaste av dem är en *kompilator* resp. *interpretator*. Medan en kompilator *översätter* källkod till maskinkod och lagrar den i en exekverbar fil, *tolkar* en interpretator källkoden direkt. Människan kan lära sig, förstå och skriva källkod, men inte maskinkod. Datorn arbetar tvärtom. Jobbet däremellan görs av kompilatorn/interpretatorn. Det finns hundratals språk för källkoden. I denna bok har vi bestämt oss för programmeringsspråket Python, men språket är bara ett medel av underordnad betydelse. Målet är att lära sig *tankesättet* och *tekniken* att programmera, oberoende av språk. Har man en gång förstått de grundläggande koncept som är gemensamma för alla språk, blir det närmast en teknikalitet att på egen hand lära sig ett nytt språk.

Pythons programmeringsmiljö

Python interpretatorn

Python skapades år 1989 av Guido van Rossum, en forskare på *National Research Institute for Mathematics and Computer Science* i Amsterdam. Det är ett *interpreterande* och *universellt* programmeringsspråk som är lämpligt för alla tillämpningar. Python kan enkelt och gratis installeras på alla plattformar utan att man behöver bry sig om licenser. Koden är nästan självbeskrivande, ligger nära pseudokod och återspeglar på ett intuitivt sätt algoritmers struktur.

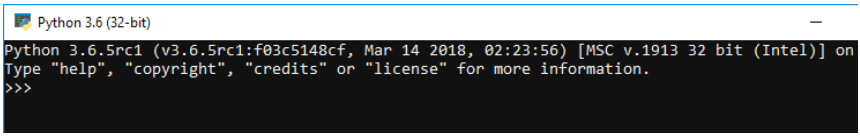
En *interpretator* är ett program som tolkar källkod till maskinkod och skickar maskinkoden till datorns processor som exekverar den direkt utan att lagra den på hårddisken. Ibland kommer vi att använda *Python* när vi menar Python interpretatorn.

Installation av Python

1. Gå till Pythons officiella webbsida för nedladdning:
   www.python.org/downloads
   Klicka på Download Python x.x.x
   x.x.x står för versionsnumret. Installationsfilen *.exe laddas ner. Spara den och (dubbel)klicka på den.

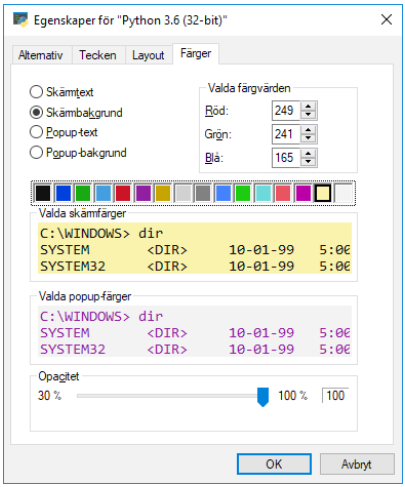
2. Dialogrutan Install Python... dyker upp.
   Viktigt! Innan du går vidare bocka först för rutan längst ner:
   Add Python x.x to PATH.
   Klicka sedan på Install Now. Vänta ett tag tills du ser Setup was successful. Stäng dialogrutan med Close.

Klicka på den lilla pythonikonen från datorns Start-knapp. Ett svart konsolfönster öppnas där markören står och blinkar efter prompten `>>>`. Prompten är symbolen för Pythons interpretator och visar att Python är beredd att ta emot kod från användaren:



Bakgrundsfärg

Om du inte gillar konsolfönstrets svarta bakgrundsfärg klicka på den lilla python-ikonen i fönstrets övre vänstra hörn och välj Egenskaper.



Markera radioknappen Skärmtext i dialogrutan som dyker upp (till höger). Välj svart med färgvärdena Röd 0, Grön 0, Blå 0. Markera Skärmbakgrund och välj färgvärdena Röd 249, Grön 241, Blå 165. Klicka på OK. Självklart kan du även välja valfri färgkombination efter din egen smak. Nu ser konsolfönstret lite färggladare ut, eller hur?

Interactive mode

Eftersom Python är interpreterande kan vi använda *Interactive mode*, vilket innebär att vi kan mata in pythonkod i interpretatorn och få svar direkt. På köpet kan man enkelt och snabbt testa vilken pythonkod som helst, innan man tar över den till något större program. Öppna Python interpretatorn och mata in den rödmakerade **`print()`**-satsen efter prompten **`>>>`** som du ser nedan:

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32
bit (Intel)] on win32
Type “help”, “copyright”, “credits” or “license” for more information.
>>>
>>> print('\n\t Välkommen till Koda matte med Python! \n')

         Välkommen till Koda matte med Python!

>>> ^Z
```

`print()` är en fördefinierad funktion i Python som skriver ut text på skärmen. Text omges i koden av apostrofer `' '`. Själva tecknet `'` är kod och visas inte i utskriften. `\n` bryter rad och `\t` skapar horisontellt avstånd. Mer om `print()` se sid 27.

För att lämna Python kan man mata in tangentsekvensen `<ctrl>-Z`. Detta innebär att man först trycker på `ctrl`-tangenten och, utan att släppa den, trycker på `z`, vilket ger kommandot `^Z` i prompten `>>>`. Sedan avslutar man med `Enter`. Man kan även lämna Python med koden `exit()` följd av `Enter`. Självklart kan man lika bra stänga pythonfönstret.

Övningar

0101

Ladda ner den senaste versionen av Python från Pythons officiella webbsida och installera den på din dator.

0102

Har du en Mac-dator leta på nätet efter instruktioner för installation av Python på Mac. Gör samma sak som i övn 0101.

0103

a) Vad är skillnaden mellan kompilering och interpretering?

b) Är Python ett kompilerande eller ett interpreterande programspråk?

0104

Öppna Python interpretatorn. Skriv *Hej* i prompten `>>>` . Vad händer och varför? Skriv sedan *'Hej'*. Vad händer nu? Skriv slutligen *"Hej"*.

Förklara och dra slutsats.

0105

Öppna *Interactive mode* (sid 15) och skriv först kod som skriver ut bokstaven a, sedan kod som skriver ut talet 3.

0106

Öppna Interactive mode. Skriv först koden `print('3')`
och sedan `print( 3 )`

Båda ger utskriften 3. Men vad är skillnaden?

0107

Testa följande kod i Interactive mode:

```
Print('Hej')
```

Vad är det för fel med denna sats?
Rätta till koden.

0108

a) Skriv ut endast texten *Hej* utan någon tillsats i Interactive mode.

b) Snygga till din utskrift med hjälp av koderna \n och \t så att *Hej* hamnar med ett avstånd från den vänstra kanten samt med en tom rad före och efter.

0109

Skriv pythonkod som skriver ut:

```
Hej, välkommen till Python!
```

Testa din kod.

0110

Modifiera övn 0109 så att utskriften blir:

```
Hej,
välkommen till Python!
```

0111

Modifiera övn 0110 så att utskriften blir:

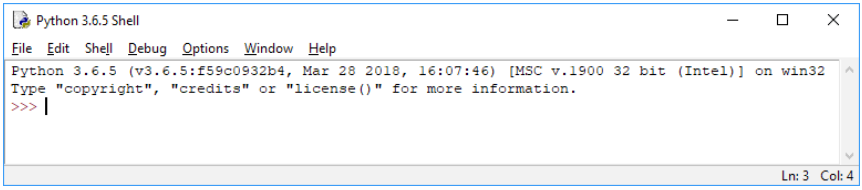
```
Hej,
välkommen
till
Python!
```

Pythons utvecklingsmiljö IDLE

Vid nedladdningen av Python enligt förra avsnitt följer utvecklingsmiljön IDLE med: *Integrated Development and Learning Environment*. Om man vill göra mer än att testa korta kodsnuttar i Interactive mode, t.ex. exekvera längre kod och spara den i en fil, behövs en utvecklingsmiljö som har en editor och andra verktyg. Detta blir faktiskt nödvändigt när man vill skriva lite längre program. Annars måste man, varje gång man vill testa ett program, mata in det i Interactive mode, vilket i längden är praktiskt ohållbart. Med hjälp av IDLE kan man öppna en fil, skriva hur mycket kod som helst, spara och lagra filen samt ladda och exekvera koden när det behövs. Här förklaras hur man kan göra det i Pythons egen utvecklingsmiljö IDLE.

Att skriva pythonkod i en fil

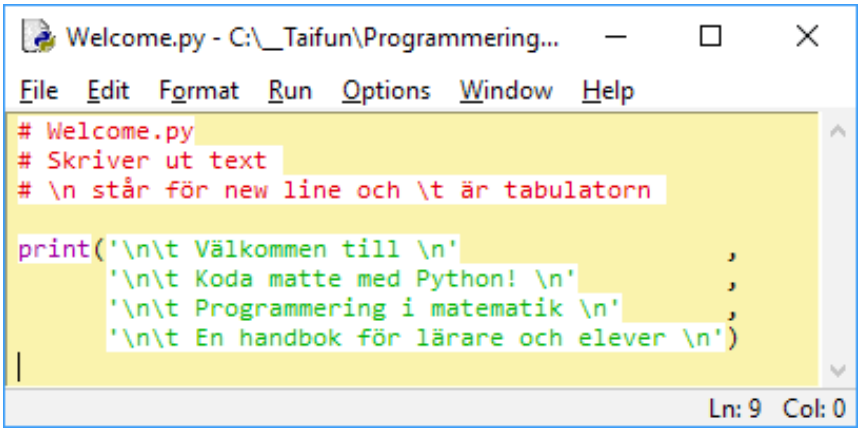
Öppna IDLE , se ikonen i huvudrubriken. Följande fönster dyker upp som kallas för Shell-fönstret:



Som man ser liknar Shell-fönstret Python interpretatorn. Skillnaden är att den (förutom den vita bakgrunden) har en menyrad med ett antal menyer File, Edit, Shell, … . Den vita ytan efter prompten >>> är faktiskt Python interpretatorn och ingår därmed i IDLE. Gå till den nya menyraden längst upp för att skapa en fil. Klicka på:

File → New File

Vi vill skriva pythonkod i filen. Ett nytt fönster dyker upp – Edit-fönstret. Skriv koden du ser på nästa sida i Edit-fönstret. Bry dig just nu inte om kodens färgläggning.



Som du ser har vi bytt bakgrundsfärgen. Se anvisningarna i slutet av nästa sida. Vad gäller kodens färger har IDLE-miljön stöd för *syntax highlighting*: Vissa delar av koden får automatiskt en färg. De tre första raderna t.ex. blir röda eftersom de är *kommentarer* som ska förklara koden. De utförs inte av Python interpretatorn.

I pythonkod inleds *radkommentar* av tecknet **#** som gäller till slutet av raden. En radkommentar kan även börja mitt på en rad, som t.ex. i vårt nästa programexempel **Aritm_uttryck** (sid 26, rad **9**).

Att kommentera koden är en fråga om god programmeringsstil. Även indragningar och tomma rader kan tjäna detta syfte. *God programmeringsstil* handlar om att skriva kod som är strukturerad och uppfyller kraven på:

- Läslighet
- Förståelighet
- Ändringsbarhet

I programmet **Welcome** ovan skriver ut `print()`-satsen (i de fyra sista raderna) text på skärmen som i koden omges av tecknet **'**. Koderna **\n**, **\t** samt tecknet **'** förklarades kort på sid 15 (Interactive mode). `print()` kommer att behandlas utförligt på sid 27. IDLE-miljön färglägger text i koden med grön.

För att spara koden ovan gå till menyraden längst upp och klicka på:

File → Save As …

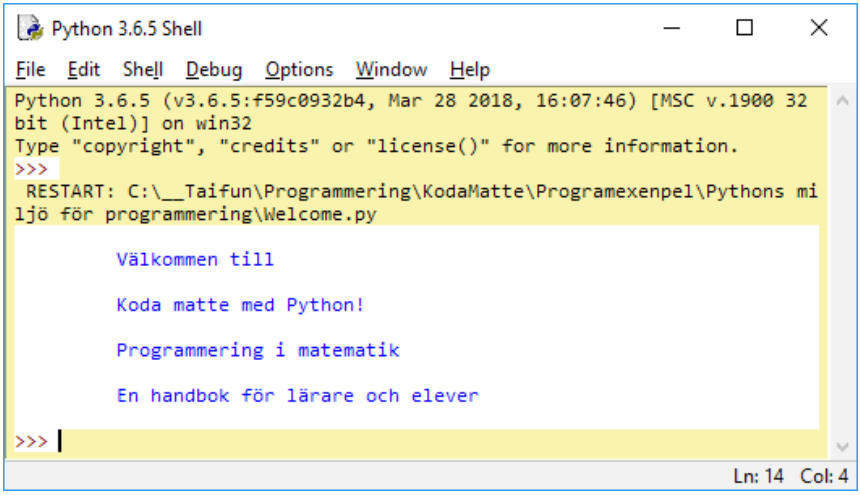
Navigera i filsystemet på din dator till ett valfritt ställe där du vill lagra dina pythonfiler. Skriv i textfältet med rubriken Filnamn: `Welcome.py`. Glöm inte att klicka på knappen Spara.

Att exekvera pythonkod från en fil

För att exekvera programmet `Welcome` klicka i menyraden längst upp på:

Run → Run Module

Om allt gått bra bör du nu se följande utskrift: Jämför den med koden på förra sidan.



Vill du byta bakgrundsfärg på IDLE-fönstren gå till menyn Options och välj Configure IDLE. Välj fliken Highlights, välj Background och Choose Color for: och välj färg.

I fortsättningen kommer vi att visa program vi skriver i filer och deras körresultat, på en form som är grafiskt mer tilltalande än det du ser ovan. Vi använder för detta Visual Studio, en alternativ utvecklingsmiljö som behandlas i slutet av boken.

Du kan förstås använda din egen favorit utvecklingsmiljö eller välja mellan IDLE och Visual Studio.

Övningar

Pythons utvecklingsmiljö IDLE

0201

Öppna din favorit utvecklingsmiljö för Python, mata in koden till programmet `Welcome` och kör så att utskriften blir exakt som på sid 19.

Har du ingen favorit pythonmiljö ladda ner och installera Python enligt anvisningarna på sid 14. Skriv in och exekvera programmet `Welcome` enligt instruktionerna på sid 17-19.

0202

Modifiera övn `0201` så att utskriften blir:

```
Välkommen till
Koda matte med Python!
Programmering i matematik
En handbok för lärare och elever
```

0203

a) Modifiera övn `0202` genom att i koden ersätta alla apostrofer (') med citationstecknet ("). Vilken slutsats drar du om kodning av strängar i Python?

b) Undersök skillnaderna mellan apostrof, citationstecken och accent.

Gå till Interactive mode. Testa att skriva ut först accenttecknen, sedan apostrofen och citationstecknet. Följ följande regel: När tecknet backslash \ sätts framför ett tecken ändras tecknets betydelse: Koderna `\'` och `\"` inom en sträng skriver ut själva apostrofen ' och citationstecknet ", för annars skulle de betyda strängens början och slut. Testa även att skriva ut själva backslash `\` .

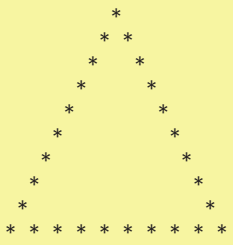
0204

Använd kunskapen från övn `0203` b) för att skriva pythonkod som skriver ut:

```
"Adjö" skrivs på franska "Adieu".

I Python omges text av ' eller ".

\n ger newline, \t ger tabulator.

\b ger backspace, \\ ger backslash.
```

0205

Skriv ett pythonprogram som skriver ut en triangel byggd av stjärnor:



0206

Skriv ett pythonprogram som ritar följande hjärtlig hälsning på skärmen:

Kurs
Matematik 1 (a, b, c)

Kap 1 Taluppfattning

1.1 Aritmetiska uttryck

Hur räknar Python?

$6 + 3 \cdot 5$ → Först $6 + 3 = 9$, sedan $9 \cdot 5 = 45$
eller
Först $3 \cdot 5 = 15$, sedan $6 + 15 = 21$?

Vanligt fel: $6 + 3 \cdot 5 = 9 \cdot 5 = 45$

Rätt: $6 + 3 \cdot 5 = 6 + (3 \cdot 5) = 6 + 15 = 21$

Felet begås inte bara av elever som ”glömmer bort” aritmetikens prioritetsregler utan även av digitala verktyg där prioritetsreglerna inte är implementerade. Men hur fungerar Python i detta avseende? För att besvara det och lära oss att hantera aritmetiska uttryck i Python ska vi testa exemplet ovan och annat av aritmetiskt intresse. Men först lite fakta om Pythons aritmetiska operatorer samt deras prioritetsordning:

OPERATION	UTTRYCK	PYTHONKOD	OPERATOR	PRIORITET
Tilldelning	$a = 9$	`a = 9`	`=`	5
Addition	$a + 3$	`a + 3`	`+`	4
Subtraktion	$f - g$	`f + g`	`-`	4
Multiplikation	$p\,q$	`p * q`	`*`	3
Division	c/d eller $\frac{c}{d}$	`c / d`	`/`	3
Heltalsdivision	c/d eller $\frac{c}{d}$	`c // d`	`//`	3
Modulo	$r \bmod s$	`r % s`	`%`	3
Exponentiering	b^x	`b ** x`	`**`	2
Parentes	$(\quad)$	`(  )`	`(  )`	1

Prioritet 1 betyder högsta och 5 lägsta prioritet. Därför utförs t.ex. parentes med prioritet 1 först och tilldelning med prioritet 5 sist.

OBS! *Lägre* prioritetsordning (siffra) betyder *högre* prioritet.

Exempel: Multiplikation (prioritet 3) går före addition (prioritet 4): $6 + 3 \cdot 5 = 21$

Att inkludera parentesen i operatortabellen ovan kan låta märkligt. Vi är inte vana vid att behandla parentesen som operator. Endast i uttryck kan parentesen anses som en operator. Inom programmering brukar man behandla parentesen som en aritmetisk operator precis som alla andra. Detta ger inte bara en enhetlig syn på operatorprioritet, utan även en bättre förståelse för förenkling av uttryck. Vad vore aritmetiken utan operatorernas ”konung” som har den högsta prioriteten bland alla operatorer?

Även tilldelningen med symbolen = (som i programmering inte betyder likhet) behandlas som en aritmetisk operator, vilket kommer att förklaras utförligt i nästa avsnitt.

Om flera operatorer med samma prioritet (utom **) förekommer i ett uttryck, utförs de från vänster till höger, t.ex.:

$$8 / 4 \cdot 3 \quad = \quad (8 / 4) \cdot 3 \quad = \quad 6$$

Men: Flera exponentieringar (**) utförs från höger till vänster, t.ex.:

$$2^{3^2} \quad = \quad 2^{(3^2)} \quad = \quad 2^9 \quad = \quad 512$$

Exponentiering fungerar alltså själv som en parentes. Nästlade parenteser beräknas *inifrån*, precis som nästlade funktioner, t.ex. f (g(x)), först g(x) sedan f(x), alltså från höger till vänster. Vi ska testa alla dessa regler i Python, dessutom följande

Uppgift:

Beräkna uttrycket y :s värde för $x = 5$:

$$y \quad = \quad 2x^2 + 3x + 7 - 4(x + 5)$$

Vi låter Python göra jobbet, först direkt i Interactive mode som ett slags experiment, och sedan i programmet `Aritm_uttryck` (sid 25-26).

Python som smart kalkylator

På sid 15 hade vi introducerat Pythons *Interactive mode* som innebar att vi direkt kunde mata in pythonkod i interpretatorn och få svar med detsamma. Detta ger oss möjligheten att använda Python som en smart kalkylator.

Klicka på pythonikonen  för att öppna Python interpretatorn (sid 14). Mata in de aritmetiska uttryck som vi skrev ovan efter prompten **>>>**.

Resultaten av denna inmatning ser du nedan:

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900
32 bit (Intel)] on win32
 Type “help”, “copyright”, “credits” or “license” for more informa
tion.
>>>
>>> 6 + 3 * 5
21
>>> 8 / 4 * 3
6.0
>>> 2**3**2
512
>>> x = 5
>>> 2*x**2 + 3*x + 7 - 4*(x + 5)
32
>>> (2 + 6) / (3 + 1)
2.0
>>> ^Z
```

Att `6+3*5` ger `21` är klart. Men varför ger `8/4*3` `6.0` och inte `6`? Varför blir `(2+6)/(3+1)` `2.0` och inte `2`? Det beror på att vanlig division **/** alltid ger decimaltal.

När vi matar in `x = 5` som betyder att `x` får värdet `5`, får vi inget svar utan prompten kommer tillbaka, eftersom Python lagrar värdet `5` i minnescellen `x`. Hade vi sedan skrivit `x` hade vi fått `5`. Men istället matar vi in det långa uttrycket med `x` i nästa rad, se även uppgiften ovan. Då beräknar Python uttryckets värde för `x = 5` (från minnet) åt oss och skriver ut resultatet `32`. Självfallet kan `x` användas och har värdet `5` endast i denna session av Python interpretatorn.

Inmatningen `(2 + 6) / (3 + 1)` förklaras på nästa sida:

Osynlig parantes

Exempel:

$$\frac{2+6}{3+1}$$

Vanligt fel med digitalt verktyg: $2+6/3+1=2+2+1=5$

Rätt med digitalt verktyg: $(2+6)/(3+1)=8/4=2$

Rätt med papper & penna: $\frac{2+6}{3+1}=\frac{(2+6)}{(3+1)}=\frac{8}{4}=2$

”Digitalt verktyg” som gör det vanliga felet kan vara miniräknare, men även Python eller något annat programmeringsspråk. I själva verket är det inte verktygets utan vårt eget fel. När vi skriver om bråkstrecket till divisionstecknet `/` får vi inte glömma att sätta parenteser som var osynliga innan. Den korrekta omskrivningen är nämligen:

$$\frac{2+6}{3+1} \quad = \quad (2+6)\,/\,(3+1)$$

Bråkstrecket inkluderar osynliga parenteser, liknande exponentiering. Denna omskrivning har använts i Python, se bilden på förra sidan.

Experimentet visar att även `x = 5` kan användas i Interactive mode precis som i vanliga program – en funktionalitet som utmärker Python bl.a. som en smart kalkylator som kan användas för att testa både matematiska beräkningar och pythonkod. Nackdelen med Interactive mode är att all kod försvinner för gott när vi lämnar Python. Om vi vill behålla koden och köra programmet när vi vill, så måste koden sparas i en fil. Detta visas nedan.

Från kalkylator till program

Öppna Pythons utvecklingsmiljö IDLE (sid 17), följ instruktionerna där och mata in i Edit-fönstret programmet **`Aritm_uttryck`** som du ser nedan. Beakta att radnumren och de indragningar som följer efter radnumren inte ska vara med i koden.

```
1   # Aritm_uttryck.py
2   # Testar Pythons aritmetiska operatorer och deras prioritet
3
4   uttryck_1 = 6 + 3 * 5
5   uttryck_2 = 8 / 4 * 3
6   uttryck_3 = 2**3**2
```

```
7   x = 5
8   y = 2*x**2 + 3*x + 7 - 4*(x + 5)
9   osynlig_parentes = (2 + 6) / (3 + 1)     # Se förra sidan
10
11  print('\n\t 6 + 3 * 5 ='  , uttryck_1,
12        '\n\n\t 8 / 4 * 3 =', uttryck_2,
13        '\n\n\t   2**3**2 =', uttryck_3,
14        '\n\n\t         y =',    y     ,
15        '\n\n\tOsynlig parentes ger', osynlig_parentes,'\n')
```

Följ instruktionerna på sid 19 för att exekvera programmet `Aritm_uttryck`. Så här borde körresultatet se ut (bortsett från form och färg):

```
 6 + 3 * 5 = 21

 8 / 4 * 3 = 6.0

   2**3**2 = 512

         y = 32

Osynlig parentes ger 2.0
```

Dessa resultat följer prioritetsreglerna för aritmetiska operatorer som visades i operatortabellen på sid 22 och överensstämer med de resultat vi fick i Interactive mode (sid 15). Skillnaden är att vi här lagrar de aritmetiska uttrycken först i variabler för att sedan skriva ut dem i en **`print()`**-sats. Variabler i programmering tas upp i nästa avsnitt. Först ska vi reda ut några begrepp:

Sats i Python

En sats (eng. *statement*) i programmering är en *instruktion* till datorn att utföra något, liksom ett kommando. En sats i Python måste avslutas med en tryckning på Enter-knappen som är en *del* av satsen, en *obligatorisk* sådan, det allra *sista tecknet* i satsen. Enter är pythonspråkets *satsavslutningstecken* vars utelämnande är ett *syntaxfel*. Observera att man inte ser Enter-tryckningen i de koder vi ser på skärmen eller skriver på papper, bl.a. i alla våra programexempel i denna bok.

Enter-tryckningen ersätter semikolonet som är motsvarande kod i de flesta programmeringsspråken som t.ex. C#, Java, C++, VB osv. Även i Python är det tillåtet att använda semikolon som satsavslutningstecken, men vi föredrar Enter för att minimera koden.

`print()`-satsen

`print()` är en inbyggd funktion i Python. Därför är också dess syntax den av en funktion. Innebörden av funktioner i programmering tas upp senare (sid 70). I programmet **`Aritm_uttryck`** (sid 25-26) börjar **`print()`**-satsen så här:

```
11    print('\n\t 6 + 3 * 5 ='   , uttryck_1,
```

Allt man vill skriva ut med **`print()`** kodas inom parenteserna i en kommaseparerad lista, kallad *parameterlista*. I raden ovan stängs inte parentesen eftersom satsen fortsätter på nästa rad. I denna lista kan data av olika typer förekomma: tal, text, variabler, konstanter osv. Listans ingredienser kallas för *parametrar*. Den första parametern i raden ovan är strängkonstanten **`'\n\t 6 + 3 * 5 ='`**. Att det är en sträng ser man på apostroferna **`' '`**.

Koden **`\n`** som betyder *newline* och bryter rad i utskriften gör att den första utskriftsraden blir en tom rad. Koden **`\t`** som betyder *tabulator* gör att den andra raden skrivs ut med ett horisontellt avstånd. Själva apostroferna **`' '`** är kod och kommer inte att visas i utskriften. Så endast **`6 + 3 * 5 =`** skrivs ut efter en tom rad och horisontellt avstånd.

Den andra parametern är heltalsvariabeln `uttryck_1`. Det som skrivs ut är förstås variabelns värde 21. I programmet **`Aritm_uttryck`** (rad **4**) tilldelas variabeln `uttryck_1` detta värde i form av uttrycket **`6 + 3 * 5`** som ju ger 21.

Sedan avslutas rad **11** i koden ovan efter kommat. Det kan man göra efter ett komma, om man inte stänger den parentes av **`print(`** som öppnades innan. Satsen fortsätter på nästa rad:

```
12          '\n\n\t 8 / 4 * 3 =', uttryck_2,
```

Koden betyder: Strängen inom apostrofer skrivs ut efter två radbyten och ett horisontellt avstånd, följt av värdet `6.0` som tilldelas variabeln `uttryck_2` genom uttrycket **`8 / 4 * 3`**, se **`Aritm_uttryck`** (rad **5**) .

På liknande sätt går det vidare till rad **15**. Där avslutas **`print()`**-satsen genom att vi stänger den parentes av **`print(`** som hade öppnats i rad **11**.

I detta avsnitt har vi flera gånger nämnt termen *variabel* vars användning i programmering behandlas mer ingående i nästa avsnitt. Där kommer vi även att lära oss innebörden samt användningen av tilldelningsoperatorn (=) som redan dök upp i operatortabellen på sid 22.

Övningar

1101

Använd Python som smart kalkylator för att beräkna följande aritmetiska uttryck:

a) $7 + 4 \cdot 2$

b) $9 - 8 / 4$

c) $12 + 18 / 9 - 6$

d) $\dfrac{12 + 18}{9 - 6}$

1102

Räkna först utan Python och kontrollera sedan dina resultat med Python:

a) $5 + 3 \cdot 8 - 6$

b) $(5 + 3) \cdot (8 - 6)$

c) $3(6 - 4) + 2(5 - 2)$

d) $6\,(3 + 1 \cdot 2) - 4 \cdot 5$

1103

Ett taxibolag tar en fast avgift på 25 kr. Därefter kostar det 10 kr per km att åka. Beräkna med Python hur mycket det kostar att åka 20 km. Ställ upp ett aritmetiskt uttryck för taxan om man åker x km.

1104

Ett mobilabonnemang har en öppningsavgift på 1,50 kr. Det kostar 25 öre per minut att ringa. Ställ upp ett aritmetiskt uttryck för kostnaden om man pratar i x sekunder. Beräkna uttryckets värde för x = 59, dvs låt Python beräkna hur mycket det kostar att prata i 59 sekunder.

1105

Beräkna följande aritmetiska uttryck utan digitalt verktyg. Kontrollera dina resultat i Pythons Interactive mode.

a) $(-5)^2 - 3^2$ b) $\dfrac{2^2 + 3^2}{(2 + 3)^2}$

c) $\dfrac{(4 - 2)^2}{4^2 - 2^2}$ d) $(2^2)^3$ e) 2^{2^3}

1106

Låt ett pythonprogram beräkna och skriva ut följande uttryckets värde för x = -1 :

$$4x^3 - 2x^2(2x + 6) + 7x\,(3 + 2x)$$

1107

a) Beräkna följande uttryck i Python:

$$(6^2 + 6^2 + 6^2) / 9$$

b) Kontrollera Pythons svar genom att förenkla uttrycket och beräkna det utan digitalt verktyg.

1108

Skriv ett pythonprogram i en fil som beräknar och skriver ut de aritmetiska uttryckens värden från övn **1102** a) - d) och övn **1105** a) - e).

1109

Experimentera med Python som smart kalkylator för att lösa följande uppgift:
Hitta ett positivt heltal för x så att uttryckets värde blir störst:
Beräkna detta maximala värde. Motivera.

$$\frac{87 + 13}{(x + 9) / 5}$$

1.2 Variabler

En variabel är en platshållare (minnescell) för ett värde (data). Den får ett namn som används för att komma åt värdet. En variabels värde kan ändras under programmets gång, men inte namnet.

Exempel: I programmet **Aritm_uttryck** (sid 25-26) är `uttryck_1`, `uttryck_2`, `uttryck_3`, `x`, `y`, och `osynlig_parentes` variabler. I raderna **4-9** tilldelas de värden som är resultat av aritmetiska uttryck vi ville testa i Python. Tilldelningen görs med den s.k. *tilldelningsoperatorn* (=) vars användning visas längre fram. Sedan skriver **`print()`**-satsen ut dessa värden (rader **11-15**). Under tiden lagras de i minnesceller.

Variabler är *logiska* adresser till datorns *fysiska* minnesceller: vi kan skriva till dem och läsa från dem. I programmet **Aritm_uttryck** skriver tilldelningsoperatorn (=) till dessa adresser och **`print()`**-satsen läser från dem.

Variabler liknar lådor med etiketter. Innehållet är värdet, etiketten är namnet. Det är avgörande att skilja mellan *namnet* och *värdet*. Precis som lådors innehåll (värden) kan ändras, men inte deras etiketter (namn), kan variablers namn inte heller ändras i ett program. Att ändra variablernas *värden* är däremot en vanlig teknik inom programmering som kallas för *överskrivning* och behandlas på sid 31. Men vilka namn kan vi och vilka bör vi ge till våra variabler? Det finns regler som vi *måste* följa, och det finns rekommendationer som vi *bör* använda.

Regler för namngivning

Ett namn på en variabel kan bestå av ett eller flera tecken och får endast innehålla:

- Alla bokstäver, inkl. svenska specialtecken
- Alla siffror
- Understreck, eng. *underscore* (_)

Ett namn på en variabel får inte:

- Börja med en siffra
- Vara ett av Pythhons *reserverade ord* (se nästa sida).

Självklart får ett namn inte innehålla mellanslag för då tolkas det inte som *ett* utan *flera* namn. Mellanslag är *avskiljare* mellan ord. Minst lika viktig som dessa regler är följande rekommendation:

Rekommendation för namngivning

För att skriva lättlästa och förståeliga program bör man välja *beskrivande namn* för sina variabler, för att lätt kunna relatera dem till variablernas roll och syfte i programmet – kort sagt deras *användning*.

Denna rekommendation har vi följt i bokens alla programexempel och lösningar.

Reserverade ord

Som alla programmeringsspråk är även Python definierat av ett antal nyckelord, språkets ordförråd, även kallade *reserverade ord* (eng. *Keywords*). Den version av Python som denna bok bygger på, har i skrivande stund följande 33 reserverade ord:

RESERVERADE ORD I PYTHON (läs kolumnvis!)

`False`	`class`	`finally`	`is`	`return`
`None`	`continue`	`for`	`lambda`	`try`
`True`	`def`	`from`	`nonlocal`	`while`
`and`	`del`	`global`	`not`	`with`
`as`	`elif`	`if`	`or`	`yield`
`assert`	`else`	`import`	`pass`	
`break`	`exept`	`in`	`raise`	

Eftersom Python permanent vidareutvecklas är denna tabell versionsberoende. Du får alltid reda på den aktuella listan om du i Interactive mode (sid 15) skriver:

```
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'break',
'class', 'continue', 'def', 'del', 'elif', 'else', 'except',
'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is',
'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return',
'try', 'while', 'with', 'yield']
>>>
```

Tabellen ovan är endast en strukturerad version av denna lista.

En annan allmän regel som gäller för all pythonkod är:

> Python är case sensitive.

Python skiljer på gemener och versaler. Om vi t.ex. skriver det reserverade ordet `True` som **`true`** kommer Python inte känna igen det och säga att **`true`** inte är definierat.

Självklart gäller case sensitivity även vid val av variabelnamn: `uttryck_1` är en annan variabel än `Uttryck_1`.

Överskrivning eller kan **`x = x + 1`** vara sant?

Vi återvänder till aritmetiska uttryck och tittar närmare på den första raden i operatortabellen på sid 22:

OPERATION	UTTRYCK	PYTHONKOD	OPERATOR	PRIORITET
TILLDELNING	$a = 9$	`a = 9`	`=`	5

Tecknet = betyder i Python *tilldelning* och är en aritmetisk *operator* precis som alla andra i tabellen. Den har den lägsta prioriteten, utförs alltså sist i aritmetiska uttryck. Symbolen är likhetstecknet, men har inte betydelsen *likhet* som i matematik. Vi ska titta närmare på skillnaden mellan tilldelning och likhet. I algebra skulle $x = x + 1$ leda till en logisk motsägelse: $0 = 1$ eller: ”Ekvationen saknar lösning”. I Python är däremot **`x = x + 1`** en korrekt, meningsfull sats om variabeln **`x`** har fått ett värde innan. Likhet har i Python en annan symbol, nämligen == (sid 45).

Tilldelningsoperatorn används i Python så här:

```
variabel    =    värde
variabel  <------  värde
```

I satsen `a = 9` t.ex. tilldelas variabeln `a` värdet 9. Eller i satsen `sum = r + t` bildas först summan `r + t` därför att + har prioriteten 4 medan = har prioriteten 5. Sedan tilldelas summan `r + t` variabeln `sum`. Förutsättningen är att i det läget `r` och `t` redan har värden. I båda dessa exempel står den tilldelade variabeln på *en* sida av = .

Nu ska vi studera en annan struktur där den tilldelade variabeln **`x`** finns på *båda* sidor av = :

```
x     =     x + 1
x   <------ x + 1
```

Här måste variabeln x till höger redan ha ett värde. Det som denna sats gör är att ändra variabeln x:s värde – en grundläggande teknik inom programmering som kallas för *överskrivning*. Den demonstreras i följande program och förklaras sedan:

```
1   # Variabel.py
2   # Tilldelar en variabel och överskriver den
3   # Skriver ut både det gamla och det nya, överskrivna värdet
4   # Demonstrerar skillnaden mellan likhet och tilldelning
5
6   x = 5                                          # Tilldelning
7   print('\n\t Variabeln har initierats till', x, '.')
8
9   x = x + 1                                      # Överskrivning
10
11  print('\n\t Sedan har den ökats med 1 och är nu', x, '.\n')
```

I rad **6** tilldelas variabeln **x** värdet **5**. I rad **9** ändras variabelns värde med:

```
x     =     x + 1
```

Före denna sats har **x** värdet **5**. Detta värde adderas med **1** och det nya värdet **6** tilldelas till variabeln **x** på nytt:

```
x  <-----  5 + 1
```

Efter satsen har **x** värdet **6**. Det nya värdet **6** *skriver över* det gamla värdet **5**:

```
x  | 6 |
```

Detta kallas för *överskrivning*. Vi har att göra med två olika värden till en och samma variabel **x**, men vid två olika tidpunkter. **5** finns i variabeln **x** *före* och **6** *efter* satsen. Det beror på att tecknet = i Python betyder tilldelning till skillnad från dess matematiska betydelse som likhet. Matematiskt är det fel att skriva $x = x + 1$. I programmering däremot är det helt okej att skriva så, eftersom det inte handlar om ett påstående som kan vara sant eller falskt, utan snarare om en *instruktion* om att ge variabeln **x** (vänster om likhetstecknet) ett nytt värde genom att öka det gamla värdet (höger om likhetstecknet) med **1**. I Python *gör* = en sak och *är* inte en sak. I matematiken borde detta formuleras med två olika variabler och skrivas t.ex. så här:

$$x_{\text{nytt}} = x_{\text{gammalt}} + 1$$

I Python är däremot **x** *en enda* variabel dvs en minnescell vars innehåll byts ut

medan namnet bibehålls. Därför kan i koden *en och samma* variabel **x** användas på båda sidor av tilldelningstecknet:

```
x = x + 1
```

I själva verket handlar det om den klassiska, filosofiska skillnaden mellan *att vara* och *att bli*, mellan *tillstånd* och *handling*, mellan den statiska likheten och den dynamiska tilldelningen. Vid tilldelning relateras sanningen till tiden dvs frågan är inte om utan när `x = 5`. Jo, precis när variabeln `x` tilldelas värdet `5`. Inte före och inte heller efteråt, för redan i nästa programsats kan ju variabeln `x` tilldelas ett annat värde. Därför är tilldelningen dynamisk medan likheten är statisk.

Att satsen `x = x + 1` utför additionen först och tilldelnigen sedan beror på att operatorn **+** har högre prioritet än tilldelningsoperatorn **=**. Därför slipper vi att skriva parenteser: `x = (x + 1)`, vilket inte vore fel, bara onödigt. Prioritetsreglerna ordnar det. Det visar att även `=` fungerar precis som alla andra operatorer.

Programmet **`Variabel`** ger följande utskrift:

```
Variabeln har initierats till 5 .
Sedan har den ökats med 1 och är nu 6 .
```

Övningar

1201

Skriv ett program (i en fil) som skriver ut:

```
Summan av 5 och 3 är 8
```

Lös uppgiften genom att skapa två variabler som får värdena `5` och `3`. Tilldela deras summa till en tredje variabel och skriv ut som ovan. Utskriften ska ske med hjälp av variablerna.

1202

Utveckla lösningen till övn **1201** vidare så att utskriften blir:

```
Summan av 5 och 3 är 8

Differensen 5 - 3 är 2
```

Använd samma variabler som i **1201**. Lägg till endast en till variabel för differensen. Skriv ut med hjälp av variablerna.

1203

Komplettera lösningen till övn **1202** så att utskriften blir:

```
5  +  3  ger  8
5  -  3  ger  2
5  *  3  ger  15
5  /  3  ger  1.6666666666666667
5  // 3  ger  1
```

Skapa ytterligare variabler och tilldela de fyra räknesätten till dem.
Inkludera Pythons operator `//` för heltalsdivision (sid 22).
Skriv ut med hjälp av variablerna.

1204

Varför ger följande kod ett felmeddelande i Python? Hitta felets orsak. Åtgärda felet.

```
a   = 1
sum = sum + a
print('\n\t sum =',  sum, '\n')
```

1205

Vilka värden kommer variablerna `tal` och `prod` att ha efter följande kod?

```
tal  = 5
prod = tal * 4
tal  = tal + tal + tal + tal
```

Svara först utan Python. Testa sedan i Interactive mode. För vilken matematisk kunskap är koden ett exempel på?

1206

Vilka värden kommer variablerna `tal` och `potens` att ha efter följande kod?

```
tal    = 5
potens = tal ** 4
tal    = tal * tal * tal * tal
```

Gör samma sak som i övn **1205** och besvara samma fråga.

1207

Vilka värden får `c` och `d` efter följande kod? Svara först. Testa sedan i Python.

```
a = 123456789
b = a
c = b ** (a - b)
d = c // (b - a)
```

1.3 Inläsning av data

Målet är alltid att skriva generella program. Därför ska vi vidareutveckla förra avsnittets program **Variabel** genom att läsa in data till en variabel istället för att hårdkoda variabelns värde i programmet (sid 32, rad 6). Indata kan matas in när ett program körs. Python uppfattar generellt all inläst data som *text*, vilket kommer att visas i ett experiment (*Datatyper*, sid 37). Anledningen är att den inmatade datans typ inte kan identifieras förrän den har lästs in. Därför måste vi hitta metoder för att anpassa datan till vårt ändamål, t.ex. att omvandla den från text till tal. Följande program visar hur man kan göra det:

```
1   # Input.py
2   # Läser in ett värde till en variabel först som text
3   # Omvandlar sedan den inlästa texten till heltal
4
5   text = input('\n\t Mata in ett heltal:       ') # Inläsning
6   x = int (text)                 # Omvandlar text till heltal
7
8   print('\n\t Variabelns värde har lästs in som', x, '.')
9   x = x + 1
10  print('\n\t Sedan har det ökats med 1 och är nu', x, '.\n')
```

Programmet **Input** ger följande utskrift:

```
Mata in ett heltal:       6297

Variabelns värde har lästs in som 6297 .

Sedan har det ökats med 1 och är nu 6298 .
```

För inläsning av data använder vi oss av pythonfunktionen **input()** som vi nu ska titta närmare på.

Funktionen `input()`

Python har ett antal inbyggda dvs fördefinierade funktioner (eng. *built-in functions*). En av dem är `input()`. I rad **5** av programmet **Input** ovan används denna funktion för att läsa in data:

```
5   text = input('\n\t Mata in ett heltal:     ')  # Inläsning
```

Funktionens parameter (den röda texten inom parentes) skrivs ut på skärmen som en slags *ledtext* för användaren. Markören väntar på inmatning. När vi matar in ett tal och trycker på Enter läser `input()` in det, men inte som tal utan som en sträng. Det beror på att `input()` returnerar text och inte tal. Vi kan inte ändra detta, eftersom funktionen är fördefinierad. Strängen tilldelas variabeln `text`. För att kunna räkna med den måste vi omvandla den till tal, vilket görs i nästa rad:

```
6   x = int (text)                  # Omvandlar text till heltal
```

Denna sats omvandlar strängen i variabeln `text` till ett heltal som sedan tilldelas variabeln `x`. I rad **9** räknar vi med den: `x = x + 1`. Endast tal kan adderas, inte en sträng. Därav nödvändigheten av typomvandling. Utan den, dvs om inmatningen direkt hade tilldelats variabeln `x`, hade programmet inte fungerat. Lösningen är att mellanlagra inmatningen i variabeln `text` (rad **5**) och typomvandla den i rad **6**. Man kan även, om man vill, slå ihop dessa två rader till en:

```
    x = int (input('\n\t Mata in ett heltal:      '))
```

Fördelen med denna kod är att man slipper variabeln `text` och därmed förkortar koden. Nackdelen är att den är en aning svårare att läsa: Det blir ett nästlat anrop av de två funktionerna `int()` och `input()`. I fortsättningen kommer vi faktiskt att använda oss av koden ovan – för korthetens skull.

Funktionen `int()`

En annan inbyggd funktion i Python är **`int()`** vars generella syntax är:

```
int (uttryck)
```

Här kan **uttryck** vara en kombination av variabler, konstanter, operatorer och vanliga parenteser. I programmet **Input** består **uttryck** endast av variabeln text som lagrar funktionen input():s returvärde som en sträng. Så konkret blir det:

```
int (text)
```

I koden ovan står ordet **int** för integer (heltal) och är en *datatyp*, se nedan. Funktionen **int()** omvandlar inmatningen text till **int** för att vi ska kunna tilldela den till variabeln x och kunna räkna med den.

Funktionen float()

En ganska liknande roll spelar funktionen **float()** i Python. Vill man läsa in data som inte är heltal utan decimaltal måste omvandlingen ske till **float** som är datatypen för s.k. *flyttal* – ett annat ord för decimaltal. Funktionen **float()** omvandlar en ev. inmatad text till **float** så att vi ska kunna tilldela den till en variabel, som vi vill göra decimaltalsberäkningar med.

Datatyper

Ett litet experiment i Pythons Interactive mode (sid 15) visar att funktionen input() returnerar text och inte tal och vad som kan hända om vi ändå matar in tal:

```
a = input('Mata in ett heltal:  ')
```

Om vi då matar in 2 tolkar Python detta som tecknet '2' och inte talet 2, eftersom input() är förprogrammerat att returnera text och inte tal. Om vi sedan kombinerar a med operatorn + kan vi få överraskande resultat vilket inmatningen av 2 för a och 4 för b demonstrerar:

```
>>> a = input('Mata in ett heltal:   ')
Mata in ett heltal:   2
>>> b = input('Mata in ett heltal till:   ')
Mata in ett heltal till:   4
>>> a + b
'24'
>>>
```

Varför ger a + b '24' och inte 6? Och vad gör egentligen + där? Frågorna kan bara besvaras om man uppmärksammar a och b:s *datatyp*. Eftersom funktionen input() läser in all data som text får det värde som den läser in, datatypen **string**. Då blir även variablerna a och b:s datatyp automatiskt **string** då de tilldelas input():s returvärde. När vi sedan skriver a + b kan + inte ha betydelsen av addition därför att det är meningslöst att addera två strängar.

Vilken betydelse + har tar vi upp på nästa sida.

Först vill vi avsluta diskussionen om datatyper med den exakta definitionen på själva begreppet:

En *datatyp* är en föreskrift som talar om:

1. på vilket sätt en viss typ av data ska lagras i datorn,
2. hur mycket minne den tar och därmed hur stora värden den kan lagra,
3. vilka operatorer (funktioner) som är definierade för data av denna typ.

Olika programmeringsspråk behandlar sina datatyper på olika sätt. C#, Java, C++ är s.k. strikt typbestämda språk (eng. *strongly typed languages*), vilket innebär att man måste deklarera alla variablers datatyper explicit. Detta gäller inte för Python. Därmed är det inte sagt att det inte finns datatyper i Python. Tvärtom, all data har en datatyp. Men Python bestämmer datatyperna automatiskt och framför allt *dynamiskt*: T.ex. gör koden a = 2 att Python sätter variabeln a:s datatyp till **`int`**. Med a = 2.0 blir a:s datatyp däremot automatiskt **`float`** som står för flyttal (decimaltal). Ett annat exempel: '24' är en **`string`** medan 24 en **`int`**. Apostroferna i koden gör att 'a' är *bokstaven* a dvs en **`string`**, medan a är en variabel vars datatyp inte bestäms förrän den fått ett värde. I Python är det främst *värden* som har datatyper. Variabler får sina datatyper från värden de blir tilldelade.

Även detta visas i Interactive mode när vi först matar in `'a'` och sedan `a`:

```
>>>
>>> 'a'
'a'
>>> a
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
NameError: name 'a' is not defined
>>>
```

Skriver man `'a'` tolkar Python detta pga `' '` som en **`string`**-konstant dvs bokstaven a och returnerar den korrekt. Om man däremot bara skriver a tolkar Python detta som en variabel. Och eftersom den inte tilldelats något värde reagerar Python med ett felmeddelande: `name 'a' is not defined` . Intressant är Pythons term `name` för variabel. En god vana är att i samma veva som vi introducerar en ny variabel, tilldela ett värde till den, så att Python kan bestämma värdets datatyp, tolka koden och returnera ett korrekt svar när vi använder variabeln. T.ex.:

```
>>>
>>> a = 2
>>> a
2
>>> a = 2.0
>>> a
2.0
>>> a = 'Hallo!'
>>> a
'Hallo!'
>>>
```

När variabeln a tilldelas heltaltet 2, bestäms datatypen automatiskt till **int**. Sedan tilldelas a på nytt decimaltaltet 2.0. Då blir datatypen **float**. Slutligen får a ”värdet” 'Hallo!' vars datatyp är **string**. På så sätt bestäms datatypen dynamiskt beroende på sammanhanget. Och det är alltid *värdet* som typbestäms inte variabeln. Man kan t.o.m. säga att det är värdet som skapar (definierar) variabeln.

Observera att Python även tolkar en enda bokstav som **string**, dvs en sträng vars längd är 1. Datatypen *char* (character) för ett tecken (bokstav), som finns i många andra programmeringsspråk, finns inte i Python.

Konkateneringsoperatorn +

Operatorn + har olika betydelser beroende på vad den används på. Betydelsen bestäms genom datatypen av de variabler som omger operatorn. När variablerna a och b:s datatyp är **string** som i exemplet på sid 37, betyder + i a + b inte längre addition utan sammanslagning av strängar eftersom a och b:s datatyp inte är **int**. Sammanslagning av strängar kallas för *konkatenering*. Vi har alltså med *två olika* operatorer att göra som har koden + : i räkne- och talsammanhang är + additions-, i strängsammanhang är det konkateneringsoperatorn. Därför slår Python ihop *(konkatenerar)* de två strängarna 2 och 4 till strängen 24. Det är sammanhanget som avgör vilken betydelse som gäller. I programmering kallas detta för *överlagring av operatorer* och är en mycket vanlig företeelse. Generellt får en operator sin slutgiltiga betydelse av sin användning i en aktuell situation.

Du kan testa konkatenering bl.a. genom att lösa övningarna **0205** och **0206** (sid 20) där du kan slå ihop delsträngar som i koden hamnar på olika rader, med konkateneringsoperatorn.

Datatypen `string`

Datatyperna **`int`** och **`string`** har vi redan stiftat bekantskap med. Medan **`int`** representerar endast *ett* heltal och därför är en *enkel* datatyp, kan **`string`** lagra *flera* tecken och är därför en *sammansatt* datatyp. I Python anses även ett enskilt tecken vara en sträng av längden 1 och därför av datatypen **`string`**. Det finns även den tomma strängen vars längd är 0 (sid 80). Skillnaden mellan enkla och sammansatta datatyper blir tydlig när man t.ex. med **`string`** kan göra saker som man inte kan göra med **`int`**. Ett exempel på det är att hitta en strängs enskilda tecken.

Vi testar i Interactive mode hur man kan extrahera bokstäver ur en sträng:

```
>>> text = 'abcdefgh'
>>>
>>> text[0]
'a'
>>> text[1]
'b'
>>> text[2]
'c'
>>>
>>> text[6]
'g'
```

Man ser att hakparentesen `[ ]` tar ut en strängs enskilda tecken: `[0]` ger strängens första, `[1]` dess andra tecken osv. Siffrorna inom hakparentesen kallas för *index*. Index börjar alltid med `0`. Index 2 ger strängens 3:e tecken, bokstaven c. Index 6 ger strängens 7:e tecken dvs bokstaven g. Du kan öva dig på att hantera en strängs enskilda tecken genom att lösa övn **1303** på nästa sida.

I nästa avsnitt kommer vi att stifta bekantskap med några av programmeringens mest grundläggande *kontrollstrukturer*: först den s.k. `if`-satsen för att undvika division med `0`. Sedan diskuterar vi dess utvidgning, `if-else`-satsen som med hjälp av modulooperatorn testar delbarhet med 2 och på så sätt avgör om ett tal är jämnt eller udda (sid 46).

Övningar

1.3 Inläsning av data

1301

Satsen print(a) ger felmeddelande. Testa i ett pythonprogram vilka utskrifter följande satser ger:

```
print('a')
print('a' + 'a')
print('a', 'a')
print(6)
print('6')
print('6' + '6')
print(6 + 6)
print(6, 6)
print(6.6 + 6.6)
print('6.6' + '6.6')
```

Förklara resultaten.

1302

Modifiera programmet **Input** (sid 35) så att variabeln x:s ökade värde tilldelas en ny variabel y. Annars ska det nya programmet ge samma utskrift som programmet **Input**.

1303

Skriv ett program som läser in tre tecken och skriver ut dem i omvänd ordning. Använd datatypen **string** i ditt program (sid 40).

1304

Skriv ett program som läser in tre heltal och beräknar deras medelvärde. Programmet ska sedan skriva ut talen i omvänd ordning samt medelvärdet.

1305

Vidareutveckla lösningen till övn **1203** (sid 34) genom att ersätta de hårdkodade indata (5 och 3) med inläsning av data. Bilda med de inlästa värdena de fyra räknesätten inkl. heltalsdivision som i övn **1203**. Tilldela resultaten till variabler och skriv ut svaren.

1306

Följande program innehåller två fel:

```
a = input(' Mata in ett heltal: ')
prod = a * b
print(a, '*', b, '=', prod, '\n')
```

Hitta felen och åtgärda dem.

1307

Skriv ett program som kodar följande algoritm (tillvägagångssätt):

```
1. Läs in ett positivt heltal.
2. Multiplicera med 8
3. Lägg till 12
4. Dividera med 4
5. Dra av 3
6. Multiplicera med 2
```

Lagra varje steg i en variabel. Skriv ut slutvärdet (steg 6). Testa programmet genom att läsa in startvärdena 2, 5, 8 och 10. Finns det något enkelt samband mellan start- och slutvärdet? I så fall beskriv det. Kommer andra startvärden att visa samma samband? Testa gärna! Bevisa sambandet matematiskt.

1.4 Delbarhet

Vi börjar behandlingen av delbarheten med ett viktigt undantag: i matematiken är division med `0` inte definierad. Så här reagerar Python på detta:

```
>>> 8 / 0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

Vi får ett felmeddelande. Hade division med `0` förekommit som ett led i ett större program hade det lett till ett programavbrott, det som man brukar kalla för en *krasch!* – i regel *utan* ett felmeddelande. I fall av en sådan oväntad händelse skulle resten av programkoden inte exekveras. Användaren får i regel inte reda på avbrottets orsak. Vi bör därför se till att divisionen inte utförs om det värde man delar med, kan bli `0`. Följande program visar hur man kan programmera detta:

```
1   # Division_0.py
2   # Förhindrar division med 0 och därmed programavbrott
3   # if-satsen, villkor, jämförelseoperatorer
4
5   text = input('\n\t Mata in ett heltal:    ')   # Inläsning
6   tal1 = int (text)                 # Omvandlar text till tal
7   text = input('\n\t Mata in ett heltal till:  ')
8   tal2 = int (text)
9
10  if tal2 != 0 :
11     print('\n\t', tal1, ' dividerad', ' med ', tal2,
12                                    ' blir ', tal1/tal2, '\n')
13  if tal2 == 0 :
14     print('\n\t OBS!\n\t Du har matat in 0 för det andra talet.',
15                  '\n\t Division med 0 är inte definierad.\n')
```

Programmet läser in två heltal i raderna `5-8` och skriver ut deras kvot. Den första `if`-satsen gör att divisionen endast sker om det tal som ska divideras med, `tal2`, *inte* är `0`. I rad `10` betyder `!=` *inte lika med*. Om `tal2` däremot är lika med `0` gör den första `if`-satsen ingenting. Programflödet hoppar över denna sats och går vidare till den andra `if`-satsen. Där skrivs ut i rad `14` en text som man skulle kunna kalla för ett egenprogrammerat felmeddelande: `OBS!...` .

Följande dialog får man, om man matar in ett värde skilt ifrån `0` till `tal2`:

```
Mata in ett heltal:       20

Mata in ett heltal till:  5

20  dividerad  med  5  blir  4.0
```

På köpet tillåter programmet också att testa regeln: `0/a = 0` för alla a. Om man matar in `0` till `tal1` och ett tal skilt ifrån `0` till `tal2`, vilket som helst, får man följande dialog:

```
Mata in ett heltal:       0

Mata in ett heltal till:  5

0  dividerad  med  5  blir  0.0
```

Om man däremot matar in `0` till `tal2` skriver den andra **`if`**-satsen ut det egenprogrammerade felmeddelandet: **`OBS!...`** :

```
Mata in ett heltal:       8

Mata in ett heltal till:  0

OBS!
Du har matat in 0 för det andra talet.
Division med 0 är inte definierad.
```

Programmet **`Division_0`**:s programmeringstekniska nyhet är **`if`**-satsen vars generella innebörd och användning i Python vi diskuterar nu:

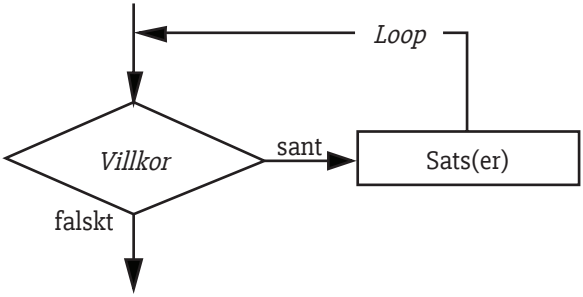
`if`-satsen

`if`-satsen är ett val mellan ett alternativ och ingenting, även kallad *enkel selektion*. Valet är mellan att göra något eller inte göra det, vilket avgörs av ett villkor. Är villkoret sant, utförs en eller flera satser. Är villkoret falskt, görs ingenting. `if`-satsens pythonkod samt dess logiska struktur som åskådliggörs med en s.k. *flödesplan* - även kallad *flödesschema* - ser ut så här:

Pythonkod

```
if villkor :
    Sats(er)
```

Flödesplan



Pythonkodens första rad kallas för `if`-satsens *huvud*, resten är *kroppen*. Endast om *villkoret* är uppfyllt utförs kroppen. Man kan skriva kroppen på samma rad som huvudet direkt efter kolonet `(:)`. Men om man skriver den på en ny rad, vilket man borde göra för bättre läslighetens skull, måste hela kroppen stå indragen.

Observera att indragningen i kodens andra rad tillhör syntaxen och inte får utelämnas. Indragningen avgör när `if`-satsen avslutas. Allt som man vill ha i `if`-satsen måste dras in. Dessutom främjar indragningen kodens läslighet.

I programmet **`Division_0`** finns det två `if`-satser. Den första `if`-satsens huvud är:

```
if tal2 != 0 :
```

Och betyder: Om `tal2` är skilt ifrån `0`

Huvudet inleds med `if` följt av villkoret `tal2 != 0` följt an ett kolon `(:)`. Kolonet tillhör syntaxen och inte får utelämnas under några omständigheter.

Den andra `if`-satsens huvud är:

```
if tal2 == 0 :
```

Och betyder: Om `tal2` är lika med `0`

Villkoret `tal2 == 0` formuleras med dubbeltecknet `==` som står för likhet, till skillnad från tecknet `=` som står för tilldelning och diskuterades tidigare (sid 31).

Villkor

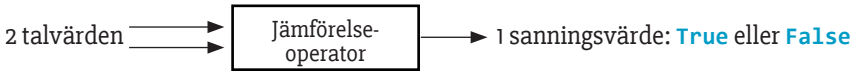
Ett *villkor* är en fråga som endast kan besvaras jakande eller nekande: är `tal2` skilt ifrån `0`, ja eller nej? Logiskt sett är ett villkor en *utsaga* som endast kan vara *sann* eller *falsk*. Man kan också säga att ett villkor är ett logiskt uttryck.

I programmering är det avgörande att skilja mellan *villkor* och *sats*. En *sats* är en instruktion som ska *utföras*, medan ett villkor endast kan *testas*. Villkoret `tal2 != 0` testar om `tal2` är skilt ifrån `0` eller ej. Dubbeltecknen `!=` och `==` som används för att formulera villkor och därmed logiska uttryck, kallas för *jämförelseoperatorer*.

Jämförelseoperatorer

<	mindre än
<=	mindre än eller lika med
>	större än
>=	större än eller lika med
==	lika med
!=	icke lika med

De tar in två talvärden och jämför dem med varandra och returnerar resultatet som ett *sanningsvärde* dvs sant eller falskt, **`True`** eller **`False`**:



Sanningsvärdena **`True`** och **`False`** är de enda värden som villkor kan anta. Exempel på villkor är:

```
number ==  0
number !=  0
     7  >  5
guessedNo  <= 17
```

Observera att de jämförelseoperatorer som är dubbeltecken, inte får innehålla mellanslag. Annars tolkas de som respektive tecken och inte som jämförelseoperatorer.

Jämna och udda tal

```
1   # Delbarhet_2.py
2   # Testar delbarhet med 2
3   # Läser in ett heltal och avgör om det är jämnt eller udda
4   # Tåvägsval: if-else-satsen
5
6   tal = int(input('\n\tMata in ett heltal:\t'))   # Inläsning
7
8   if tal % 2 == 0 :
9      print('\n\tDet inmatade talet ', tal,' är jämnt.\n')
10  else :
11     print('\n\tDet inmatade talet ', tal,' är udda.\n')
```

Jämn delbarhet med 2 testas med hjälp av modulooperatorn **%** som vi såg i operatortabellen på sid 22. I `if-else`-satsens villkor (rad **8**) används modulooperatorn **%** som behandlas mer ingående på sid 48. Just här betyder villkoret: *Ger* `tal` *heltalsdividerat med* 2 *resten* `0`? Om ja, är `tal` jämnt, annars udda. Jämna tal är jämnt delbara med 2, dvs ger resten `0`, medan udda tal ger resten 1 vid division med 2.

Så här kan två körresultat av programmet ovan se ut:

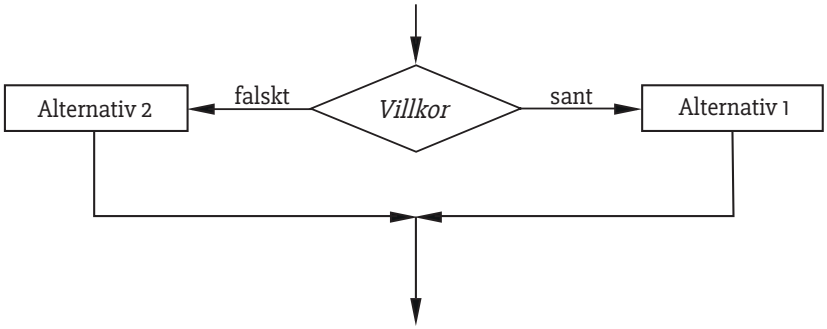
```
        Mata in ett heltal:     99

        Det inmatade talet  99  är udda.
```

```
        Mata in ett heltal:     26

        Det inmatade talet  26  är jämnt.
```

`if-else`-satsen

`if-else`-satsen är ett val mellan två alternativ, även kallat *tvåvägsval*. Precis som i `if`-satsen (sid 44) avgörs valet av ett enda villkor. Men till skillnad från `if`--satsen utförs ett andra alternativ (och inte ingenting) om villkoret är falskt. Därav *tvåvägsval*.

Flödesplan:



`if-else`-satsen är efter `if`-satsen den andra kontrollstruktur vi lär känna. Medan flödesplanen åskådliggör dess logiska struktur, visar pythonkoden dess syntax:

Pythonkod:

```
if villkor:
    Alternativ 1
else :
    Alternativ 2
```

Alternativ 1 och alternativ 2 är två olika satser eller två olika uppsättningar av satser. Endast ett av dem kommer att utföras beroende på villkorets sanningsvärde. Eftersom sanningsvärdena sant och falskt utesluter varandra, utesluter även de båda alternativen varandra.

Observera att indragningarna i kodens andra och fjärde rad tillhör syntaxen och inte får utelämnas. Dessutom främjar de kodens läslighet. Indragningarna avgör när satsens `if`- resp. `else`-del avslutas. Första raden som skrivs efter `if`-else-satsen utan indragning, tillhör inte längre `if-else`-satsen.

Det egentliga jobbet – nämligen att avgöra mellan jämnt och udda – görs av modulooperatorn **%** i rad **8**:

```
if tal % 2 == 0 :
```

Dvs: Om *resten vid heltalsdivision av* `tal` *med* `2` *är lika med* `0`

Alla jämna `tal` ger resten `0` vid heltalsdivision med `2`. Alla udda tal ger vid heltalsdivision med `2` resten `1` . Läs mer om modulooperatorn **%** på nästa sida.

Observera att hela `if-else`-satsen endast har ett villkor men två alternativ. Villkoret står i `if`-delen och upprepas inte i `else`-delen. Efter `else` står inte något annat villkor utan endast kolon: `else :`

Man kan säga att `else`-delen automatiskt får den logiska negationen av villkoret som redan står i `if`-delen. I det här fallet vore det `tal % 2 != 0`. I så fall identifierades `tal` som udda om resten av dess division med 2 inte vore `0`, dvs om `tal` inte vore jämnt delbart med 2. Här följer mer information om modulo:

Modulooperatorn

I programmet **`Delbarhet_2`** (sid 46) använde vi modulooperatorn **`%`** första gången på delbarhet. Men modulo har många andra intressanta användningar. Operatorn är besläktad med heltalsdivision vars symbol i Python är **`//`**. Vi öppnar Pythons Interactive mode för att testa den:

```
>>> 9 / 4
2.25
>>> 9 // 4
2
>>> 9 % 4
1
>>>
```

`/` är i Python symbolen för den vanliga divisionen som t.ex. ger ett decimaltal, för 9 är inte jämnt delbar med 4. Operatorn **`//`** däremot *heltalsdividerar* 9 med 4, dvs returnerar heltalsdelen 2, ignorerar resten 1 och går inte vidare till decimalerna. Just denna rest 1 tar modulooperatorn **`%`** hand om: Den heltalsdividerar också 9 med 4, men ignorerar resultatet 2 och returnerar endast resten 1. I matte skriver man: 9 mod 4 = 1.

Operationen modulo kan definieras som *resten vid heltalsdivision*. Den har koden **`%`** i Python och har samma prioritet som division (sid 22). En enkel, rolig användning av modulo är följande uppgift:

> Idag är det fredag, och du vill träffa din kompis om 11 dagar.
> Vilken veckodag blir det?

Vi numrerar veckodagarna stigande från 0 med början på söndag så att fredag blir veckodag 5. Uppgiftens lösning får du genom att addera 5 med 11, vilket ger 16, men räkna modulo 7 (antalet veckodagar). Det betyder att dra av 7 så många gånger det går: 16 - 7 - 7 = 2. Veckodag 2 är tisdag. Så du kommer att träffa din kompis på tisdag. I Python gör operatorn **`%`** jobbet åt oss:

```
>>> (5 + 11) % 7
2
>>>
```

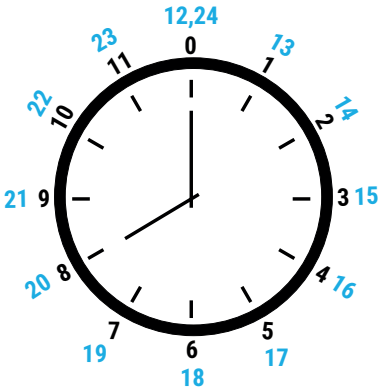
Vi skriver ett uttryck i Python: till aktuell veckodag 5 (fredag) adderas antalet dagar 11. Summan (inom parentes) räknas modulo 7, dvs 16 % 7. Python svarar med 2, vilket motsvarar tisdag. I matte skriver man:

$$(5 + 11) \ mod \ 7 \ = \ 16 \ mod \ 7 = 2$$

I själva verket handlar det om en omvandling av det decimala talsystemet med basen 10 och siffrorna 0-9 – det talsystem vi är vana vid att räkna med – till veckodagarnas system dvs till talsystemet med basen 7 och siffrorna 0-6. Därför lämpar sig modulooperatorn för att programmera omvandling av talsystem med olika baser.

Ytterligare exempel på användning av modulo är vår vanliga klocka som alltid ”räknar modulo 12” – ett talsystem med basen 12 och siffrorna 0-11. T.ex.:

```
15 % 12 = 3
18 % 12 = 6
21 % 12 = 9
24 % 12 = 12 % 12 = 0
```



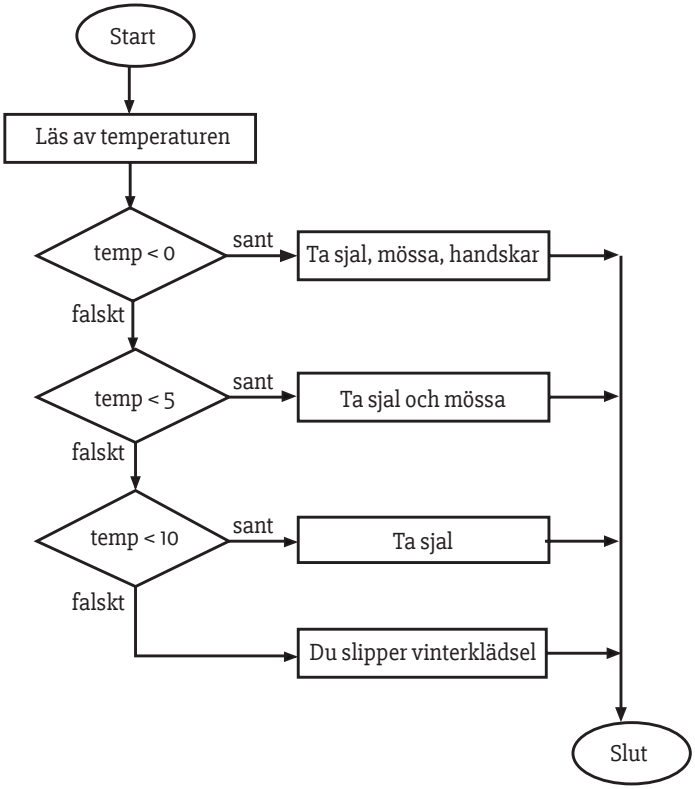
`if-elif-else`-satsen

`if-elif-else`-satsen kodar ett val mellan *flera* alternativ, även kallat *flervägsval*. Ett specialfall av den är `if-else`-satsen, *tvåvägsvalet*. Låt oss titta på följande algoritm (*Vinterklädsel*) som innehåller ett val mellan fyra alternativ. På de följande sidorna formulerar vi den som *pseudokod*, *flödesplan* och *pythonprogram*. Alla tre är olika former av beskrivning av en och samma algoritm. De första två används ofta med fördel inom programmering för att logiskt kartlägga och planera problemets lösning inför den slutliga kodningen. De anses vara lämpliga verktyg för programmering, när problemets logiska komplexitet växer.

Pseudokod:

```
Start Vinterklädsel
Läs av temperaturen
OM temperatur < 0
    ta sjal, mössa och handskar
ANNARS
    OM temperatur < 5
        ta sjal och mössa
    ANNARS
        OM temperatur < 10
            ta sjal
        ANNARS
            slipper du vinterklädsel
Slut Vinterklädsel
```

Flödesplan:



Pythonprogram:

```
1  # Vinterklädsel.py
2  # Läser in ett värde för temperatur, avgör val av klädsel
3  # Flervägsval: if-elif-else-satsen
4
5  temp = int(input('\n\tMata in temperatur:\t\t'))
6
7  if temp < 0 :                                  # Om...
8     print('\n\tTa sjal, mössa och handskar!\n')
9  elif temp < 5 :                                # Annars om...
10     print('\n\tTa sjal och mössa!\n')
11 elif temp < 10 :                               # Annars om...
12     print('\n\tTa sjal!\n')
13 else :                                         # Annars...
14     print('\n\tDu slipper sjal, mössa och handskar.\n')
```

Strukturen av `if-elif-else`-satsen framgår av programmet ovan. Jämför den med flödesplanen och pseudokoden på förra sidan. Observera även Pythons syntax inklusive indragningarna som hör till koden.

Här följer några körexempel för de fyra relevanta fallen:

```
Ange temperatur:          -5

Ta sjal, mössa och handskar!
```

```
Ange temperatur:          3

Ta sjal och mössa!
```

```
Ange temperatur:          8

Ta sjal!
```

```
Ange temperatur:          15

Du slipper sjal, mössa och handskar.
```

Övningar

1401

Försök att i Interactive mode beräkna följande uttryckets värde för $x = -9$.
Tolka Pythons svar.

$$\frac{87 + 13}{(x + 9) / 5}$$

1402

Ta över beräkningen av uttrycket i övn **1401** till ett program i en fil som beräknar uttryckets värde när man matar in ett värde för *x*. Bygg in `if`-satser som förhindrar division med `0` om man matar in `-9` för *x* på liknande sätt som i programmet **`Division_0`** (sid 42).

1403

Modifiera lösningen till övn **1402** ovan genom att ersätta `if`-satserna med en `if-else`-sats. Annars ska det nya programmet göra samma sak och ge samma utskrift som det gamla.

1404

a) Marcus som är 1,75 m stor och väger 76 kg vill veta om han är överviktig. Enligt Body Mass Index (BMI) anses man vara överviktig om BMI > 25. Testa i Interactive mode om Marcus är överviktig med formeln:

$$\text{BMI} = \frac{\text{Vikt i kg}}{\left(\text{Längd i m}\right)^2}$$

b) Skriv ett program (BMI Calculator) som läser in vikten i kg och längden i cm som heltal och skriver ut *Överviktig* om BMI > 25 annars *OK*. Som kontroll skriv även ut BMI-värdet.

c) Gör samma sak som i b), men läs in längden i *meter* som decimaltal. För typomvandling använd funktionen **`float()`** (sid 37).

1405

Skriv ett program som läser in tre heltal, hittar och skriver ut det största av dem. Vilken ändring i koden leder till det minsta talet?

1406

Testa i Python om 4592 är jämnt delbart med 7. Motivera. Generalisera:
Skriv en pythonkod som avgör om a är jämnt delbart med b.

1407

Testa följande algoritm i Pythons Interactive mode för flera startvärden:

1. Välj ett positivt heltal n.
2. Bilda n:s siffersumma och ev. siffersummans siffersumma.
3. Beräkna n % 9 och jämför med 2.

Beskriv dina observationer.

1408

Koda följande algoritm i Python:

1. Läs in ett positivt heltal n.
2. Beräkna $d = (n + 1)(n - 1)$
3. Om d är jämnt delbart med 8 skriv ut YES annars NO.

Testa programmet med 6 udda och 6 jämna startvärden. Beskriv resultatet.

Kan resultatet generaliseras? Om ja, bevisa det.

1409

Vilka värden kommer variablerna `tal` och `mod` att ha efter följande kod?

```
tal = 32
d = 6
mod = tal % d
tal = tal - d - d - d - d - d
```

Svara först utan Python. Testa sedan i Interactive mode. För vilken matematisk kunskap är koden ett exempel på?
Svara först. Testa sedan i Python.

1410

Idag är det onsdag. Julia vill träffa sin kompis om 13 dagar och vill veta vilken veckodag det blir. Lös problemet generellt:

Skriv ett program som frågar efter aktuell veckodag. Mata in en siffra för veckodagen. Numrera veckans dagar stigande från 1-7 med början på måndag. Sedan ska programmet fråga när användaren vill träffa din kompis och få som svar ett antal dagar. Beräkna och skriv ut den planerade träffens veckodag som nummer.

1411

Koda följande algoritm:

```
1. Läs in tre heltal till timmar,
   minuter och sekunder.
2. Omvandla allt till totalsekun-
   der. Skriv ut totalsek.
3. Ta sekunderna från steg 2.
4. Omvandla tillbaka till timmar,
   minuter och sekunder.
5. Skriv ut tim, min, sek.
```

Gör utskriften användarvänlig. Testa ditt program t.ex. för 7 timmar, 58 minuter och 34 sekunder.

1412

Koda följande algoritm:

```
1. Läs in tre heltal till år, må-
   nader, veckor och dagar.
2. Omvandla allt till totaldagar
   och skriv ut resultatet.
3. Ta totaldagarna från steg 2.
4. Omvandla tillbaka till år, må-
   nader, veckor och resterande
   dagar.
5. Skriv ut år, månader, veckor
   samt resterande dagar.
```

Gör utskriften användarvänlig. Testa ditt program t.ex. för 2 år, 11 månader, 3 veckor och 6 dagar.

1413

Skriv ett program som läser in begynnelsebokstaven till en veckodag, med `if-elif-else`-satsen bestämmer vilken veckodag det är och skriver ut den.

Fixa problemet med tisdag/torsdag genom att bygga in en `if-else`-sats ifall det matas in `'t'`. Läs in och bearbeta i så fall den andra bokstaven.

Ta även hand om felaktig inmatning: Om det matas in en bokstav som inte är en veckodags första eller – ifall av tisdag/torsdag – andra bokstav, skriv ut t.ex. *Detta är ingen veckodag.*

1.5 Gissa tal – ett spel

Här introduceras ett litet enkelt spel som i fortsättningen kommer att utvecklas steg för steg. I varje version kommer vi att lära oss ett nytt programmeringskoncept. Låt oss kalla speltet för *Gissa tal*. Användaren ska gissa fram ett hemligt tal inom ett visst intervall. Som hjälp får användaren reda på om det gissade talet är mindre än, större än eller lika med det hemliga talet. Därför är spelet ett exempel på ett *trevägsval* som vi lärde oss koda med `if-elif-else`-satsen i förra avsnitt. Önskemålet är att kunna spela om spelet tills man gissat rätt. Detta kommer att leda till att vi lär oss *loopar* i Python. För att kontrollera loopars korrekta avslutning behöver man kunskaper i logik – en viktig ingrediens i programmering. Till en början kan det hemliga talet vara hårdkodat. Sedan kan vi gå över till att generera det hemliga talet slumpmässigt. Så kommer vi att lära oss Pythons *hantering av slumptal*. Vi börjar med:

Gissa tal, ver 1

```
1   # GissaTal_1.py
2   # Användaren ska gissa programmets hemliga tal (hårdkodat)
3   # genom att få hjälpen: rätt, mindre eller större
4
5   secret = 17              # Programmets hemliga tal
6   guess = int(input('\n\tGissa ett tal mellan 1 och 20:\t'))
7
8   if guess == secret :     # Om man gissat rätt
9      print('\n\tGRATTIS, du har gissat rätt!\n')
10  elif guess < secret :    # Annars om guess är mindre
11     print('\n\tFel:', guess, '< hemliga talet.\n')
12  else :                   # Annars om inte rätt, inte mindre
13     print('\n\tFel:', guess, '> hemliga talet.\n')
```

`if-elif-else`-satsens inbyggda logik gör att programmet blir enkelt. Jämför med den logiska struktur som flödesplanen för algoritmen *Vinterklädsel* visar (sid 50). Körexempel med de tre relevanta fallen ger:

```
        Gissa ett tal mellan 1 och 20:  13

        Fel: 13 < hemliga talet.
```

```
Gissa ett tal mellan 1 och 20:  19
Fel: 19 > hemliga talet.
```

```
Gissa ett tal mellan 1 och 20:  17
GRATTIS, du har gissat rätt!
```

`while`-satsen

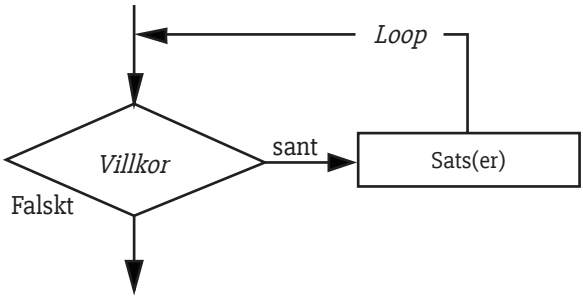
Önskemålet är ju att kunna spela flera omgångar. För att bygga in detta i version 2 öppnar vi här en parentes om *loopar*. Loop är det engelska ordet för slinga.

I programmering används för upprepade beräkningar kontrollstrukturen *repetition* *. Beroende på hur repetitionen formuleras skiljer man mellan olika typer av repetition. En av dem är `while`-satsen vars pythonkod och flödesplan visas nedan. Flödesplanen återspeglar kontrollstrukturens logiska struktur. Den visar att ett vilkor testas *före* satserna. Därför kallar man `while`-satsen även för *för-testad repetition*. Den inleds med `while`. Så länge (eng. *while*) villkoret är uppfyllt utförs satserna.

Pythonkod

`while` *villkor* **:**
Sats(er)

Flödesplan



Pythonkodens första rad kallas för `while`-satsens *huvud*, resten är *kroppen*. Endast om *villkoret* är uppfyllt utförs kroppen. Man kan skriva kroppen på samma rad som huvudet direkt efter kolonet (`:`). Men om man skriver den på en ny rad, vilket man borde göra för bättre läslighetens skull, måste hela kroppen stå indragen – regler som vi redan stött på i kontrollstrukturerna `if` och `if-else`.

* Ibland kallas repetition även för *iteration* som är en term inom numerisk analys som sysslar med lösning av matematiska problem som inte (eller svårt) kan lösas analytiskt. En besläktad term är *rekursion* som behandlas senare (sid 98).

Precis som där gäller att indragningen i kodens andra rad tillhör syntaxen och inte får utelämnas. Indragningen avgör när `while`-satsen avslutas. Allt som man vill ha i `while`-satsen måste dras in.

Villkoret måste vara sant i början för att komma in i loopen. Därför måste den initieras korrekt innan loopen. Sedan måste villkorets sanningsvärde ändras inuti kroppen så att loopen kan avslutas någon gång. Annars blir det en evighetsloop. Loopen måste alltså ha ett fungerande avslutningskriterium.

Gissa tal, ver 2

```
1   # GissaTal_2.py
2   # Version 2 omges av en loop:
3   # while-sats används för att kunna gissa flera omgångar
4   # Logisk variabel styr loopen. Antal försök räknas.
5
6   secret = 17                     # Programmets hemliga tal
7   guessNo = 0                     # Antal försök initieras
8   wrongGuess = True               # Logisk variabel initieras
9
10  while wrongGuess :              # Så länge fel gissat (loop)
11     guess=int(input('\n\tGissa ett tal mellan 1 och 20:\t'))
12     if guess == secret :
13        print('\n\tGRATTIS, du har gissat rätt efter',
14                                          guessNo, 'försök.\n')
15        wrongGuess = False        # Bryter loopen: rätt gissat
16     elif guess < secret :
17        print('\n\t\t',guess,'< hemliga talet. Försök igen!')
18     else :
19        print('\n\t\t',guess,'> hemliga talet. Försök igen!')
20     guessNo = guessNo + 1        # Antal försök uppdateras
```

I rad **10** står `while`-satsens huvud som avslutas med kolon. Där börjar loopen. Resten är kroppen som består av indragna satser (rad **11-20**) som alla tillhör `while`-satsen. Loopen avslutas först när programmet avslutas eftersom det inte finns någon icke-indragen rad efter **10**. Loopen går från rad **10** till slutet av programmet.

`while`-satsens villkor är **`wrongGuess`** (rad **10**), en s.k. *logisk variabel* eller en variabel av typ *boolean*. Villkoret måste vara sant för att komma in i loopen. Därför initieras **`wrongGuess`** innan loopen i rad **8** till **`True`**. I loopen ändras villkorets sanningsvärde till **`False`** (rad **15**). Då avslutas loopen.

Logiska variabler

I programmet `GissaTal_2` initieras variabeln **`wrongGuess`** till värdet `True`:

```
8     wrongGuess = True              # Logisk variabel initieras
```

Variabler vars värden är `True` eller `False`, kallas för *logiska variabler*. Själva `True` och `False` är logiska konstanter och de enda värden som datatypen *boolean* har. Det är därför vi kan använda logiska variabler som villkor i `while`-satsen:

```
10    while wrongGuess :             # Så länge fel gissat (loop)
```

De är dessutom utbytbara mot heltal: `False` står för `0` och `True` för alla heltal $\neq$ `0`. Eftersom vi i rad `8` initierat **`wrongGuess`** till `True` får vi inträde i loopen. Inuti loopen i rad `15` får denna variabel det nya värdet `False`:

```
15          wrongGuess = False     # Bryter loopen: rätt gissat
```

Det nya värdet gör att vi efter denna ändring inte längre får inträde i loopen. `while`-satsen bryts och därmed avslutas programmet. Placeringen av denna ändring i `if` med villkoret `guess == secret` gör att ändringen endast sker när spelaren gissat rätt. Efter rätt gissning ska ju spelet avslutas. Om däremot spelaren gissar fel behåller **`wrongGuess`** värdet `True` och `while`-satsen fortsätter att loopa.

Och det är just så vi vill ha det: Loopen ska låta oss spela en omgång till när vi gissar fel. Då kommer vi även till `while`-satsens sista del, `if-elif-else`-satsen, där vi får chansen att gissa vidare. Informationen om vi med vår gissninng ligger under eller över programmets hemliga tal, hjälper oss att i nästa försök gissa bättre, tills vi gissar rätt.

En annan nyhet i version 2 är att vi, när vi gissat rätt, får reda på hur många gånger vi gissat fel. Antalet felförsök tas hand om av variabeln **`guessNo`** som initieras till `0` innan loopen (rad `7`). I loopen skrivs den ut när vi gissat rätt (rad `14`) och uppdateras med `1` (rad `20`). I nästa version låter vi Python slumpa fram programmets hemliga tal:

Gissa tal, ver 3

```
1   # GissaTal_3.py
2   # Version 3 förses med slumptal
3   # Slumpar fram ett hemligt tal mellan 1 och 100
4   # Kan avslutas innan man gissat rätt: Hemliga talet visas
5
6   import random                # Importerar modulen random
7                                # inneh. funktionen random()
8   secret = 1 + int(random.random() * 100)    # Slumptal
9   print('\n\tProgrammets hemliga tal är mellan 1 och 100.')
10
11  guessNo = 0                  # Antal försök initieras
12  wrongGuess = True            # Logisk variabel initieras
13
14  while wrongGuess :           # Så länge fel gissat
15     guess = int(input('\n\tGissa vilket heltal',
16                                      '(Avsl. med 0):  '))
17     if guess == 0 :           # Om man vill avsluta
18        print('\n\tAvbrott: Programmets hemliga tal var\t',
19                                             secret, '\n')
20        wrongGuess = False     # Bryter loopen: vill sluta
21     elif guess == secret :
22        print('\n\t\tGRATTIS, du har gissat rätt efter',
23                                        guessNo, 'försök.\n')
24        wrongGuess = False     # Bryter loopen: rätt gissat
25     elif guess < secret :
26        print('\n\t\t', guess, '< hemliga talet.',
27                                          \tFörsök igen!')
28     else :
29        print('\n\t\t', guess, '> hemliga talet.',
30                                          \tFörsök igen!')
31     guessNo = guessNo + 1      # Antal försök uppdateras
```

En nyhet i spelets version 3 är att det hemliga talet genereras slumpvis (rad 8). För att göra spelet lite intressantare har vi också utökat gissningsintervallet från `[1, 20]` till `[1, 100]` (rad 9). Den utförliga behandlingen av slumptal tas upp i nästa avsnitt (sid 63). En annan nyhet är att man kan, om man vill, avsluta spelet innan man gissat rätt och får reda på det hemliga slumptalet.

Vi avslutar detta avsnitt med en diskussion om hur man bäst kan spela Gissa tal, dvs hur man kan gissa för att minimera antalet felförsök. I centrum står är en algoritm som visar sig vara användbar även i andra sammanhang:

Algoritmen Intervallhalvering

Finns det en optimal strategi för att med så få försök som möjligt gissa rätt i Gissa tal-spelet? Vi ska presentera här en enkel algoritm – den här gången inte för att skriva ett program – utan för att spela mot programmet `GissaTal_3` så effektivt som möjligt. Man skulle kunna även programmera denna algoritm och låta de två programmen tävla mot varandra eller låta användaren ange ett hemligt tal och låta programmet hitta det. Det får förbli en utmaning för intresserade utvecklare. Vår algoritm *Intervallhalvering* går ut på att minimera antalet försök för att gissa rätt i Gissa talspelet. Här följer en beskrivning:*

1. Starta i ett intervall som innehåller det hemliga talet.
2. Halvera intervallet.
3. Välj den halvan av intervallet som innehåller det hemliga talet.
4. Gå tillbaka till steg 2 och upprepa 2-4 tills hemliga talet hittats.

(Rekursion)

Problemet behöver inte vara Gissa tal-spelet utan något annat problem, t.ex. en ekvation. Och ”det hemliga talet” kan vara ekvationens lösning. Ja, lösningen behöver inte ens vara heltal. Den kan vara decimaltal. Algoritmen kommer att fungera ändå. Den är så pass generell att den t.o.m. kan användas som en numerisk metod för ekvationslösning, vilket kommer att tas upp i kursen Matematik 2.

Men tillbaka till Gissa tal. Följande körexempel ska demonstrera algoritmen Intervallhalvering. Vi startar programmet `GissaTal_3` och får följande utskrift:

```
Programmets hemliga tal är mellan 1 och 100.

Gissa vilket heltal (Avsl. med 0):
```

För att minska arbetet som krävs för att leta efter det hemliga talet som ligger mellan `1` för `100`, halverar vi intervallet `[1, 100]` och matar in `50` som vår gissning:

```
Programmets hemliga tal är mellan 1 och 100.

Gissa vilket heltal (Avsl. med 0):  50

         50 > hemliga talet.    Försök igen!

Gissa vilket heltal (Avsl. med 0):
```

Vi får informationen att `50` är större än det hemliga talet. Alltså måste det hemliga

* Jämför med algoritmen Primtalsfaktorisering (sid 95) och läs om *rekursion* på sid 98.

talet ligga i intervallet [1, 49]. Vi halverar nu intervallet [1, 49] och matar in 25:

```
Gissa vilket heltal (Avsl. med 0):  25

         25 < hemliga talet.    Försök igen!

Gissa vilket heltal (Avsl. med 0):
```

Slutsats: det hemliga talet måste ligga i intervallet [26, 49]. Halvering av detta intervall ger 37,5. Eftersom vi måste hålla oss till heltal matar vi in, t.ex. 37:

```
Gissa vilket heltal (Avsl. med 0):  37

         37 < hemliga talet.    Försök igen!

Gissa vilket heltal (Avsl. med 0):
```

Nu vet vi att det hemliga talet ligger i intervallet [38, 49]. Halvering av detta intervall ger 43,5. Vi matar in t.ex. 44:

```
Gissa vilket heltal (Avsl. med 0):  44

         44 < hemliga talet.    Försök igen!

Gissa vilket heltal (Avsl. med 0):
```

Det hemliga talet måste ligga i intervallet [45, 49]. Mittpunkten är 47:

```
Gissa vilket heltal (Avsl. med 0):  47

         47 < hemliga talet.    Försök igen!

Gissa vilket heltal (Avsl. med 0):
```

Vilken rysare! Det hemliga talet kan bara vara 48 eller 49. Vi chansar med 48:

```
Gissa vilket heltal (Avsl. med 0):  48

         48 < hemliga talet.    Försök igen!

Gissa vilket heltal (Avsl. med 0):
```

Nej! Är det sant? Nu kan det hemliga talet bara vara 49:

```
        Gissa vilket heltal (Avsl. med 0):  49

                GRATTIS, du har gissat rätt efter 6 försök.

Press any key to continue . . .
```

Vi hade extrem otur att det hemliga talet 49 låg så knappt under 50. Ändå klarade vi oss ganska bra med bara 6 försök, vilket heltalsvariabeln **guessNo** i programmet **GissaTal_3** tar hand om (raderna **11**, **23** och **31**). Å andra sidan var det extrema exemplet intressant som demonstration.

Ett annat körexempel som är lite bättre lottat visas sammanhängande nedan. Genom att använda algoritmen Intervallhalvering kan man klara sig med endast 4 försök. Här är beviset:

```
        Programmets hemliga tal är mellan 1 och 100.

        Gissa vilket heltal (Avsl. med 0):  50

                50 < hemliga talet.    Försök igen!

        Gissa vilket heltal (Avsl. med 0):  75

                75 < hemliga talet.    Försök igen!

        Gissa vilket heltal (Avsl. med 0):  87

                87 > hemliga talet.    Försök igen!

        Gissa vilket heltal (Avsl. med 0):  81

                81 < hemliga talet.    Försök igen!

        Gissa vilket heltal (Avsl. med 0):  84

               GRATTIS, du har gissat rätt efter 4 försök.
```

Övningar

1501

a) Använd en loop med `while`-satsen för att skriva ut de första 10 positiva heltalen.

b) Vilken ändring i koden till a) måste göras för att få fram de första 20 positiva heltalen?

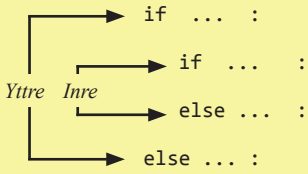
1502

a) Skriv ett program som skriver ut de första 10 jämna talen.

b) Modifiera a) så att endast de första 10 udda talen skrivs ut.

1503

I programmet **GissaTal_1** (sid 54) används en `if-elif-else`-sats för att koda trevägsval. Ett sådant val kan även kodas med en nästlad `if-else`-sats som har följande struktur:



Modifiera programmet **GissaTal_1** så att `if-elif-else`-satsen ersätts av en nästlad `if-else`-sats.

1504

a) Skriv ett program som summerar de första 10 positiva heltalen.

b) Generalisera a) så att programmet beräknar summan av de första n positiva heltalen där n kan matas in. Testa för $n = 100$ och $1\,000$.

c) Skriv ett program som summerar de första n pos. heltalen med formeln:

$$summa = n\,(n+1)\,/\,2$$

Testa om du får samma svar i b) och c) för $n = 1\,000$, $5\,000$ och $1\,000\,000$.

1505

Koda följande algoritm till ett program:

```
Läs in ett positivt heltal
Skriv ut talet
Så länge talet ≠ 1 REPETERA:
   OM talet är udda
      multiplicera med 3, addera 1
   ANNARS
      dividera talet med 2
   Skriv ut talet
```

Testa programmet genom att läsa in heltalen 3, 6, 7, 13 och 50. Ange slutresultaten. Testa gärna fler startvärden. Studera de talföljderna du får. Finns det någon förklaring för slutresultaten? *

1506

Koda följande algoritm till ett program:

```
Läs in två positiva heltal a och b
 Så länge a ≠ b REPETERA:
   OM a > b
      a = a - b
   ANNARS
      b = b - a
 Skriv ut a
```

Testa för 48 och 60. Vad gör algoritmen?

* Känt som (3n+1)-problemet som hittills är obevisat! Det härstammar från den tyske matematikern *Lothar Collatz* (1910 - 1990) som var författarens lärare vid Hambrugs Universitet på 60-talet.

1.6 Hantering av slumptal

```
random.random() ger slumptal mellan 0 och 1:
              0.4283006803438053
              0.5105081765443273
              0.5035213041398865
1 + int(random.random() * 20) ger slumptal
mellan 1 och 20:
[1, 2, 2, 2, 2, 3, 3, 4, 5, 5, 5, 7, 7, 7, 7,
7, 7, 8, 8, 9, 10, 10, 10, 10, 11, 11, 11,
11, 12, 13, 13, 14, 15, 15, 15, 16, 16, 16,
18, 18, 18, 18, 18, 18, 18, 19, 20, 20, 20]
```

Teoretiskt kan man med datorn som en deterministisk maskin inte producera äkta slumptal. Man kan bara *simulera* slumptal genom att enligt en viss algortim *beräkna* tal i en följd, vilket – strikt talat – inte ger slumptal utan endast *pseudo*slumptal.

Som vi såg i programmet `GissaTal_3` kan man göra det i Python med funktionen **`random()`** som är definierad i modulen `random` (sid 58, rad 8). Denna funktion slumpar decimaltal mellan `0` och `1`, närmare bestämt från och med `0` till, men inte med, `1`. Det vill säga:

```
0 <= random.random() < 1
```

De första raderna i utskriften ovan demonstrerar detta. Ur användningssynpunkt är det dock slumptal i decimalform inte särskilt relevanta. Oftast vill man ha heltal och dessutom själv kunna bestämma inom vilka gränser talen ska vara.

För att skräddarsy pythonfunktionen **`random()`**, t.ex. slumpmässigt få heltal mellan `1` och `20` gör vi i formeln nedan en *skalning* med faktor `20`, en omvandling till heltal och en *skiftning* med `1`, för att även få med gränserna `1` och `20` i slumptalen:

```
1 + int(random.random() * 20)
```

Följande program testar både funktionen **`random()`** och vår transformation till heltalsintervallet `[1, 20]`. Programmet producerar även utskriften ovan:

```
1   # Slumptal.py
2   # Slumpar decimaltal mellan 0 och 1 med funktionen random()
3   # Slumpar heltal mellan 1 och 20 med en skräddarsydd variant
4   #                                                (se rad 15)
5   import random              # Importerar modulen random som
6                              # innehåller funktionen random()
7   print('\n   random.random() ger slumptal mellan 0 och 1:\n')
8   for i in range(1, 4) :
9       print('\t\t', random.random())
10
11  print('\n   1 + int(random.random() * 20) ger slumptal',
12        '\n   mellan 1 och 20:\n')
```

```
13  slumplista = [ ]            # Skapar en tom lista
14  for n in range(1, 50) :     # För alla n från 1 till 50 gör:
15      slumplista = slumplista + [1 + int(random.random()*20)]
16  print(sorted(slumplista), '\n') # Lista skrivs ut sorterad
```

Skalningen `random.random()*20` förstorar slumpvärdena och ger decimaltal från och med `0` till, men inte med, `20.` Omvandlingen med `int()` ger heltal mellan `0` och `19`. Slutligen ger skiftningen `1 +` … en förskjutning av intervallet `[0, 19]` till `[1, 20]` och därmed det önskade resultatet, nämligen slumptal mellan `1` och `20` inkl. gränserna.

`for`-satsen

I programmet `Slumptal` används en annan variant av loop än `while`-satsen (sid 55), nämligen `for`-satsen (rad `8-9` och `14-15`). Dess huvud i rad `14` ser ut så här:

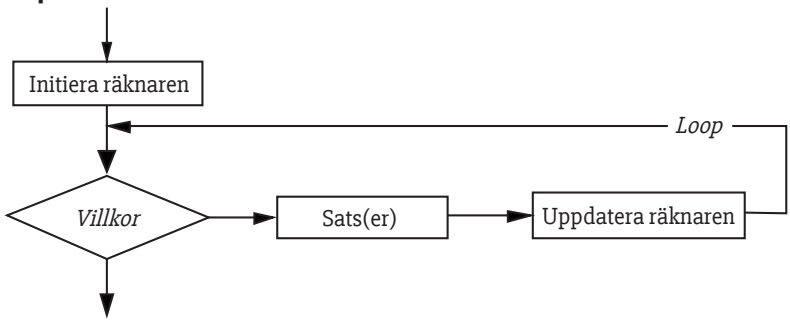
```
14    for n in range(1, 50) :
```

n är loopens *räknare*, även kallad *styr-* eller *kontrollvariabel*, tar hand om antalet repetitioner. `range(1, 50)` är en pythonfunktion som ger `True` så länge **n < 50**, annars `False`. I början initieras **n** till **1** och uppdateras i varje varv av loopen automatiskt (ökas med 1), vilket fungerar som ett inbyggd villkor. I `while`-satsen styr ett egetskrivet villkor antalet repetitioner. I `for`-satsen kan vi explicit ange antalet repetitioner som är känt i förväg. Därför kallas denna typ av repetition för den *bestämda repetitionen*. Nedan följer `for`-satsens pythonkod samt flödesplan som visar dess logiska struktur. Det hela ska läsas med preciseringen på nästa sida.

Pythonkod:

```
for räknare in range (startvärde, slutvärde)
    Sats(er)
```

Flödesplan:



Precisering av `for`-satsens flödesplan:

- Initiera räknaren till *startvärdet*.
- *Villkor:* Ligger räknaren i intervallet *(startvärde , slutvärdet)*, närmare bestämt: Är räknaren < slutvärdet ?
- Uppdatera räknaren: räknare = räknare + 1
- Loopen: så länge *villkoret* är uppfyllt utför satserna, annars lämna `for`-satsen.

`for`-satsens struktur är alltså mer komplicerad än `while`-satsens. I princip utför båda samma sak. Man kan koda repetitioner i regel med vilken loopvariant som helst. I programmet **`Slumptal`** blir koden faktiskt kortare och enklare att följa med `for`-satsen, eftersom räknarens hantering i `for` är automatiserad till skillnad från `while`.

Allt vi sade om indragningar i koden, när vi behandlade `while`-satsen (sid 55), gäller förstås även här. Av denna anledning ingår t.ex. **`print()`**-satsen i rad **`16`** (sid 64) inte i `for`-satsen utan står utanför den och utförs därför bara en gång.

Listor i Python

En lista kallas den form som t.ex. slumptalen mellan `1` och `20` i utskriften på sid 63 står i. I programmet **`Slumptal`** skapas denna lista med följande kod:

```
13    slumplista = [ ]                    # Skapar en tom lista
```

Samtidigt initieras den till en tom lista **`[ ]`**, kallad **`slumplista`** som bara är ett *namn* som vi gett till denna variabel.

Lista är en s.k. *sammansatt* datatyp i Python. Den kallas så eftersom den representerar *fler* än ett värde åt gången, t.ex. *flera* **`int`**, *flera* **`float`**, *flera* **`string`** osv. som i sin tur kallas för *enkla* datatyper. En lista däremot kan vara sammansatt av olika enkla datatyper.

I varje varv av `for`-satsen läggs ett nytt element i den tomma listan **`slumplista`**:

```
15    slumplista = slumplista + [1 + int(random.random()*20)]
```

Detta görs med **+** som är en operator definierad i datatypen lista. När vi diskuterade datatyper (sid 37) sade vi att **+** betydde addition i räknesammanhang, men sammanslagning (konkatenering) i strängsammanhang. Nu har vi en tredje betydelse för **+** , nämligen *konkatenering av listor*. Koden ovan tillfogar slumptal mel-

lan 1 och 20 till den redan befintliga listan **slumplista**. Innan dess (rad 13) hade vi initierat den till en tom lista. I for-satsen blir den nu successivt ifylld med slumptal. Så får vi slutligen den lista över slumptal som vi ser i utskriften på sid 63.

import-satsen

I början av programmen GissaTal_3 (sid 58) och Slumptal (sid 63-64) förekommer efter kommentarraderna satsen:

```
import random
```

Här ges Python interpretatorn instruktionen att ladda modulen random ur Pythons bibliotek till vårt program så att vi kan anropa funktioner som är definierade i den.

En *modul* är en samling funktioner som är fördefinierade i Python. Vill man använda dessa funktioner i sitt eget program måste man importera modulen med import-satsen från Pythons programbibliotek.

Den funktion som vi flera gånger anropar i programmen ovan är **random()**, t.ex. i programmet Slumptal (rad 9 och 15). Denna funktion är definierad i modulen random. Utan import-sats skulle dessa anrop inte fungera.

Slumptal i önskat intervall

```
I vilket intervall vill du ha slumptalen?
Ange intervallets början (pos. heltal):  456
Ange intervallets slut   (pos. heltal):  12435
Hur många slumptal vill du ha?           50
Var så god!    50 slumptal mellan 456 och 12435 :
[821, 1413, 1430, 1442, 1551, 1968, 2109, 2262,
2916, 2996, 3400, 3858, 3970, 4076, 4309, 4920,
5219, 5233, 5633, 5858, 6004, 6053, 6142, 6155,
6283, 7217, 7413, 7857, 8120, 8359, 8859, 9252,
9450, 9627, 9858, 9865, 10192, 10335, 10397,
10438, 10672, 10773, 10932, 11423, 11546, 11547,
11683, 11929, 12305, 12331]
```

Vad gör man om man inte vill ha slumptal mellan **1** och **20** utan mellan **12** och **35** t.ex.? Ska man då skriva ett nytt pythonprogram för det?

Det vore väl inte bara slöseri med tiden utan även dålig stil. Man vill helst ha ett generellt program som slumpar heltal mellan vilka intervallgränser som man själv kan välja. Se körexemplet till vänster som skriver ut `50` slumptal mellan `456` och `12435`.

Transformationen från decimaltalsintervallet **`[0, 1]`** till heltalsintervallet **`[1, 20]`** utvecklades i förra avsnitt:

```
1 + int(random.random() * 20)
```

Den kan generaliseras: Vill man ha slumptal mellan a och b där a < b, kan man transformera decimaltal mellan `0` och `1` till heltal mellan a och b så här:

```
a + int(random.random() * (b-a+1))
```

Men om a > b, blir platserna på a och b i formeln ombytta vilket ger:

```
b + int(random.random() * (a-b+1))
```

Med dessa formler kan vi skriva ett program som slumpar heltal mellan a och b:

```
1   # MyRandom.py
2   # Slumpar ett önskat antal heltal i önskat intervall [a, b]
3
4   import random
5   print('\n  I vilket intervall vill du ha slumptalen?')
6   a = int(input(
7           '\n  Ange intervallets början (pos. heltal):  '))
8   b = int(input(
9           '\n  Ange intervallets slut   (pos. heltal):  '))
10  antal = int(input(
11                '\n  Hur många slumptal vill du ha?\t   '))
```

```
12  slumplista = [ ]
13  for n in range(1, antal + 1) :
14      if a < b :                          # Generella formler:
15         slumplista = slumplista +
16                      [a + int(random.random() * (b-a+1))]
17      else :
18         slumplista = slumplista +
19                      [b + int(random.random() * (a-b+1))]
20  print('\n  Var så god!\t', antal, 'slumptal mellan', a,
21                                          'och', b, ':\n')
22  print(' ', sorted(slumplista), '\n')
```

Programmet `MyRandom` ger körexemplet på förra sidan. Men vad händer om användaren matar in ett större värde för intervallets början än för slutet?

`if-else`-satsen som är inbakad i `for`-satsen (rad **14**-**19**) ordnar detta och väljer korrekt formel. Med båda generella formler (rad **16** och **19**) får man nu korrekt resultat, även om man anger ett större värde för intervallets början än för slutet. Ett annat körresultat visar detta:

```
I vilket intervall vill du ha slumptalen?

Ange intervallets början (pos. heltal):  78

Ange intervallets slut   (pos. heltal):  12

Hur många slumptal vill du ha?           50

Var så god!    50 slumptal mellan 78 och 12 :

[17, 17, 18, 22, 23, 24, 24, 24, 25, 26, 28, 29, 29, 29, 29, 30, 34, 38,
38, 39, 40, 41, 41, 42, 42, 43, 44, 46, 46, 51, 53, 55, 55, 57, 57, 61,
62, 63, 66, 67, 70, 70, 71, 71, 72, 74, 74, 75, 76, 77]
```

Övningar

1601

a) Skriv ett program som använder en loop med `for`-satsen för att skriva ut 10 slumptal mellan 0 och 1.

b) Skräddarsy Pythons funktion `random()` för att slumpa 20 heltal mellan 1 och 50. Bygg in den i ett program som skriver ut slumptalen.

1602

Vi vill simulera tärningskast.

a) Generera 10 slumptal mellan 1 och 6 och skriv ut dem.

b) Skapa först en tom lista i Python, lägg de 10 slumptalen från a) i listan och skriv ut den.

c) Skriv ut en lista som innehåller 500 tärningskast.

1603

Skriv ett program som räknar förekomsten (frekvensen) av de 6 slumptalen i övn **1602** c):s lista över 500 tärningskast. Skapa 6 dellistor och lägg i var och en endast samma slumptal. Skriv ut dellistornas längder. Är de ungefär lika stora? Ange ett exempel. Avsluta programmet med en kontroll genom att summera alla dellistornas längder, vilket borde ge 500.

1604

Skriv ett program som skriver ut endast var 10:e tal i heltalsintervallet [1, 5 000]. Låt programmet läsa in steget 10 som en variabel. Om steget är *n* ska vart *n*:te tal skrivas ut. Testa för olika *n*. Använd lista för att kunna se utskriften på skärmen.

1605

Den inbyggda pythonfunktionen `ord()` returnerar heltalskoden till en bokstav medan funktionen `chr()` returnerar bokstaven till en heltalskod. T.ex. är `ord('a') = 97` och `chr(97) = 'a'`. Använd dessa funktioner för att med en `for`-sats skriva ut det engelska alfabetets stora bokstäver A-Z. Mer info om `ord()` och `chr()` finns på sid 82.

1606

Familjen Pettersson tänker plundra sina tre spargrisar för att gå till Gröna Lund. De vill uppskatta hur mycket de kommer att få ihop. Troligen finns det mellan 90-120, 70-85 och 35-50 kr i varje spargris.

Den yngste sonen Max som läst *Koda matte med Python* vill simulera plundringen och tänker skriva ett program som använder sig av slumptal för de uppskattade grisvärdena för att beräkna ett närmevärde till den totala spargrisförmögenheten. Skriv programmet åt Max.

1607

En borrutrustning för bergvärme kan borra 25 m i en viss tomtmark under den första timmen. Under de följande timmarna minskar borrdjupt med uppskattningsvis 10-20 % för varje timme.

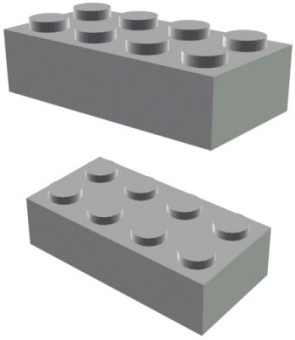
Skriv ett program som anger ett närmevärde till det totala borrdjupet om borren går oavbrutet i 8 timmar. Programmet ska använda slumptal för den uppskattade minskningen av utrustningens prestation efter den första timmen.

1.7 Funktioner i programmering

Ofta är man inte direkt intresserad av t.ex. slumptal utan vill använda dem som ett mellanled i ett större sammanhang, som ett medel för att uppnå ett högre mål. Låt oss säga att vi t.ex. vill skriva ett krypteringsprogram där krypteringsnyckeln är ett slumptal. Just detta kommer vi faktiskt att göra i nästa avsnitt. Då är fokuset att hitta en bra krypteringsalgoritm och implementera den. För att kunna göra det vill vi inte upprepa utvecklingsarbetet vi lagt på programmet `MyRandom` (sid 67). Istället vill vi *använda* det befintliga program vi en gång skrivit som en färdig *modul* och koncentrera oss på krypteringen. Dessutom kan vi återanvända samma modul även i andra, mer avancerade program. Tillvägagångssättet heter:

Modularisering eller Lego-principen

De flesta har väl någon gång lekt med Lego-bitar. Principen är enkel: Av små enkla moduler kan komplexa objekt byggas. Det omvända är också sant: Vill man bygga ett komplext objekt, kan man bryta ned det i ett antal mindre moduler. Sedan sätter man ihop de små enkla lösningarna till den stora, komplexa lösningen. Principen heter *modularisering* och används vid nästan all problemlösning. De mindre modulerna motsvarar Lego-bitarna. I programmering kallas dessa moduler för *funktioner*. Stora program bryts ned i ett antal funktioner. Varje funktion löser ett visst delproblem som är oberoende av andra, dessutom enklare att koda än det stora programmet. Sedan gäller det att sätta ihop modulerna till det stora programmet. Både *Modularisering* och *återanvändning av kod* är viktiga strukturerings- och modelleringsprinciper inom programvaruutveckling.



Funktionsbegreppet i programmering

Man känner till begreppet *funktion* från matematiken. Även i programmering finns den matematiska synen på funktion som underliggande koncept och historisk utgångspunkt. Men under IT:s frammarsch har den fått en bredare tolkning då den tillämpats på all datoriserad problemlösning. Så här kan begreppet *funktion* i programmering definieras:

> En funktion är ett antal satser vars kod definieras som en *separat, namngiven modul*, men utförs först när funktionen anropas. En funktion är en *föreskrift* om vad som skulle hända om funktionen skulle anropas. Det är en kodmodul i väntan på att bli anropad (aktiverad). Vid anropet kan funktionen ta emot indata, s.k. *parametrar*, bearbeta dem enligt sin definition och returnera utdata, det s.k. *returvärdet*.

Som ett ”antal satser” är en funktion en del av ett program som isoleras och t.o.m. kan skrivas i en separat fil, för att kunna anropas av flera olika program. Syftet är modularisering och återanvändning av kod. Med andra ord är en funktion ett *underprogram* (eng. *subroutine*).

Det som i programmering kallas för parameter heter i matematiken *argument*. Den matematiska funktionen $y = f(x)$ har x som argument och y som funktionsvärde som i regel är tal. I programmering brukar man kalla x för parameter och y för returvärde som kan vara av vilka datatyper som helst. I grunden är det samma koncept, bara annorlunda beteckningar – en utvidgad synvinkel på samma sak.

Ur praktisk synpunkt kan en funktion i programmering jämföras med en (svart) låda i vilken man lägger in parametrar (indata) och får returvärdet (utdata):

En funktion kan ha inga, en eller flera parametrar. Den kan däremot ha endast 1 eller inget returvärde. En funktion kan inte ha flera returvärden. Men både parametrarna och returvärdet kan vara tal, tecken, strängar, sanningsvärden eller sammansatta datatyper (t.ex. en lista). Funktionen beabetar de ev. inkommande parametrarna enligt sin definition och returnerar ev. ett värde. Det finns även funktioner utan parameter och/eller utan returvärde.

Svart är lådan bara om vi inte vet på vilket sätt funktionen arbetar, om vi endast *använder* den för att lösa ett visst problem. Sådana funktioner har vi redan använt i våra program:

`input()`, `print()`, `int()`, `random()`, `range()`

De är svarta lådor för oss därför att de är förprogrammerade i Pythons programbibliotek. Vi vet bara vad de gör, men inte *hur* de gör, vilket dock räcker för att använda dem. Det typiska kännetecknet för funktioner är parenteserna `()` som står efter – eller bättre sagt i – namnet. De tillhör nämligen namnet, även när parentesen är tom.

Men nu vill vi inte bara använda svarta lådor utan skriva egna funktioner för att kunna använda dem som separata moduler i andra, mer avancerade program. Vi börjar med att skriva om programmet `MyRandom` (sid 67) till en funktion.

`MyRandom` som funktion

```
1   # RandFkt.py
2   # Definierar en funktion som tar in två parametrar a och b
3   # och returnerar ett slumptal i intervallet [a, b]
4   # Separat modul som kan anropas från olika program
5
6   import random
7
8   def myRand(a, b) :      # Funktionens huvud med namn
9       if a < b :          # och a, b som formella parametrar
10          return a + int(random.random() * (b-a+1))
11      else :
12          return b + int(random.random() * (a-b+1))
```

De viktigaste delarna i koden ovan är:

```
def myRand(a, b) :
    ...
    return ...
```

`def` inleder i Python definitionen till en funktion. Sedan följer funktionsnamnet `myRand()` samt parentesen som kallas för *parameterlistan*. Den innehåller `a` och `b` som heter *formella parametrar*. Hela första raden är *funktionshuvudet* följt av kolon. Sedan kommer funktionens *kropp* som skrivs indragen mot huvudet. Indragningen är en del av koden vars uppgift är att gruppera funktionens satser under huvudet till ett block.

`return`-satsen returnerar uttryckets värde som står efter `return`. `return`-uttryck-

ets värde, kort *returvärdet*, överförs till funktionsnamnet **myRand()**. På så sätt exporteras detta värde ur funktionen när denna anropas. Dessutom avslutar **return**-satsen funktionen: Eventuell kod som står efter **return** kommer inte att utföras.

Medan returvärdet är funktionens output (utdata) är parametrarna funktionens input (indata). Resten av koden i funktionen **myRand()**, dvs hela kroppen, består av en enda `if-else`-sats som vi tagit över från programmet **MyRandom** (sid 67), inkl. de generella formler som vi utvecklade där för beräkning av slumptal i ett intervall. Bara att vi slipper här listan **slumptal**, vilket beror på att vi endast behöver *ett* slumptal. Den nya koden har skrivits i den separata filen **RandFkt.py**.

Generellt kan strukturen av en funktionsdefinition anges så här där **fpar** står för formell parameter:

```
def funktionsnamn (fpar1, fpar2, ...) :
    sats(er)
    return uttryck
```

Om man nu skulle exekvera koden på förra sidan skulle ingenting hända, därför att allt som står där, endast är en funktions *definition*. För att aktivera koden måste funktionen *anropas*. Det görs i programmet nedan som skrivits i filen **RandTest.py**, medan funktionsdefinitionen lagras i filen **RandFkt.py**. Båda filer måste ligga i samma mapp och utgör *ett* program vars delar finns i två separata filer – ett resultat av modularisering:

```
1   # RandTest.py
2   # Importerar filen RandFkt.py, i samma mapp som denna fil
3   # Anropar funktionen myRand() definierad i RandFkt.py som
4   # i sin tur slumpar ett heltal i önskat intervall [A, B]
5
6   import RandFkt          # Importerar modulen RandFkt
7
8   print('\n\t I vilket intervall vill du ha slumptalet?')
9   A = int(input(
10          '\n\t Ange intervallets början (pos. heltal):  '))
11  B = int(input(
12          '\n\t Ange intervallets slut   (pos. heltal):  '))
13
14  print('\n\t Var så god!\tEtt slumptal mellan', A, 'och',
15                        B, ': ', RandFkt.myRand(A, B), '\n')
16                              # Anrop av funktionen myRand()
```

För att detta program ska hitta koden till funktionen **myRand()** måste vi först im-

portera modulen `RandFkt` i vilken funktionen är definierad. Detta sker i rad `6`. Sedan kan vi anropa funktionen, vilket görs i rad `15` med `RandFkt.myRand(A, B)`. Observera att anropet måste ange den fullständiga "sökvägen" till funktionen `myRand()` dvs inkludera den importerade modulen `RandFkt` – den logiska motsvarigheten till den fysiska filen `RandFkt.py` där funktionen är definierad.

Formella och aktuella parametrar

Man kan undra varför vi i prorammet `RandTest` definierat variablerna `A` och `B` med stora bokstäver och som en logisk följd även i rad `15` anropat funktionen `myRand()` med dem:

```
RandFkt.myRand(A, B)
```

Medan i funktionen `myRand()`:s *definition* (sid 72) parametrarna är betecknade med små bokstäver `a` och `b`:

```
def myRand(a, b) :
```

Vi vet också att Python är case sensitive, dvs `a` och `A` är olika variabler.

Anledningen till att vi gör så är att visa att vi har att göra med två olika situationer som ska behandlas i koden på två olika sätt:

`a` och `b` är funktionen `myRand()`:s *formella* parametrar.

`A` och `B` är funktionen `myRand()`:s *aktuella* parametrar.

De formella parmetrarna `a` och `b` används i funktionens definition och får där – till skillnad från "vanlig" kod – inga värden. De blir inte tilldelade där.

Ta som exempel experimentet på sid 38: Vi skrev bara `a` i Pythons Interactive mode och fick felmeddelandet: `name 'a' is not defined`. Python tolkade `a` som en variabel, och eftersom den inte tilldelats något värde blev det fel.

I funktionen `myRand()`:s definition har `a` och `b` inte heller tilldelats några värden (sid 72). Ändå kan vi exekvera koden utan felmeddelande. Det beror på att koden i en funktionsdefinition inte är en instruktion att utföra något utan endast en *föreskrift* om vad som *skulle* hända om funktionen *skulle* anropas. Det är kod i väntan på att den blir anropad (sid 70). De formella parmetrarna får sina värden först vid anropet. Så fungerar mekanismen hos funktioner.

De aktuella parmetrarna **A** och **B** däremot används i funktionens *anrop* och måste vara tilldelade innan anropet. Man skulle kunna kalla dem för *anropsparametrar*. De tilldelas värden i programmet **RandTest** på sid 73 (rad **9-12**). Värdena som de får där överförs sedan vid funktionsanropet till de formella parametrarna (rad **15**).

Här följer ett körresultat av programmet **RandTest**:

```
I vilket intervall vill du ha slumptalet?

Ange intervallets början (pos. heltal):  50

Ange intervallets slut   (pos. heltal):  250

Var så god!    Ett slumptal mellan 50 och 250 :  112
```

Övningar

1701

Funktionen $y = f(x) = x^2$ kan i Python definieras så här:

```
def f(x) :

    return x**2
```

a) Inkludera funktionen i ett program som anropar den för att skriva ut följande värdetabell för alla heltal x i intervallet [-5, 5]:

x	y
-5	25
-4	16
-3	9
-2	4
-1	1
0	0
1	1
2	4
3	9
4	16
5	25

Markera i ditt program definitionen och anropen med kommentar.

b) Utöka värdetabellen till x-intervallet [-10, 10].

1702

Definiera i Python funktionen:

$$y = f(x) = x^3$$

a) Inkludera funktionen i ett program som anropar den för att skriva ut en värdetabell för alla heltal x i intervallet [-5, 5] på samma sätt som i övn 1701 a).

b) Skriv värdena från a) i listor: Skapa en lista för alla x och en för alla y. Lägg in i dem respektive värdena och skriv ut listorna. Markera med kommentar funktionens definition och anrop.

1.7 Funktioner i programmering

1703

Skriv de fyra räknesätten inkl. heltalsdivision och modulo som funktioner i Python. Läs in två heltal. Anropa funktionerna och skriv ut resultaten så att du får följande utskrift när du läser in 5 och 3:

```
5  +  3  ger  8
5  -  3  ger  2
5  *  3  ger  15
5  /  3  ger  1.6666666666666667
5  // 3  ger  1
5  %  3  ger  2
```

Ditt program ska bli en modularisering av övn 1305:s lösning.

1704

Modularisera lösningen till övn 1411 genom att definiera beräkningen av `totalsek` som en funktion. Välj olika variabler för de formella och de aktuella parametrarna. Anropa funktionen. Testa ditt program t.ex. för 7 timmar, 58 minuter och 34 sekunder. Får du samma resultat som i övn 1411?

1705

Modularisera programmet **GissaTal_3** (sid 58) genom att generera slumptalet **secret** (rad 8) med funktionen **myRand()** (sid 72). Bibehåll **myRand()** i en separat fil och importera den tillhörande modulen i ditt program.

1.8 Kryptering

För omväxlingens skull visas här först körresultatet av krypteringsprogrammet **EncryptText** (sid 78) som vi kommer att utveckla och gå igenom i detalj sedan:

```
         Originaltext:

Denna text krypteras av ett pythonprogram. Samma program dekrypterar
texten. För att testa krypteringen skrivs ut originaltexten, den kryp-
terade och den återställa texten på skärmen. Den slumpade krypterings-
nyckeln visas.

         Krypterad text:

†§°°£b¶§º¶b´»²¶§´£µb£¸b§¶¶b²»¶ª±°²´±©´£¯pb•£¯¯£b²´±©
´£¯b¦§´»²¶§´£´b¶§º¶§°pb□κ´b£¶¶b¶§µ¶£b´»²¶§´«°©§°bµ-
´«¸µb·¶b±´«©«°£®¶§º¶§°nb¦§°b´»²¶§´£¦§b±¥ªb¦§°bħ¶§´µ¶Ħ®®£b¶§º¶§°b²ħbµ-
Ħ´¯§°pb□§°bµ®·¯²£¦§b´»²¶§´«°©µ°»¥§®°b¸«µ£µp

         Återställd text:

Denna text krypteras av ett pythonprogram. Samma program dekrypterar
texten. För att testa krypteringen skrivs ut originaltexten, den kryp-
terade och den återställa texten på skärmen. Den slumpade krypterings-
nyckeln visas.

Krypteringsnyckeln:      66
```

Det här är bara ett av flera möjliga körresultat man kan få när man kör programmet **EncryptText**, därför att krypteringsnyckeln är ett slumptal och därför olika vid varje köring. Just här vid den aktuella körningen är den 66.

Programmet **EncryptText** som genererar utskriften ovan är modulariserat och består av tre moduler som var och en lagras i en separat fil med ändelsen **.py**, alla liggande i samma mapp:

EncryptText	har	huvudprogrammet som anropar de två andra funktionerna
RandFkt		slumpfunktionen **myRand()**, sid 72
EncryptFkt		krypteringsfunktionen **encrypt()**, sid 79

I huvudprogrammet **EncryptText** importeras de två andra modulerna med sina resp. modulnamn strax i början (nästa sida, rad **7-8**). Så vi har deras koder med i resten av programmet. Denna består i huvudsak av variabeldefinitioner, **print()**-satser och anrop av de två funktionerna **encrypt()** och **myRand()**. Båda

dessa är definierade i resp. modul.

Programmet slumpar fram i rad **15** ett heltal i intervallet [50, 250] som används som krypteringsnyckel – vi kallar det i fortsättningen kort *slumpnyckel*. Sedan används den negativa slumpnyckeln för att återställa texten.

```
1     # EncryptText.py
2     # Skriver ut en text, krypterar den med en slumpnyckel
3     # Återställer sedan texten och skriver ut den både
4     # den krypterade och återställda texten
5     # Slumpnyckeln ger vid varje körning en annan kryptering
6
7     import RandFkt                          # Sid 72
8     import EncryptFkt                       # Sid 80
9
10    print('\n\t Originaltext:\n')
11    text = 'Denna text krypteras av ett pythonprogram. Samma
program dekrypterar texten. För att testa krypteringen skrivs
ut originaltexten, den krypterade och den återställa texten på
skärmen. Den slumpade krypteringsnyckeln visas.'
12
13    print(text)                             # Originaltext
14
15    key = RandFkt.myRand(50, 250)           # Slumpnyckeln
16
17    print('\n\t Krypterad text:\n')
18    text = EncryptFkt.encrypt(text, key)    # Krypterar text
19    print(text)
20
21    print('\n\t Återställd text:\n')
22    text = EncryptFkt.encrypt(text, -key)   # Återställer text
23    print(text)
24                                            # Visar slumpnyc-
25    print('\nKrypteringsnyckeln:\t', key, '\n')       # keln
```

I rad **11** skapas **string**-variabeln **text** för att lagra texten. Efter att ha skrivit ut den (rad **13**) anropas funktionen **myRand()** – från modulen **RandFkt** – som genererar ett slumptal mellan 50 och 250. Variabeln **key** tilldelas denna slumpnyckel i samma sats som funktionen **myRand()** anropas (rad **15**). Sättet att anropa en egendefinierad funktion från en externlagrad modul hade vi redan sett i programmet **RandTest** (sid 73, rad **15**). Här blir det så här:

```
15    key = RandFkt.myRand(50, 250)                # Slumpnyckeln
```

Dvs det räcker inte med att importera modulen **RandFkt**. Vi måste även i anropet ange den importerade modulens namn **RandFkt** med s.k. *punktnotation*. Så kallas i programmering skrivsättet i anropet ovan, nämligen att först skriva modulens namn, följt av en punkt, och sedan funktionens namn med de önskade aktuella parametrarna. Vi önskar att få ett slumptal mellan 50 och 250. Därför skickar vi i anropet de aktuella parametrarna 50 och 250 till funktionen **myRand()** där de tas emot av de formella parametrarna a och b. I själva verket överförs värdena 50 till a och 250 till b. Det är platsen i parameterlistan som avgör vilken aktuell parameter som överförs till vilken formell parameter. Funktionen **myRand()** som vi definierat tidigare returnerar det önskade slumptalet till variabeln **key** som vi i fortsättningen använder som slumpnyckel. Se koden till **myRand()** på sid 72.

I rad **18** av programmet **EncryptText** ingår sedan slumpnyckeln **key** som vi fick i rad **15** som parameter i anropet av krypteringsfunktionen **encrypt()**:

```
18    text = EncryptFkt.encrypt(text, key)     # Krypterar text
```

Även **text** som innehåller originaltexten är en parameter i anropet av funktionen **encrypt()** ovan. Originaltexten skickas alltså till **encrypt()**, krypteras där, kommer tillbaka som krypterad text och tilldelas igen till **string**-variabeln **text**. Dvs variabeln **text** överskrivs med nytt innehåll som nu – efter rad **18** – är den krypterade texten som sedan skrivs ut i rad **19**.

Hur krypteringen går till avslöjar funktionen **encrypt()**:

```
1     # EncryptFkt.py
2     # Definierar funktionen encrypt() med param. oldText och k
3     # Krypterar oldText, förskjuter alla tecken med k steg
4     # Krypterad text skrivs teckenvis till variabeln newText
5     # Sist returneras den krypterade texten
6
7     def encrypt(oldText, k) :
8         newText = ''                 # Tom sträng
9         for n in range(0, len(oldText)) :
10            ch = oldText[n]          # Tar tecknen från oldText
11            ch = chr(ord(ch) + k)    # Krypterar tecknen
12            newText = newText + ch # Lägger tecknen i newText
13        return newText               # Reurnerar krypterad text
```

Med den första parametern `oldText` får funktionen **`encrypt()`** tillgång till den `string`-variabeln **`text`**:s innehåll som skapades i programmet `EncryptText`, kort sagt till originaltexten. Detta pga att i anropet i rad `18` av programmet `EncryptText` **`text`** står på första parameterplats i *anropet* av funktionen **`encrypt()`**. Detta motsvaras av den första plats i parameterlistan av *definitionen* av funktionen **`encrypt()`** där vi hittar `oldText` (rad `7`). Det är alltså den aktuella parametern **`text`** som överförs till den formella parametern `oldText`. På så sätt hamnar originaltexten i funktionen och representeras där av `oldText`.

Den andra formella parametern i funktionen **`encrypt()`** är `k` som får sitt värde från den andra aktuella parametern **`key`** som vi hittar i anropet i rad `18` av programmet `EncryptText` på andra plats i parameterlistan. Värdet används i rad `11` av funktionen **`encrypt()`** för att förskjuta alla tecken i originaltexten med `k` steg i teckentabellen. I det inledningsvis visade körexemplet var detta värde 66.

Den tomma strängen

I kroppen av funktionen **`encrypt()`** definieras först en `string`-variabel **`newText`** för att ta hand om den krypterade texten. Den blir av datatypen `string` genom att den initieras till den tomma strängen (rad `8`). I Python får man en tom sträng med koden `''`. Observera att koden `''` är två apostrofer utan mellanslag. Skriver vi dem istället med mellanslag: `'␣'` blir det felmeddelande när vi kör programmet `EncryptText`. `'␣'` är inte en tom sträng utan en sträng av längden 1 bestående av tecknet mellanslag, vilket leder till invecklade följdfel i programkoden som vi inte tar upp här. Testa gärna själv! Men skillnaden mellan `''` och `'␣'` kan man lätt ta reda på i Pythons Interactive mode:

```
>>> ''
''
>>> len('')
0
>>> ' '
' '
>>> len(' ')
1
```

`len()` är en fördefinierad funktion i Python som tar in en sträng och returnerar dess längd (antal tecken). Experimentet ovan visar att längden av strängen `''` är `0` medan längden av strängen `'␣'` är `1`. Detta talar om att den tomma strängen är `''` och att `'␣'` är en sträng bestående av ett enda tecken, närmare bestämt mellanslaget.

Krypteringsalgoritmen

Själva krypteringen i funktionen **encrypt()** pågår i for-satsen (rad 9-12) med huvudet:

```
9         for n in range(0, len(oldText)) :
```

for-satsen går igenom alla tecken i originaltexten oldText genom att initiera räknaren **n** till 0 och avsluta loopen när räknaren har nått strängens sista tecken. Att man börjar med 0 beror på att Python numrerar strängens första tecken med 0, det andra med 1 osv. Därför har vi avslutningsvillkoret **n** < len(oldText), se sid 64 om for-satsen. För att få tag i originaltexten oldText:s enskilda tecken börjar for-satsens kropp med:

```
10            ch = oldText[n]          # Tar tecknen från oldText
```

För att förstå detta, speciellt betydelsen av **[n]** öppnar vi Pythons Interactive mode:

```
>>> oldText = 'abcdefgh'
>>> oldText[0]
'a'
>>> oldText[1]
'b'
>>> oldText[2]
'c'
>>> oldText[7]
'h'
>>> len(oldText)
8
```

Man ser att hakparentesen tar ut en strängs enskilda tecken: [0] ger strängens första, [1] dess andra tecken osv. Siffrorna inom hakparentes kallas för *index*. Index börjar alltid med 0. Index 2 ger strängens 3:e tecken, bokstaven c. Index 7 ger strängens 8:e dvs sista tecknet. len() bekräftar att strängen har 8 tecken.

I for-satsen går räknaren **n** igenom strängens första tecken (index 0) till strängens sista tecken. I varje varv tas ut ett tecken ur originaltexten oldText och tilldelas variabeln **ch** (rad 10). Sedan krypteras detta tecken genom att addera till dess s.k. ASCII-kod slumpnyckeln k och omvandla ASCII-koden tillbaka till motsvarande tecken (bokstav). Variabeln **ch** tilldelas då det nya tecknet som är förskjutet framåt med k steg i teckentabellen (se nedan om ASCII-koder):

```
11            ch = chr(ord(ch) + k)          # Krypterar tecknen
```

För att förstå detta måste vi öppna här en liten parentes om hur bokstäver kodas i datorn med heltalskoder och hur man med Pythons funktioner **`ord()`** samt **`chr()`** kan få tag i koderna och omvandla bokstav till kod och omvänt.

ASCII-koder

ASCII (uttalas ”aski”) står för *American Standard Code for Information Interchange* och är en standard för kodning av tecken skapad av det amerikanska standardiseringsorganet. ASCII är en universell standard och används i alla datorer över hela världen.

ASCII omfattar alla engelska bokstäver, siffrorna 0-9, de vanligaste specialtecknen och en del styr- och kontrolltecken. Men den har vissa begränsningar. T.ex. specialtecken i andra språk än engelskan saknas, bl.a. de svenska tecknen å, ä, ö, Å, Ä, Ö. För att få bukt med dessa begränsningar har en större teckenstandard skapats i vilken ASCII ingår som en delmängd och som heter *Unicode*. Alla programmeringsspråk inkl. Python använder Unicode. När vi i boken paratar om teckentabellen menar vi Unicode-tabellen.

Unicode är en teckenkodningsstandard som inkluderar och utvidgar ASCII-koderna. Den är identisk med ASCII i kodintervallet 0-127. Med Unicode kan man koda ett väsentligt större antal tecken, även sådana från andra språk som arabiska, japanska, kinesiska, hebreiska, kyrilliska osv. Eftersom de flesta tecken vi använder i programmering ligger i kodintervallet 0-127 där ASCII = Unicode, kommer vi i fortsättningen för enkelhetens skull att endast prata om ASCII-koder eller kort koder.

Funktionen **`ord()`**

Denna funktion är en av Pythons *built-in functions*. Vad den gör är att ge oss ett teckens ASCII-kod. Vi testar **`ord()`** i Interactive mode:

```
>>> ord('a')
97
>>> ord('z')
122
```

ASCII-koden till bokstaven a är 97 och z:s kod är 122. Funktionen **ord()** returnerar alltså koden till en bokstav. Den inversa (omvända) funktionen till **ord()** är:

Funktionen chr()

Även denna funktion är inbyggd i Python och ger oss tecknet tillhörande en ASCII-kod:

```
>>> chr(97)
'a'
>>> chr(122)
'z'
```

Bokstaven till ASCII-koden 97 är a och koden 122:s bokstav är z. Funktionen **chr()** returnerar alltså bokstav tillhörande koden.

Programmet **Char2int** nedan använder sig av funktionen **ord()** för att ge oss ASCII-koden till vilket tecken som helst vare sig bokstav, siffra eller specialtecken:

```
1    # Char2int.py
2    # Ger ASCII-koden till ett inmatat tecken
3
4    letter = input('\n\tMata in ett tecken:\t')
5
6    print('\n\tDet inmatade tecknet', letter,
7                  'har ASCII-koden', ord(letter), '.\n')
```

Följande körning visar ASCII-koden till bokstaven A:

```
        Mata in ett tecken:     A

        Det inmatade tecknet A har ASCII-koden 65 .
```

Det omvända problemet

Programmet **Char2int** gav oss ASCII-koden när vi matade in ett tecken. Programmet nedan löser det omvända problemet: Int2char använder sig av funktionen chr() för att skriva ut tecknet när vi matar in ett heltal som är tecknets ASCII-kod.

```
1    # Int2char.py
2    # Ger tecknet till ett inmatat heltal (ASCII-kod)
3
4    code = int(input('\n\tMata in ett positivt heltal:\t'))
5
6    print('\n\tTalet', code, 'är ASCII-koden till',
7                                        chr(code), '.\n')
```

Följande körning visar bokstaven tillhörande ASCII-koden 65:

```
Mata in ett positivt heltal:    65

Talet 65 är ASCII-koden till A .
```

En kombination av de två funktionerna **ord()** och **chr()** för ger oss möjligheten att kryptera t.ex. bokstaven a genom att addera till dess ASCII-kod slumpnyckeln 66 och på så sätt komma till tecknet £. Sedan kan vi få tillbaka (återställa) bokstaven a genom att subtrahera samma slumpnyckel från tecknet £:s kod:

I for-satsens sista sats fylls den tomma strängen **newText** med innehåll:

```
12     newText = newText + ch    # Lägger tecknen i newText
```

Detta görs genom att vi i varje varv av for-satsen konkatenerar **newText** med det enskilda tecknet **ch** som vi tagit från oldText och förändrat i rad 11. På så sätt fylls **newText** med de nya tecknen och bildar – beroende på **key**:s värde – den krypterade eller återställda texten som sedan returneras via funktionsnamnet **encrypt()** till programmet.

Tillbaka till programmet EncryptText (sid 78), där vi efter första anropet av funktionen **encrypt()** aropar den en andra gång för att dekryptera texten:

```
22   text = EncryptFkt.encrypt(text, -key)   # Återställer text
```

Här ska tecknet **–** framför **key** inte tolkas som bindestreck utan som minustecknet till variabeln **key**:s talvärde. Vi skickar alltså slumpnyckeln **key**:s negativa värde till samma krypteringsfunktion **encrypt()** för att sätta tillbaka alla tecken på sina ursprungliga platser i teckentabellen. Den aktuella parametern **key** överförs vid anrop till den formella parametern k. När k får ett positivt **key**-värde, ökas teckenens ASCII-kod med k steg. Ett negativt **key**-värde minskar ASCII-koderna med samma belopp. Därför kan vi använda samma funktion även för återställning av originaltexten.

Övningar

1801

Experimentera med programmet **Char-2int** (sid 83) för att ta reda på differensen mellan gemeners och versalers ASCII-koder. Skriv ett program som läser in en gemen och skriver ut dess versal och sedan läser in en versal och skriver ut dess gemen. Använd Pythons funktioner **ord()** och **chr()** (sid 82).

1802

Skriv ett program som läser in ett tecken och förskjuter det i teckentabellen med ett visst antal steg som en slags krypteringsnyckel. Skriv ut både det inlästa och det förskjutna tecknet på ett användarvänligt sätt.

1803

Skriv ett program som läser in fem tecken och skriver ut dem förskjutna med steget 1 i teckentabellen så att t.ex. inmatningen Kalle ger utskriften Lbmmf. Återställ sedan det krypterade ordet utgående från den krypterade versionen. Vidareutveckla programmet genom att öka steg och läsa in krypteringsnyckeln.

1804

Vidareutveckla lösningen till övn **1803** genom att utöka och läsa in antalet tecken. Mata in text av godtycklig längd, kryptera den och återställ sedan utgående från den krypterade texten.

1805

Modularisera lösningen till övn **1804** genom att skriva koden för både kryptering och återställning som en funktion kallad **myEncrypt()**. Anropa funktionen med den inlästa nyckeln för att kryptera texten och med den negativa nyckeln för att återställa den.

1.8 Kryptering

1806

Modifiera lösningen till övn **1805** genom att generera krypteringsnyckeln med slumptal. Använd funktionen **myRand()** (sid 72) för detta och välj själv intervallet för val av slumptal. Låt **myRand()** ligga i en separat fil och importera den som en modul i ditt program.

1807

Skriv ut med en loop en teckentabell i ASCII-intervallet [33, 256]. Utskriften ska visa varje tecken bredvid sin ASCII-kod i en tabell med 8 kolumner, dvs gör radbyte var 8:e utskrift. För att undvika radbyten med **print()**-satser i loopen gör så här: Skapa en tom sträng före loopen, lägg i den alla utskrifter i loopen och skriv ut den efter loopen med en enda **print()**-sats.

1808

Skriv ett program som skapar slumplösenord bestående av 8 tecken med följande policy: 3 små bokstäver, 2 siffror och 3 stora bokstäver. Experimentera med ASCII-tabellen från övn **1807** för att få reda på kodintervallerna till lösenordens olika delar. Programmet ska fråga efter önskat antal lösenord och skriva ut dem i två kolumner: I den första ska stå: user1, user2, ... I den andra ska stå ett slumpvis genererat lösenord till varje användare.

1.9 Primtal

Alla heltal är jämnt delbara med sig själv och med 1. De flesta av dem är dessutom jämnt delbara med andra tal. Men några kan inte delas jämnt med något annat tal än med sig själv och med 1. De kallas för *primtal*. Exempel:

Primtal: 2, 3, 5, 7, 11, 13, 17, 19, 23, ...

Inga primtal: 4, 6, 8, 9, 10, 12, 14, ... *Sammansatta tal* – sammansatta av primtal.

Primtal är alltså positiva heltal > 1 som är jämnt delbara *endast* med sig själv och med 1. I denna bemärkelse är primtal odelbara – talsystemets atomer så att säga.

Aritmetikens fundamentalsats

Varje positivt heltal kan på ett entydigt sätt delas upp i en produkt av primtal, bortsett från faktorernas inbördes ordning.

Exempel:

$$12 = 2 \cdot 2 \cdot 3$$

$$98 = 2 \cdot 7 \cdot 7$$

$$11111111 = 11 \cdot 73 \cdot 101 \cdot 137$$

Primfaktorernas inbördes ordning är utan betydelse därför att multiplikation är kommutativ: $a \cdot b = b \cdot a$. Medan de första två exemplen är någorlunda begripliga är det svårt att verifiera det tredje exemplet utan miniräknare. Och även då kan det endast verifieras i en riktning: från höger till vänster. Omvänt, från vänster till höger, dvs hur själva faktoriseringen av 11111111 går till är en jobbig procedur, även om man kan det.

Det vore kul att låta ett program göra jobbet åt oss. Man skulle kunna mata in vilket tal som helst och få dess uppdelning i primfaktorer – en slags digital realisering av aritmetikens fundamentalsats. I detta avsnitt ska vi utveckla ett sådant program. Vi gör det i tre steg:

a) Primtalstest (här) med programmen: `PrimtalsTest, PrimFkt, PrimTest`

b) Alla primtal i ett intervall (sid 91) `AllaPrimtal`

c) Primtalsfaktorisering (sid 92) `PrimFaktorer`

a) Primtalstest

Hur kan vi avgöra om ett givet tal är primtal? Viktig fråga, t.ex. när vi vill förkorta bråk, hitta alla primtal i ett intervall eller dela upp ett tal i primfaktorer. Vi behöver bara testa om det givna talet är jämnt delbart med alla mindre positiva heltal. Delbarhet med 2 har vi lärt oss koda på sid 46. Det avgörande verktyget var modulooperatorn (sid 48) som vi även kommer att använda nu. Nedan följer programmet som implementerar denna enkla algoritm:

1. Ange ett tal. Testa om det är jämnt delbart med *alla* mindre positiva heltal.
2. Om ja, är det ett primtal, annars är det sammansatt.

```
1   # PrimtalsTest.py
2   # Läser in ett tal och avgör om det är ett primtal eller ej
3   # genom att testa jämn delbarhet med alla mindre tal
4
5   tal = int(input('\n\tMata in ett positivt heltal:\t'))
6   prim = True                    # tal antas vara primtal
7   k = 2
8
9   while k <= tal-1 and prim :    # För alla mindre tal loopa
10     if tal % k != 0 :           # tal inte delbart med k:
11        prim = True              #                 primtal
12     else:
13        prim = False             # Annars sammansatt
14     k = k + 1
15
16  if prim :
17     print('\n\tDet inmatade talet', tal,'är ett primtal.\n')
18  else :
19    print('\n\tDet inmatade talet', tal,'är sammansatt.\n')
```

Här följer några körresultat:

```
        Mata in ett positivt heltal:    13

        Det inmatade talet 13 är ett primtal.
```

```
Mata in ett positivt heltal:    12
Det inmatade talet 12 är sammansatt.
```

```
Mata in ett positivt heltal:    997
Det inmatade talet 997 är ett primtal.
```

```
Mata in ett positivt heltal:    1234567890123456789
Det inmatade talet 1234567890123456789 är sammansatt.
```

Observera att vi i det sista körexemplet har testat om ett **19**-siffrigt heltal är primt eller ej. Det visade sig att det var sammansatt. Vi kommer med programmet **PrimFaktorer** dela upp detta stora heltal i sina primfaktorer (sid 96, sista körningen).

Algoritmen i programmet **PrimtalsTest** testar om det inmatade talet är jämnt delbart med *alla* tal mindre än det inmatade. Detta innebär en del upprepade beräkningar av samma typ som kodas i en `while`-loop (rad **9-14**) som introducerades på sid 55. För att förstå `while`-satsen i **PrimtalsTest** ska vi undersöka dess villkor som är nyckeln till algoritmens funktionalitet.

Den logiska operatorn **and**

`while`-satsens villkor är ett *sammansatt villkor* som bildas med **and**:

```
9     while k <= tal-1 and prim :  # För alla mindre tal loopa
```

Villkoret består av två delar: **k <= tal-1** och **prim** där den första är en jämförelse och den andra en logisk variabel (sid 57). De är sammansatta med det reserverade ordet **and** som är en *logisk operator* och står för det logiska OCH. Villkoret säger: Så länge **k** är mindre eller lika med **tal-1** OCH **prim** är sant, loopa. Med **and** bildas alltså ett nytt, sammansatt villkor. Den logiska innebörden av **and** är: För att det sammansatta villkoret ska vara sant måste villkorets båda delar vara sanna. Om en del är falsk blir hela villkoret falskt.

Observera att **prim** är en logisk variabel. Detta betyder att **prim**:s sanningsvärde

testas när den förekommer i ett villkor. Samma gäller när **prim** kombineras med **and** i rad 9 ovan: **and prim** betyder samma som **and prim == True**.

Anledningen till denna formulering av while-villkoret är att algoritmen kräver att det inmatade talet ska testas på delbarhet med tal som är *mindre* än det inmatade (del 1), *OCH* att detta ska ske för *alla* sådana tal (del 2). Dvs ska det inmatade talet vara primtal, får det inte finnas *någon* delare bland alla tal som är mindre. Därför initierar vi variabeln **prim** till **True** i början utanför loopen. Vi antar alltså först att det inmatade talet är primtal (optimistiskt!) och låter det komma in i loopen. Om if-else-satsen (rad **10-13**) hittar en delare till talet i ett av loopens varv, dvs om **tal % k == 0**, åker talet ut och kommer aldrig in i loopen igen. Programmet skriver ut talet som sammansatt. Detta på grund av **and prim** i villkoret. **prim** förblir däremot **True** om talet klarar av loopens alla varv, dvs om if-else-satsen inte hittar någon delare till talet i något av loopens varv. Det ordnas av if-else-satsens villkor: **tal % k != 0** (rad **10**). Man ser hur while-satsens villkor är avgörande för hela algoritmens funktionalitet.

PrimtalsTest som funktion

Man vill inte bara veta om ett givet tal är primtal eller ej utan t.ex. låta datorn hitta alla primtal i ett heltalsintervall. Eller skriva ett program där vi matar in vilket heltal som helst och får alla dess primfaktorer, dvs programmera aritmetikens fundamentalsats om entydig uppdelning av alla heltal i primfaktorer. Säkert kommer vi i de nya programmen behöva testa om ett givet tal är primtal eller ej. Men vi vill i dessa program helst inte göra om allt programmeringsarbete vi lagt på **PrimtalsTest**. Vi vill använda den kod vi redan har. Det kan vi bara göra om vi skriver **PrimtalsTest** som en funktion, vilket vi gör på nästa sida. Vi döper funktionen till **primtest()**. För terminologin om funktioner hänvisas till avsnitt 1.7 *Funktioner i programmering* (sid 70-75).

I fortsättningen refererar vi till modulen **PrimFkt** på nästa sida där funktionen **primtest()** är definierad. Den viktigaste delen av koden där är:

```
def primtest(n) :
    ...
    return prim
```

Funktionsnamnet är **primtest()**. Parameterlistan innehåller **n** som formell parameter. **return**-satsen returnerar den logiska variabeln **prim**, dvs ett returvärde av typ boolean. Funktionen tar in ett heltal **n** och returnerar **True** om **n** är primtal eller **False** om **n** är sammansatt. Detta returvärde överförs till funktionsnamnet **primtest()** och tas emot om av det program som anropar funktionen.

```
# PrimFkt.py
# Definierar en funkion som tar in ett tal via parametern n
# Returnerar True om n är primtal, False om n är sammansatt
# Separat modul som kan anropas från olika program

def primtest(n) :              # Funktionens huvud med namn
    prim = True                # och n som formell parameter
    k = 2
    while k <= n-1 and prim :
       if n % k != 0 :
          prim = True
       else:
          prim = False
       k = k + 1
    return prim
```

Resten av koden i funktionen ovan, dvs hela kroppen, är direkt kopierad från programmet `PrimtalsTest` (sid 87). Den nya koden har skrivits i filen `PrimFkt.py`.

Om man nu skulle exekvera koden ovan skulle ingenting hända, därför att allt som står där, endast är en funktions *definition*. För att aktivera koden måste funktionen *anropas*. Det görs i programmet nedan som skrivits i filen `PrimTest.py` medan funktionsdefinitionen lagras i filen `PrimFkt.py`. Båda filer måste ligga i samma mapp. De utgör *ett* program vars delar finns i två separata filer – ett resultat av modularisering.

I programmet `PrimTest` nedan kan vi nu anropa funktionen **`primtest()`**, vilket ger exakt samma körresultat som visades på sid 87-88. Vi återger dem inte en gång till här, utan: testa gärna själv!

```
# PrimTest.py
# Importerar filen PrimFkt.py från samma mapp som denna fil
# Anropar funktionen primtest() definierad i PrimFkt.py

import PrimFkt              # Importerar modulen PrimFkt

tal = int(input('\n\tMata in ett positivt heltal:\t'))

if PrimFkt.primtest(tal) : # Anrop av funktionen primtest()
   print('\n\tDet inmatade talet', tal,'är ett primtal.\n')
else :
   print('\n\tDet inmatade talet', tal,'är sammansatt.\n')
```

För att programmet **PrimTest** ska hitta koden till funktionen **primtest()** måste vi först importera dess fil i början (rad 5) och sedan anropa den med **PrimFkt.primtest(tal)** i rad 9. I anropet måste anges att funktionens definition ligger i den importerade modulen **PrimFkt** – den logiska motsvarigheten till den fysiska filen **PrimFkt.py**.

Att anropet kan skrivas i `if`-satsens villkor beror på att funktionen **primtest()** returnerar **True** eller **False** och därför kan användas som ett villkor.

b) Alla primtal i ett intervall

Nu när vi skrivit programmet **PrimtalsTest** som en funktion (sid 90) kan vi använda den för att i princip hitta alla primtal med hjälp av datorn. Men våra resurser (tid och rum) är begränsade. Så vi får nöja oss med att göra det i ett begränsat intervall. Det som behövs är en loop som skannar igenom intervallet och med hjälp av funktionen **primtest()** avgör om det aktuella talet är primtal eller ej. Om det är fallet läggs det i en primtalslista som skrivs ut när loopen är klar.

Exakt detta gör följande program som på köpet även ger oss *antalet* primtal i intervallet:

```
1   # AllaPrimtal.py
2   # Hittar och skriver ut alla primtal i ett givet intervall
3   # Anropar primtest() från filen PrimFkt.py (sid 74)
4
5   import PrimFkt                          # Koden av primtest()
6
7   print('\n I vilket intervall vill du ha primtalen?')
8   a=int(input('\n Ange intervallets början (pos. heltal):  '))
9   b=int(input('\n Ange intervallets slut   (pos. heltal):  '))
10
11  primtal = [ ]
12
13  for tal in range(a, b) :
14      if PrimFkt.primtest(tal) and tal!=1 : # Anrop av fkt.
15         primtal = primtal + [tal]
16
17  print('\n ', primtal, '\n')
18  print(' Det finns', len(primtal), 'primtal i intervallet.\n')
```

Så här får vi t.ex. alla primtal mellan 1 och 60 samt deras antal när vi exekverar:

```
I vilket intervall vill du ha primtalen?

Ange intervallets början (pos. heltal):  1

Ange intervallets slut   (pos. heltal):  60

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59]

Det finns 17 primtal i intervallet.
```

I programmet **AllaPrimtal** (förra sidan, rad **11**) skapas en tom lista med koden **[ ]** som vi döper till **primlista**. Primtalen 2-59 i utskriften ovan visas i denna lista. Om den sammansatta datatypen lista läs på sid 65.

len() är en fördefinierad funktion i Python som returnerar antalet element i listan. Den anropas i rad **18** och ger svaret att det finns 17 primtal i intervallet (1, 60).

for-satsen (rad **13-15**) går igenom det givna intervallet med hjälp av räknaren **tal**. När vi som i det här fallet känner till intervallets start- och slutvärde känns det naturligt att använda en for-sats (sid 64).

Nu ska vi ägna oss åt lösningen av problemet att hitta alla primfaktorer av ett givet heltal. Den teoretiska garantin för att en sådan faktorisering alltid *existerar* är aritmetikens fundamentalsats om unik uppdelning av positiva heltal i primfaktorer.

c) Primtalsfaktorisering

Med division menas i hela detta avsnitt *heltalsdivision*. Med delbart menas alltid jämnt delbart. Ett sätt att hitta ett tals primfaktorer är att dividera det med 2, 3, 4, ... tills resten blir 0. Det första tal man delar med som ger resten 0 är en primfaktor.

Exempel utan Python:

Om vi delar 65 med 2, 3 och 4 blir resten inte 0, dvs 65 är inte delbart 2, 3 och 4. Därför kan dessa tal inte vara primfaktorer av 65. Om vi däremot fortsätter och delar 65 med 5 blir resten 0, dvs 65 är delbart 5. Därför är 5 en primfaktor av 65.

Om vi sedan delar 65 med primfaktorn 5 dvs "tar bort" 5 från 65 blir 13 kvar som är ett primtal. Vi har fått fram 65:s primtalsfaktorisering: $65 = 5 \cdot 13$. Om 65 delat med 13 inte hade gett ett primtal, så hade vi fått produkten av de resterande primfaktorerna. Då hade vi kunnat upprepa förfarandet med denna produkt.

Samma exempel i Pythons Interactive mode:

```
>>>
>>> 65 % 2                      # Faktorisera 65
1
>>> 65 % 3
2
>>> 65 % 4
1
>>> 65 % 5
0
>>> faktorer = [5]              # 5 Primfaktor -> Lista faktorer
>>> 65 // 5                     # Tar bort faktorn 5 från 65
13
>>> faktorer = faktorer + [13] # 13 är ett primtal -> Lista
>>> faktorer                    # 65:s primtalsfaktorisering:
 [5, 13]
>>> 5 * 13                      # Kontroll
65
>>>
```

Vi dividerar `65` med `2`, `3`, `4`, ... tills resten blir `0`. Det första tal som ger resten `0` är `5`. Alltså är det `65`:s första primfaktor. Vi definierar en lista som vi kallar för **`faktorer`** och lägger `5` i den: **`faktorer = [5]`**. Dvs vi initierar den till att innehålla endast elementet `5`. Sedan ”förkortar” vi `65` genom att dela det med `5`. Egentligen borde vi upprepa nu samma förfarande med resultatet av `65/5`. Men vi konstaterar att resultatet `13` är ett primtal. Alltså lägger vi det till listan: **`faktorer = faktorer + [13]`**. Dvs vi konkatenerar den gamla listan med elementet `13`. Sist skriver vi ut listan **`[5, 13]`** och gör kontroll. Talet `65`:s primtalsfaktorisering är $65 = 5 \cdot 13$.

Vad vi kallade för förfarandets upprepning behövde inte genomföras eftersom resultatet blev primtalet `13`. Om det hade varit sammansatt hade vi kunnat utnyttja det för att förkorta och förenkla algoritmen.

Följande exempel med ett större tal skulle kunna ge oss en förståelse för upprepningsmomentet och bidra till algoritmens allmänna formulering:

```
>>>
>>> 132 % 2                    # Faktorisera 132
0
>>> faktorer = [2]             # 2 primfaktor -> Lista faktorer
>>> 132 // 2                   # Tar bort faktorn 2 från 132
66
>>> 66 % 2                     # Faktorisera 66: Upprepa med 66
0
>>> faktorer = faktorer + [2]  # En till 2:a som primfaktor ->
Lista
>>> 66 // 2                    # Tar bort faktorn 2 från 66
33
>>> 33 % 2                     # Faktorisera 33: Upprepa med 33
1
>>> 33 % 3
0
>>> faktorer = faktorer + [3]  # 3 är en primfaktor -> Lista
>>> 33 // 3 # Tar bort faktorn 3 från 33
11
>>> faktorer = faktorer + [11] # 11 är ett primtal -> Lista
>>> faktorer                   # 132:s primtalsfaktorisering
[2, 2, 3, 11]
>>> 2 * 2 * 3 * 11             # Kontroll
132
>>>
```

Skillnaden med det första exemplet är att vi nu efter den första divisionen av `132` med `2` som ger resten `0`, inte får ett primtal utan `66`. Vi tar in `2` i listan över primfaktorer och upprepar förfarandet med `66`: Division av `66` med `2` med resten `0` ger igen ett sammansatt tal: `33`. Vi tar in denna andra `2`:a i primfaktorlistan och upprepar förfarandet med `33`: Efter två försök, division med `2` och `3`, ger det sista resten `0`. Därför tas `3` in i listan. Resultatet `11` är ett primtal och hamnar i listan: **`[2, 2, 3, 11]`**. Så blir talet `132`:s primtalsfaktorisering: $132 = 2 \cdot 2 \cdot 3 \cdot 11$.

Om vi hade en algoritm som dividerade ett tal med `2`, `3`, `4`, ... tills resten blir `0`, hade vi kunnat skicka det sammansatta talet som blir resultatet av den sista divisionen (med rest `0`) tillbaka till denna algoritm för att *upprepa förfarandet*. Detta tillbakaskickande för att upprepa samma sak ska vi nu utnyttja i algoritmens allmänna formulering.

När vi sammanfattar våra lärdomar från de två experimenten ovan kan vi formulera följande allmän algoritm för uppdelning av positiva heltal i primfaktorer.

Algoritmen Primtalsfaktorisering

Låt **n** vara det tal som ska delas upp i primfaktorer. Skapa en tom lista.

1. Dividera talet **n** med 2, 3, 4, ... tills resten blir 0.
2. Låt **k** vara det första tal du dividerat med som gav resten 0. **k** är primfaktor.
3. OM **n** är primtal lägg **n** i listan ovan och avsluta algoritmen,
   ANNARS lägg primfaktorn **k** i listan ovan,
   gå tillbaka till punkt 1 med: **n** = **n** / **k** (*Rekursion*!)

Algoritmen är rekursiv pga tillbakagången i sista raden. Om rekursion läs i nästa avsnitt på sid 98 (och jfr. med sid 59). Nedan följer en rekursiv pythonfunktion som implementerar den.

Programmet Primtalsfaktorisering

```
1  # PrimFaktorer.py
2  # Definierar rekursiv funktion faktorisera() och anropar den
3
4  def faktorisera(n) :                # Definierar rek. funktion
5      faktorer = []                   # Lista över primfaktorer
6      k = 1
7      rest = 1
8      while k <= n-2 and rest != 0 : # Letar efter primfaktor k
9         k = k + 1
10        rest = n % k
11
12     if rest != 0 :
13        faktorer = faktorer + [n]    # n primtal -> Lista
14     else :
15        faktorer = faktorer + [k]    # k delar n -> Lista
16        faktorer = faktorer + faktorisera(n//k) # Rekursion:
17                              # Anropar sig själv i definitionen
18     return faktorer
19                                     # Programmet:
20  tal = int(input('\n\t Mata in ett positivt heltal:\t'))
21  print('\n\t Talets primfaktorer:\t', faktorisera(tal), '\n')
22                                     # Anropar rekursiv funktion
```

Koden ovan består av två delar: I raderna **4-18** definieras den rekursiva funktionen **faktorisera()**. I raderna **20-21** står ett program som först läser in ett tal och

sedan anropar funktionen **faktorisera()**. Vid anropet skickas det inlästa talet till funktionen som är en implementering av algoritmen ovan i pythonkod.

I funktionen **faktorisera()** dividerar `while`-satsen talet **n** med 2, 3, 4, ... tills resten blir 0. Då har den hittat den faktor **k** som delar **n** och därför är en primfaktor. I `else`-delen av `if`-satsen läggs denna faktor i listan **faktorer**. Sedan bryts **k** loss från **n** genom heltalsdivisionen **n // k**. Sedan startas hela algoritmen om, nu med **n // k** som ska faktoriseras. Dvs funktionen **faktorisera()** anropar sig själv i sin egen definition genom anopet **faktorisera(n // k)** i rad **16**. Detta kallas för rekursion som vi kommer att titta närnare på med hjälp av ett annat exempel i nästa avsnitt. Här följer några körresultat:

```
Mata in ett positivt heltal:   12
Talets primfaktorer:    [2, 2, 3]
```

```
Mata in ett positivt heltal:   98
Talets primfaktorer:    [2, 7, 7]
```

```
Mata in ett positivt heltal:   997
Talets primfaktorer:    [997]
```

```
Mata in ett positivt heltal:   11111111
Talets primfaktorer:    [11, 73, 101, 137]
```

```
Mata in ett positivt heltal:   1234567890123456789
Talets primfaktorer:    [3, 3, 101, 3541, 3607, 3803, 27961]
```

Den sista körningen ger upphov till funderingar om hur stora heltal man kan jobba med i Python utan att råka ut för problem. Just detta exempel med **19** heltalssiffror gick ju bra. Men det finns som i alla system – så även i Python – en övre gräns för hur långt man kan gå. Python räknar med dubbelprecision. Vi går inte in på detaljer eftersom denna gräns beror på många faktorer som är oförutsägbara: Pythons version, miljöns inställningar, datorns prestation osv. Läs, om du vill, i manualer och tutorials på nätet. Det säkraste är alltid att själv bygga in någon kontroll i sitt program. Här skulle det räcka att kontrollera om produkten av primfaktorerna ger oss tillbaka det ursprungliga talet.

Övningar

1901

Matematikern *Fermat* trodde på 1600-talet att talet $2^{32} + 1 = 4\ 294\ 967\ 297$ var ett primtal. På 1700-talet fann *Euler* att det var sammansatt.

Kör programmet **PrimtalsTest** (sid 87) för att visa att talet är sammansatt. Ange talets primfaktorer genom att köra programmet **PrimFaktorer** (sid 95).

1902

Skriv ett program som kodar följande algoritm – en metod för att avgöra om ett tal är primtal *(Eratosthenes såll)*:

```
1. Läs in ett positivt heltal
   mellan 1 och 100.

2. Är det siffran 1 är det var-
   ken ett primtal eller ett
   sammansatt tal.

3. Är det siffran 2, 3, 5 eller
   7 är det ett primtal, annars:

4. Om det är delbart med 2 är det
   ett sammansatt tal, annars:

5. Om det är delbart med 3 är det
   ett sammansatt tal, annars:

6. Om det är delbart med 5 är det
   ett sammansatt tal, annars:

7. Om det är delbart med 7 är det
   ett sammansatt tal, annars:

8. Är det ett primtal.
```

Testa programmet för olika startvärden. Kommer programmet att ge korrekt svar om man matar in startvärden som är > 100? Testa med 121 och 169. Är programmets svar korrekt? Avgör med programmet **PrimtalsTest** (sid 87) om 121 och 169 är primtal eller ej. Förklara.

1903

Talet 1 uppfyller definitionen för primtal. Ändå sägs det i algoritmen i övn **1902**, punkt 2, att 1 varken är ett primtal eller ett sammansatt tal. Varför? Försök att hitta en förklaring.

1904

Vidareutveckla programmet i övn **1902** så att man med säkerhet kan avgöra om tal mellan 1 och 200 är primtal eller ej. Testa igen för 121 och 169. Hur går det för tal > 200? Testa för 529. Avgör med **PrimtalsTest** (sid 87) om 529 är primtal. Motivera algoritmen i ditt program.

1905

Programmet **AllaPrimtal** skriver ut alla primtal i ett givet positivt heltalsintervall (sid 91). Modifiera det så att det nya programmet frågar efter antalet n primtal och listar ut de första n primtalen i en lista som börjar med 2. Importera modulen **PrimFkt** och anropa funktionen **primtest()** för att avgöra om ett tal är primtal (sid 90).

1906

Antalet primtal $\leq n$ brukar man beteckna med $\pi(n)$. T.ex. är $\pi(10) = 4$ eftersom 2, 3, 5 och 7 är de enda primtalen ≤ 10. Skriv en funktion **antalPrim(n)** som beräknar $\pi(n)$. Anropa funktionen i ett program för att skriva ut antalet primtal $\leq n$ för:

$$n = 10, 10^2, 10^3, 10^4$$

Tips: Använd och modifiera koden i programmet **AllaPrimtal** (sid 91).

1.10 Rekursion

I programmet `PrimFaktorer` anropar funktionen **`faktorisera()`** sig själv, närmare bestämt i definitionen (sid 95, rad **16**). Innan dess har algoritmen Primtalsfaktorisering gjort det i sin sista rad. Utan tvekan har rekursion gjort formuleringen av denna algoritm enklare och begripligare vilket i sin tur lett till implementering av en rekursiv funktion i Python. Rekursiva funktioner är sådana som anropar sig själva när de definieras.

Rekursion är ett koncept som ofta används i datoriserad problemlösning genom successiv upprepning. Ordet rekursiv kommer från det latinska *recurrere* som på engelska betyder *to run back* eller *to run again*. Man återvänder till en struktur som redan finns och är beprövad. Se även algoritmen Intervallhalvering (sid 59). Rekursiva algoritmer genererar ofta kort och elegant kod som är nära matematisk notation.

En annan typ av problem som kan formuleras med rekursion är den klassiska uppgiften om kaniners fortplantning som den italienske matematikern *Leonardo Pisano Fibonacci* beskrev i sin bok *Liber abaci* (Boken om räknekonsten) år 1202:

> Ett kaninpar föder från den andra månaden av sin tillvaro ett nytt par varje månad. Samma gäller för de nya paren.
> Hur många par kommer det att finnas om ett år?

Fibonacci hade väl knappast kunnat drömma om att hans problem skulle bli föremål för datoriserad lösning med rekursion mer än 800 år senare. Om vi följer uppgiftens lydelse och räknar fram de första månaderna får vi följande:

Antal månader	1	2	3	4	5	6	7	8	...
Antal kaninpar	1	1	2	3	5	8	13	21	...

I den andra raden av tabellen uppstår en talföljd som kallas för *Fibonaccis talföljd* eller kort *fibonaccitalen*. Så här kommer de till enligt uppgiftens lydelse ovan:

De två första månaderna finns det 1 kaninpar. De föder sitt första barnpar först efter 2 månader dvs i månad nr 3, varför det finns 2 kaninpar i månad 3. I månad 4 föder det första paret sitt andra barnpar, varför det finns 3 par i månad 4. I månad

5 föder det första paret sitt tredje barnpar, men även deras första barnpar föder ett nytt par, eftersom det har gått 2 månader sedan deras födelse. Därför finns det 5 par i månad 5. Osv. ...

Praktiskt blir det allt svårare att hålla reda på antalet kaninpar när antalet månader växer. Man måste kanske rita någon sorts diagram och anteckna allt från månad till månad. Ett sätt att slippa det är att leta efter ett mönster, en struktur, en slags laglighet i bildandet av fibonacciföljden som kan beskrivas matematiskt. Undersöker man tabellens andra rad noga kan man upptäcka följande mönster:

> Summan av två på varandra följande fibonaccital ger nästa fibonaccital.

Kan man beskriva detta mönster matematiskt? Ja, med en s.k. *rekursionsformel*.

Vi inför beteckningarna:

n = Antalet månader
F(n) = Antalet kaninpar i månaden n

Mönstret som beskrevs ovan kan nu formuleras så här:

Fiboaccis rekursionsformel

$$F(n) = \begin{cases} 1 & \text{om } n = 1 \\ 1 & \text{om } n = 2 \\ F(n-1)+F(n-2) & \text{om } n = 3, 4, 5, \ldots \end{cases} \qquad n \text{ heltal}$$

Formeln har två startvärden som står i de första två raderna. Det är de första två fibonaccitalen *F(1)* och *F(2)* som är 1. Den tredje raden säger att det n-te fibonaccitalet är summan av de två föregående. Utgående från de två första startvärdena *F(1)* och *F(2)* kan vi beräkna alla andra successivt.

Men vad är det *rekursiva* i denna formel? I vanliga, icke-rekursiva formler står den sökta storheten vänster om likhetstecknet och alla givna storheter höger om likhetstecknet. Men här står den sökta storheten, fibonaccitalen, på *båda* sidor likhetstecknet, fast för olika månader. För att beräkna ett fibonaccital måste man redan ha beräknat de två föregående. Detta resulterar, när vi kodar formeln, i en funktion som anropar sig själv, fast med olika månader, dvs olika aktuella parametrar (sid 74).

Både den rekursiva funktionen som implementerar rekursionsformeln och ett program som anropar den, visas här:

```
1   # Fibonacci.py
2   # Definietrar rekursiv funktion fib() som ger fibonaccitalen
3   # Anropar fib() för de första 30 fibonaccitalen
4
5   def fib(n) :
6       if n <= 2 :
7          return 1
8       else :
9          return fib(n-1) + fib(n-2)       # Rekursiva anrop
10                                          # i definitionen
11  FibonacciLista = [ ]
12  print('\nDe första 30 fibonaccitalen är:\n\t')
13  for i in range(1, 31) :
14      FibonacciLista = FibonacciLista + [fib(i)]  # 30 anrop
15  print(FibonacciLista, '\n')
```

I raderna **5-9** definieras funktionen **fib()**. Koden är en implementering av Fibonaccis rekursionsformel i pythonkod. Därför är den också väldigt kort: För **n = 1** och **2** returneras **1** som enligt formeln är de första två fibonaccitalen. För alla andra **n** returneras summan av de två föregående dvs **fib(n-1) + fib(n-2)**. Men de i sin tur är var och en, anrop av **fib()** som båda står i kroppen till funktionen **fib()**, vilket är just det rekursiva.

Ett anrop av **fib(4)** t.ex. resulterar i att **fib(3)** och **fib(2)** anropas, **fib(3)** i sin tur resulterar i att **fib(2)** och **fib(1)** anropas, osv. Varje anrop av funktionen resulterar i ett stort antal följdanrop. Växer **n** leder det till en väldigt stor mängd av beräkningar. För stora fibonaccital är tidsåtgången stor.

I raderna **11-15** står programmet som anropar funktionen **fib()**. Det sker 30 gånger i `for`-satsen (rad **13-14**). Dess räknare **i** blir funktionens aktuella parameter, vilket genererar de första 30 fibonaccitalen. Med hjälp av en lista skrivs de ut.

Här följer programmet **Fibonacci**:s körresultat:

```
De första 30 fibonaccitalen är:

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
317811, 514229, 832040]
```

Så kan vi besvara den inledande frågan: Det kommer att finnas 144 kaninpar om ett år.

Nackdelen med rekursion

Rekursiva funktioner kan ha en stor beräkningskomplexitet, speciellt när de bygger på rekursionsformler. Funktionen **fib(n)**:s tidskomplexitet är av typ 2^n, dvs är exponentiellt växande, vilket kan för stora n bli väldigt ineffektivt. För stora **n** är det lämpligare att använda en alternativ, icke-rekursiv, t.ex. en iterativ implementering av Fibonaccis rekursionsformel. Därmed är det inte sagt att rekursiva funktioner alltid är ineffektiva. Det finns problem som enklast löses med rekrursion, t.ex. att manipulera datastrukturer som träd och grafer. Det finns t.o.m. problem där rekursiva metoder leder till effektivare lösningar än alternativa icke-rekursiva algoritmer, t.ex. sortering. Ett annat problem är hur svårt det är att beskriva och implementera dessa algoritmer.

I fallet av primtalsfaktoriseringen visar rekursion däremot vara effektiv eftersom både algoritmens formulering och dess implementering blev kortare, enklare och bättre förståelig just pga rekursionen. Algoritmen är av en annan typ än Fibonaccis problem: den bygger inte på en rekursionsformel där varje elements beräkning kräver beräkningen av alla tidigare element.

Övningar

1.10 Rekursion

1011

Följande algoritm beräknar summan av de första n positiva heltalen som en rekursiv funktion:

```
Funktion sum(n)
   OM n = 1
      returnera 1
   ANNARS
      returnera n + sum(n-1)
```

`sum(n-1)` betyder ett anrop av funktionen `sum()` med parametern `n-1`.

a) Varför är funktionen rekursiv?
b) I vilken ordning adderar algoritmen de pos. heltalen, fram- eller baklänges? Förklara varför den gör så.
c) Skriv algoritmen som en funktion i Python. Bygg in funktionen i ett program och anropa den för n = 10 och för n = 100.

1012

Följande algoritm beräknar siffersumman för ett positivt heltal:

```
Funktion siffSum(n)
   OM n < 10
      returnera n
   ANNARS
      rest = n % 10
      m = (n - rest) // 10
      returnera rest + siffSum(m)
```

a) Gå igenom algoritmen steg för steg för n = 385 och förklara vad den gör.
b) Skriv ett program som definierar funktionen `siffSum()` och anropar den för n = 385 och andra heltal som läses in.
c) Kontrollera ditt programs resultat genom att bilda $n \% 9$ vilket ger n:s siffersumma resp. siffersummans siffersumma, jfr. övn **1407** (sid 52).

1013

Euklides algoritm beräknar den största gemensamma faktorn (**gcd** = greatest common divisor) av de positiva heltalen a och b. Här är en rekursiv beskrivning av den:

```
Funktion gcd(a, b)
   OM b = 0
      returnera a
   ANNARS
      rest = a % b
      returnera gcd(b, rest)
```

a) Ställ upp ett räkneschema för manuellt genomförande av algoritmen och demonstrera den för a = 60 och b = 48.
b) Skriv en funktion **`gcd()`** i Python och anropa den för a = 60 och b = 48 från ett program som kan läsa in även andra heltal.
c) Jämför ditt programs resultat med körresultat från övn **1506**:s lösning som kodar Euklides algoritm på ett iterativt sätt. Förklara varför man får identiska resultat fast **1506** använder upprepad subtraktion medan funktionen **`gcd()`** räknar med modulooperatorn?
d) Skriv en annan version av funktionen **`gcd()`** som använder upprepad subtraktion istället för modulo.

Lösningar

Anmärkningar till lösningsdelen

- Denna del av boken består av fullständiga lösningsförslag till alla övningar.

- Till skillnad från matematiken har uppgifter i programmering ofta inte en entydig lösning. Man kan koda lösningen på många olika sätt.Vilket som är bäst må vara föremål för diskussion. Därför ges här endast *förslag* till lösningar. Bokens lösningsförslag är framtagna genom val av främst följande kriterier:

    - Enkelhet
    - Effektivitet
    - Allmängiltighet
    - Användarvänlighet
    - God programmeringsstil (sid 18)

- I slutet av varje *avsnitt* finns övningar. T.ex. är 1.1 Aritmetiska uttryck avsnitt 1 i kursen Matematik 1.

- Bokens övningar är identiska med övningarna i appen *Mattekollen* (sid 156). I vissa fall kan det dock förekomma små skillnader eftersom den mobila pythonmiljön ibland beter sig lite annorlunda än Interactive mode. Vill man ha renodlad Interactive mode bör man använda sig av Python interpretatorn (sid 14).

Python interpretatorn – övn sid 16

0101

Ladda ner den senaste versionen av Python från Pythons officiella webbsida och installera den på din dator.

```
Gå till sid 14 och följ anvisningarna där.
```

0102

Har du en Mac-dator leta på nätet efter instruktioner för installation av Python på Mac. Gör samma sak som i övn **0101**.

```
Öppna din webbläsare, gå till adressen www.python.org/downloads.
Klicka på knappen Download Python... . Installationsfilen *.pkg laddas
ner. Dubbelklicka på den. Följ instruktionerna.

Efter installationen öppnas mappen Python x.x som bl.a. innehåller fi-
len IDLE.app. Dra en genväg från denna fil till ditt skrivbord. När du
dubbelklickar på den öppnas Python interpretatorn. Man kan lämna den
med exit().
```

0103

a) Vad är skillnaden mellan kompilering och interpretering?
b) Är Python ett kompilerande eller ett interpreterande programspråk?

```
a)  Kompilering innebär översättning av källkod till maskinkod där
    maskinkoden lagras i en fil som sedan kan exekveras.Interpretering
    innebär tolkning av källkod till maskinkod där maskinkoden direkt
    exekveras i datorn utan att spara maskinkoden.
b)  Python är ett interpreterande programmeringsspråk.
```

0104

Öppna Python interpretatorn. Skriv *Hej* i prompten >>> . Vad händer och varför? Skriv sedan *'Hej'*. Vad händer nu? Skriv slutligen *"Hej"*. Förklara och dra slutsats.

```
>>> Hej
Traceback (most recent call last):
                  File “<stdin>”, line 1, in <module>
NameError: name 'Hej' is not defined
>>> 'Hej'
'Hej'
>>> "Hej"
'Hej'
```

```
Koden Hej ger felmeddelande eftersom Python inte kan tolka den. Hej har
inte definierats innan. Koden 'Hej' tolkas som texten Hej och återges.
Samma händer med "Hej". Slutsats: Text kan i Python även omges av "  ".
```

0105

Öppna *Interactive mode* (sid 15) och skriv först kod som skriver ut bokstaven a, sedan kod som skriver ut talet 3.

```
>>> print('a')
a
>>> print(3)
3
```

Eller:

```
>>> 'a'
'a'
>>> 3
3
```

0106

Skriv i Interactive mode först koden print('3') och sedan print(3). Båda ger utskriften 3. Men vad är skillnaden?

```
>>> print('3')
3
>>> print( 3 )
3

Skillnaden är att print('3') skriver ut tecknet 3 medan print( 3 )
skriver ut talet 3.
```

0107

Testa i Interactive mode koden `Print('Hej')`. Vad är det för fel med denna sats? Rätta till koden.

```
      >>> Print('Hej')
      Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      NameError: name 'Print' is not defined

Felet är att den fördefinierade print()-satsen i Python skrivs med
litet p. Python är case sensitive (sid 31). Den korrekta koden är:

      >>> print('Hej')
      Hej
```

0108

a) Skriv ut i Interactive mode endast texten Hej utan någon tillsats.
b) Snygga till din utskrift med hjälp av koderna \n och \t så att Hej hamnar med ett avstånd från den vänstra kanten samt med en tom rad före och efter.

```
a)   >>> print('Hej')
     Hej

b)   >>> print('\n\tHej\n')

             Hej

     >>>
```

0109

Skriv pythonkod som skriver ut: `Hej, välkommen till Python!` Testa din kod.

```
>>> print('Hej, välkommen till Python!')
Hej, välkommen till Python!
```

0110

Modifiera övn 0109 så att utskriften blir:

```
Hej,
välkommen till Python!
```

```
>>> print('Hej,\nvälkommen till Python!')
Hej,
välkommen till Python!
```

0111

Modifiera övn 0110 så att utskriften blir:

```
Hej,
välkommen
till
Python!
```

```
>>> print('Hej,\nvälkommen\ntill\nPython!')
Hej,
välkommen
till
Python!
```

Pythons utvecklingsmiljö IDLE – övn sid 20

0201

Öppna din favorit utvecklingsmiljö för Python, mata in koden till programmet **Welcome** och kör så att utskriften blir exakt som på sid 19.

Lösning i IDLE (sid 17):

Öppna Pythons egen utvecklingsmiljö IDLE som följer med vid nedladdning av Python. Gå till menyraden längst upp och klicka på: File → New File. Mata in i editfönstret följande kod:

```
print('\n\t Välkommen till \n'                  ,
      '\n\t Koda matte med Python! \n'          ,
      '\n\t Programmering i matematik \n'       ,
      '\n\t En handbok för lärare och elever \n')
```

Klicka i menyraden på: File → Save. Välj valfri plats på din dator och döp filen till Welcome.py. Klicka i menyraden på: Run → Run Module.

Lösning i Visual Studio (sid 159):

Öppna Visual Studio och klicka i menyraden på menyn:File → New → File. Expandera i dialogrutan New File i den vänstra kolumnen Installed och markera Python. Markera i den mellersta kolumnen Empty Module och klicka på Open. Klicka på menyn: File → Save Selected Items As… . Välj valfri plats på din dator och döp filen till Welcome.py. Klicka på Save. Mata in koden ovan. Klicka i menyraden på menyn: Project → Start Without Debugging.

0202

Modifiera övn **0201** så att utskriften blir:

```
Välkommen till
Koda matte med Python!
Programmering i matematik
En handbok för lärare och elever
```

```
# 0202.py

   print('\n\t Välkommen till'                    ,
         '\n\t Koda matte med Python!'            ,
         '\n\t Programmering i matematik'         ,
         '\n\t En handbok för lärare och elever \n')
```

0203 a)

Modifiera övn **0202** genom att i koden ersätta alla apostrofer (') med citationstecknet ("). Vilken slutsats drar du om kodning av strängar i Python?

```
# 0203 a).py

print("\n\t Välkommen till"                    ,
      "\n\t Koda matte med Python!"            ,
      "\n\t Programmering i matematik"         ,
      "\n\t En handbok för lärare och elever \n")

Slutsats: Strängar (text) kan i pythonkod även omges av citationsteck-
net.
```

0203 b)

Undersök skillnaderna mellan apostrof, citationstecken och accent. ...

```
På tangentbordet:

   Apostrof är              '      (Tillsammans med * tangenten)
   Citationstecken är       "      (Tillsammans med 2:ans tangent)
   Två typer av accent är´ och '   (Till höger om   + tangenten)
```

```
I Python:

   >>> print('´')
   ´
   >>> print('''')
   '
   >>> print('\'')
   '
```

```
>>> print('\"')
"
>>> print('\\')
\
>>>
```

0204

Använd kunskapen från övn **0203 b)** för att skriva pythonkod som skriver ut:

```
"Adjö" skrivs på franska "Adieu".
I Python omges text av ' eller ".
\n ger newline, \t ger tabulator.
\b ger backspace, \\ ger backslash.
```

```
# 0204a.py

print('\n\t "Adjö" skrivs på franska "Adieu".            \n'  ,
      '\n\t I Python omges text av \' eller ".           \n'  ,
      '\n\t \\n ger newline, \\t ger tabulator.          \n'  ,
      '\n\t \\b ger backspace, \\\\ ger backslash.       \n'  )
```

Eller:

```
# 0204b.py

   print("\n\t \"Adjö\" skrivs på franska \"Adieu\".  \n"  ,
         "\n\t I Python omges text av ' eller \".     \n"  ,
         "\n\t \\n ger newline, \\t ger tabulator.    \n"  ,
         "\n\t \\b ger backspace, \\\\ ger backslash. \n"  )
```

0205

Skriv ett pythonprogram som skriver ut en triangel byggd av stjärnor.

```
# 0205.py
# Skiljer åt delsträngarna på olika rader med komma

   print('\n\t               *          '    ,
         '\n\t              * *         '    ,
         '\n\t             *   *        '    ,
         '\n\t            *     *       '    ,
         '\n\t           *       *      '    ,
         '\n\t          *         *     '    ,
         '\n\t         *           *    '    ,
         '\n\t        *             *   '    ,
         '\n\t       *               *  '    ,
         '\n\t      * * * * * * * * * * '    ,
    '\n\n')
```

0206

Skriv ett pythonprogram som ritar en hjärtlig hälsning på skärmen.

```
# 0206.py
# Slår ihop delsträngarna med konkateneringsoperatorn + (sid 39)

   print('\n\t         *       *         '    +
         '\n\t       *   *   *   *       '    +
         '\n\t     *       *       *     '    +
         '\n\t    *                 *    '    +
         '\n\t    *                 *    '    +
         '\n\t    *     Grattis     *    '    +
         '\n\t     *                *     '    +
         '\n\t       *            *       '    +
         '\n\t         *        *         '    +
         '\n\t           *   *           '    +
         '\n\t             *             '    +
         '\n\n')
```

1.1 Aritmetiska uttryck – övn sid 28

1101

Använd Python som smart kalkylator för att beräkna följande aritmetiska uttryck:

a) $7 + 4 \cdot 2$ b) $9 - 8 / 4$ c) $12 + 18 / 9 - 6$ d) $\dfrac{12 + 18}{9 - 6}$

```
>>>
>>> # a)
>>> 7 + 4 * 2
15
>>>
>>> # b)
>>> 9 - 8 / 4
7.0
>>>
```

```
>>> # c)
>>> 12 + 18 / 9 - 6
8.0
>>>
>>> # d)
>>> (12 + 18) / (9 - 6)
10.0
>>>
```

1102

Räkna först utan Python och kontrollera sedan dina resultat med Python:

a) $5 + 3 \cdot 8 - 6$ b) $(5 + 3) \cdot (8 - 6)$

c) $3(6 - 4) + 2(5 - 2)$ d) $6\,(3 + 1 \cdot 2) - 4 \cdot 5$

Utan Python:

a) $5 + 3 \cdot 8 - 6 = 5 + 24 - 6 = 29 - 6 = \underline{23}$

b) $(5 + 3) \cdot (8 - 6) = 8 \cdot 2 = \underline{16}$

c) $3(6 - 4) + 2(5 - 2) = 3 \cdot 2 + 2 \cdot 3 = \ 6 + 6 = \underline{12}$

d) $6\,(3 + 1 \cdot 2) - 4 \cdot 5 = 6 \cdot (3 + 2) - 20 = 6 \cdot 5 - 20 = 30 - 20 = \underline{10}$

Med Python:

```
>>>
>>> # a)
>>> 5 + 3 * 8 - 6
23
>>>
>>> # b)
>>> (5 + 3) * (8 - 6)
16
```

```
>>> # c)
>>> 3 * (6 - 4) + 2 * (5 - 2)
12
>>>
>>> # d)
>>> 6 * (3 + 1 * 2) - 4 * 5
10
>>>
```

1103

Ett taxibolag tar en fast avgift på 25 kr. Därefter kostar det 10 kr per km att åka. Beräkna med Python hur mycket det kostar att åka 20 km. Ställ upp ett aritmetiskt uttryck för taxan om man åker x km.

```
>>>
>>> 25 + 10 * 20
225
>>>
>>> # Det kostar 225 kr att åka 20 km med taxin.
```

Aritmetiskt uttryck: taxa = 25 + 10 * x

1104

Ett mobilabonnemang har en öppningsavgift på 1,50 kr. Det kostar 25 öre per minut att ringa. Ställ upp ett aritmetiskt uttryck för kostnaden om man ringer x sekunder. Beräkna uttryckets värde för x = 59 i Python, dvs låt Python beräkna hur mycket det kostar att ringa 59 sek.

Aritmetiskt uttryck:

25 öre per min = $\frac{0,25}{60}$ kr per sek

Kostnad = $1,50 + \frac{0,25}{60} \cdot x$

Beräkning med Python:

```
>>> x = 59
>>>
>>> 1.50 + 0.25 / 60 * x
1.7458333333333333
>>>
>>> # Det kostar 1,75 kr att
>>> # ringa 59 sekunder.
```

1105

Beräkna följande aritmetiska uttryck utan digitalt verktyg. Kontrollera dina resultat i Pythons Interactive mode:

a) $(-5)^2 - 3^2$ b) $\frac{2^2 + 3^2}{(2+3)^2}$ c) $\frac{(4-2)^2}{4^2 - 2^2}$ d) $(2^2)^3$ c) 2^{2^3}

Utan digitalt verktyg:

a) $(-5)^2 - 3^2 = (-5) \cdot (-5) - 3 \cdot 3 = 25 - 9 = 16$

b) $\frac{2^2 + 3^2}{(2+3)^2} = (2^2 + 3^2) / (2+3)^2 = (4+9) / 5^2 = 13 / 25 = \frac{13}{25}$

c) $\frac{(4-2)^2}{4^2 - 2^2} = (4-2)^2 / (4^2 - 2^2) = 2^2 / (16 - 4) = 4 / 12 = \frac{1}{3} = 0,333...$

d) $(2^2)^3 = 4^3 = 4 \cdot 4 \cdot 4 = 16 \cdot 4 = 64$

e) $2^{2^3} = 2^{(2^3)} = 2^8 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 4 \cdot 4 \cdot 4 \cdot 4 = 16 \cdot 16 = 256$

Med Python:

```
>>>
>>> # a)
>>> (-5)**2 - 3**2
16
>>>
>>> # b)
>>> (2**2 + 3**2) / (2 + 3)**2
0.52
>>> 13 / 25
0.52
```

```
>>> # c)
>>> (4 - 2)**2 / (4**2 - 2**2)
0.3333333333333333
>>>
>>> # d)
>>> (2**2)**3
64
>>> # e)
>>> 2**2**3
256
>>>
```

1106

Låt ett pythonprogram beräkna och skriva ut följande uttryckets värde för $x = -1$:

$$4x^3 - 2x^2(2x + 6) + 7x(3 + 2x)$$

```
# 1106.py

x = -1
uttryck_1108 = 4*x**3 - 2*x**2 * (2*x + 6)+ 7*x*(3 + 2*x)

print('\n\t Uttrycket 4x^3 - 2x^2(2x+6) + 7x(3+2x)'
      '\n\n\t har för x = -1 värdet:     '  , uttryck_1108, '\n')
```

1107

a) Beräkna i Python följande uttryck: $(6^2 + 6^2 + 6^2) / 9$

b) Kontrollera Pythons svar genom att förenkla uttrycket och beräkna det utan digitalt verktyg.

a) Python:

```
>>>
>>> # a)
>>>
>>> (6**2 + 6**2 + 6**2) / 9
12.0
>>>
```

b) Förenkling:

$$(6^2 + 6^2 + 6^2) / 9 = 3 \cdot 6^2 / 9 =$$

$$\frac{3 \cdot 6^2}{9} = \frac{6^2}{3} = \frac{36}{3} = \underline{12}$$

1108

Skriv ett pythonprogram i en fil som beräknar och skriver ut de aritmetiska uttryckens värden från övn **1102** a) - d) och övn **1105** a) - e).

```
# 1108.py
# Beräknar och skriver ut de aritmetiska uttrycken
# från övn 1102 och # 1105

   uttryck_1102a = 5 + 3 * 8 - 6
   uttryck_1102b = (5 + 3) * (8 - 6)
   uttryck_1102c = 3 * (6 - 4) + 2 * (5 - 2)
   uttryck_1102d = 6 * (3 + 1 * 2) - 4 * 5
   uttryck_1105a = (-5)**2 - 3**2
   uttryck_1105b = (2**2 + 3**2) / (2 + 3)**2
   uttryck_1105c = (4 - 2)**2 / (4**2 - 2**2)
   uttryck_1105d = (2**2)**3
   uttryck_1105e = 2**2**3

   print('\n\t 5 + 3 * 8 - 6 ='  , uttryck_1102a,
         '\n\n\t (5 + 3) * (8 - 6) =', uttryck_1102b,
         '\n\n\t 3 * (6 - 4) + 2 * (5 - 2) =', uttryck_1102c,
         '\n\n\t 6 * (3 + 1 * 2) - 4 * 5 =', uttryck_1102d,
         '\n\n\t (-5)**2 - 3**2 =', uttryck_1105a,
         '\n\n\t (2**2 + 3**2) =', uttryck_1105b,
         '\n\n\t (4 - 2)**2 / (4**2 - 2**2) =', uttryck_1105c,
         '\n\n\t (2**2)**3 =', uttryck_1105d,
         '\n\n\t 2**2**3 =', uttryck_1105e, '\n')
```

1109

Experimentera med Python som smart kalkylator för att lösa följande uppgift. Hitta ett positivt heltal för x så att följande uttryckets värde blir störst. Beräkna detta maximala värde. Motivera.

$$\frac{87+13}{(x+9)/5}$$

Python:

```
>>> x = 1
>>> (87 + 13) / ((x + 9) / 5)
50.0
>>>
>>> x = 2
>>> (87 + 13) / ((x + 9) / 5)
45.45454545454545
>>>
>>> x = 3
>>> (87 + 13) / ((x + 9) / 5)
41.66666666666667
>>>
>>> x = 4
>>> (87 + 13) / ((x + 9) / 5)
38.46153846153846
>>>
>>> x = 5
>>> (87 + 13) / ((x + 9) / 5)
35.714285714285715
```

Svar:

x = 1 och max.-värdet = 50

Motivering:

För x = 1 blir uttryckets värde störst dvs 50, därför att värdet minskar för alla följande positiva heltal 2, 3, 4, 5,…, se Python.

Uttryckets värde kommer även att minska för alla pos. heltal > 5 därför att x står i uttryckets nämnare.

Ju större x som man delar med, desto mindre blir ut trycket. x = 1 är det minsta positiva heltalet.

1.2 Variabler – övn sid 34

1201

Skriv ett program som skriver ut: `Summan av 5 och 3 är 8`
Lös uppgiften genom att skapa två variabler som får värdena 5 och 3. Tilldela deras summa till en tredje variabel och skriv ut som ovan. Utskriften ska ske med hjälp av variablerna.

```
# 1201.py

tal1 = 5
tal2 = 3
sum = tal1 + tal2

print('\n\t Summan av', tal1, 'och', tal2, 'är', sum, '\n')
```

1202

Utveckla övn **1201** vidare så att utskriften blir:

```
Summan av 5 och 3 är 8

Differensen 5 - 3 är 2
```

Använd samma variabler som i **1201**. Lägg till endast en variabel mer för differensen. Skriv ut med hjälp av variablerna.

```
# 1202.py

   tal1 = 5
   tal2 = 3
   sum  = tal1 + tal2
   diff = tal1 - tal2
   print('\n\t Summan av'  , tal1, 'och', tal2, 'är', sum,  '\n'
         '\n\t Differensen', tal1, '-',   tal2, 'är', diff, '\n')
```

1203

Komplettera lösningen till övn **1202** så att utskriften blir:

```
5  +  3  ger  8
5  -  3  ger  2
5  *  3  ger  15
5  /  3  ger  1.6666666666666667
5  // 3  ger  1
```

Dvs skapa ytterligare variabler, tilldela till dem de fyra räknesätten. Inkludera Pythons operator `//` för heltalsdivision (sid 22). Skriv ut med hjälp av variablerna.

```
# 1203.py

    tal1 = 5
    tal2 = 3
    sum  = tal1 + tal2
    diff = tal1 - tal2
    prod = tal1 * tal2
    div  = tal1 / tal2                    # Vanlig division
    intDiv  = tal1 // tal2                # Heltalsdivision
    print('\n\t', tal1, ' + ' , tal2, ' ger ', sum   , '\n\t',
                 tal1, ' - ' , tal2, ' ger ', diff  , '\n\t',
                 tal1, ' * ' , tal2, ' ger ', prod  , '\n\t',
                 tal1, ' / ' , tal2, ' ger ', div   , '\n\t',
                 tal1, ' //' , tal2, ' ger ', intDiv, '\n  ' )
```

1204

Varför ger följande kod i Python felmeddelande?
Förklara felets orsak. Åtgärda felet.

```
tal = 1
sum = sum + tal
print('\n\t sum =',  sum, '\n')
```

```
# 1204.py

tal = 1                       # Felets orsak: Variabeln sum måste tilldelas
sum = 9                       # ett värde innan den används i satsen
sum = sum + tal               # sum = sum + tal till höger om = .
print('\n\t sum =', sum,'\n') # Åtgärd: Vi tilldelar sum t.ex. värdet 9.
```

1205

Vilka värden kommer variablerna `tal` och `prod` att ha efter följande kod?

```
tal  = 5
prod = tal * 4
tal  = tal + tal + tal + tal
```

Svara först utan Python. Testa sedan i Interactive mode. För vilken matematisk kunskap är koden ett exempel på?

Utan Python:

prod = $5 \cdot 4$ = 20

tal = $5 + 5 + 5 + 5$ = 20

Slutsats: $5 \cdot 4 = 5 + 5 + 5 + 5$

Koden är ett exempel på:
Produkten $5 \cdot 4$ är upprepad addition av 5 med sig själv, 4 gånger.

Interactive mode:

```
>>> tal = 5
>>> prod = tal * 4
>>> tal = tal + tal + tal + tal
>>>
>>> tal
20
>>> prod
20
```

1206

Vilka värden kommer variablerna `tal` och `potens` att ha efter följande kod?

```
tal   = 5
potens = tal ** 4
tal   = tal * tal * tal * tal
```

Gör samma sak och besvara samma fråga som i övn **1205**.

Utan Python:

$potens = 5^4 = 5 \cdot 5 \cdot 5 \cdot 5 = 25 \cdot 25 = 625$

$tal = 5 \cdot 5 \cdot 5 \cdot 5 = 25 \cdot 25 = 625$

Slutsats: $5^4 = 5 \cdot 5 \cdot 5 \cdot 5$

Koden är ett exempel på:

Potensen a^b är en upprepad multiplikation av basen a med sig själv, b gånger.

Interactive mode:

```
>>> tal = 5
>>> potens = tal ** 4
>>> tal = tal * tal * tal * tal
>>>
>>> tal
625
>>> potens
625
```

1207

Vilka värden får c och d efter följande kod? Svara först. Testa sedan i Python.

```
a = 123456789
b = a
c = b ** (a - b)
d = c // (b - a)
```

Utan Python:

$a = 123456789$

$b = 123456789$

$a - b = 0$

$c = b^{(a-b)} = b^0 = 1$

Interactive mode:

```
>>> a = 123456789
>>> b = a
>>> c = b ** (a - b)
>>>
>>> c
1
```

Utan Python:

$b - a = 0$

$d = c / (b - a) = c / 0$

d är inte definierat eftersom c/0 pga division med 0 inte är definierat.
Detta gäller även för heltalsdivision (//).

Interactive mode:

```
>>> d = c // (b - a)

Traceback (most recent call last):
  File "<pyshell#5>", line 1, in
<module>
    d = c // (b - a)
ZeroDivisionError: integer divi-
sion or modulo by zero
>>>
```

1.3 Inläsning av data – övn sid 41

1301

Satsen `print(a)` ger felmeddelande. Testa i ett pythonprogram vilka utskrifter följande satser ger:

```
print('a')              print('6' + '6')
print('a' + 'a')       print(6 + 6)
print('a', 'a')        print(6, 6)
print(6)               print(6.6 + 6.6)
print('6')             print('6.6' + '6.6')
```

Förklara resultaten.

```
# 1301.py

# print(a)                    # Ger felmeddelande
print('a')                    # Bokstaven a:                a
print('a' + 'a')              # Konkatenering:              aa
print('a', 'a')               # Två tecken:                 a a
print(6)                      # Talet 6:                    6
print('6')                    # Tecknet 6:                  6
print('6' + '6')              # Konkatenering:              66
print(6 + 6)                  # Addition av 6 med 6:        12
print(6, 6)                   # Två tal:                    6 6
print(6.6 + 6.6)              # Addition av 6.6 med 6.6:    13.2
print('6.6' + '6.6')          # Konkatenering:              6.66.6
```

Förklaringar:

```
print(a) ger felmeddelande därför att a är en odefinierad variabel.
print('a') ger a därför att 'a' är ett tecken: bokstaven a.
print('a' + 'a') ger aa därför att + slår ihop bokstäverna a och a.
print('a', 'a') ger a a därför att , gör att bokstäverna a och a skrivs ut.
print(6) skriver ut talet 6.
print('6') skriver ut tecknet 6.
print('6' + '6') ger 66 därför att + slår ihop tecknen 6 och 6.
print(6 + 6) ger 12 därför att + adderar talen 6 och 6.
print(6, 6) ger 6 6 därför att , gör att talen 6 och 6 skrivs ut separat.
print(6.6 + 6.6) ger 13.2 därför att 6.6 är decimaltal och + adderar talen.
print('6.6' + '6.6') ger 6.66.6 därför att + slår ihop strängarna '6.6'.
```

1302

Modifiera programmet **Input** (sid 35) så att variabeln **x**:s ökade värde tilldelas en ny variabel **y**. Annars ska det nya programmet ge samma utskrift som programmet **Input**.

```
# 1302.py

text = input('\n\t Mata in ett heltal\t')     # Inläsning
x = int (text)                                # Omvandling till heltal
y = x + 1

print('\n\t Variabelns värde har lästs in som'  , x, '\n' ,
      '\n\t Sedan har det ökats med 1 och är nu', y, '.\n')
```

1303

Skriv ett program som läser in tre tecken och skriver ut dem i omvänd ordning. Använd datatypen **string** i ditt program (sid 40).

```
# 1303.py

text = input('\nMata in tre tecken skilda med mellanslag:\t')

tecken1 = text[0]
tecken2 = text[2]
tecken3 = text[4]

print('\nTecknen i omvänd ordning:\t\t       ',
               tecken3, tecken2, tecken1, '\n')
```

1304

Skriv ett program som läser in tre heltal och beräknar deras medelvärde. Programmet ska sedan skriva ut talen i omvänd ordning samt medelvärdet.

```
# 1304.py

tal1 = int(input('\nMata in ett heltal:\t\t'))
tal2 = int(input('\nMata in ett andra heltal:\t'))
tal3 = int(input('\nMata in ett tredje heltal:\t'))

medelvärde = (tal1 + tal2 + tal3) / 3

print('\nTalen i omvänd ordning:\t       ', tal3, tal2, tal1, '\n',
                  '\nMedelvärdet:\t\t       ',   medelvärde, '\n')
```

1305

Vidareutveckla lösningen till övn **1203** (sid 116) genom att ersätta de hårdkodade indata (5 och 3) med inläsning av data. Bilda med de inlästa värdena de fyra räknesätten inkl. heltalsdivision som i övn **1203**. Tilldela resultaten till variabler och skriv ut svaren.

```
# 1305.py

tal1 = int(input('\n\t Mata in ett heltal:    '))   # Inläsning och
tal2 = int(input('\n\t Mata in ett heltal till: ')) # omvandling till
                                                    # heltal
sum  = tal1 + tal2
diff = tal1 - tal2
prod = tal1 * tal2
div  = tal1 / tal2                                  # Vanlig division
intDiv  = tal1 // tal2                              # Heltalsdivision

print('\n\t', tal1, ' + ' , tal2, ' ger ', sum   , '\n\t',
             tal1, ' - ' , tal2, ' ger ', diff  , '\n\t',
             tal1, ' * ' , tal2, ' ger ', prod  , '\n\t',
             tal1, ' / ' , tal2, ' ger ', div   , '\n\t',
             tal1, ' //' , tal2, ' ger ', intDiv, '\n  ' )
```

1306

Följande program innehåller två fel. Hitta felen och åtgärda dem.

```
a = input(' Mata in ett heltal: ')
prod = a * b
print(a, '*', b, '=', prod, '\n')
```

```
# 1306.py

a = int(input('\n\t Mata in ett heltal:\t'))   # Inläsning och om-
b = 5                                          # vandling till heltal
prod = a * b
print('\n\t', a, '*', b, '=', prod, '\n')
```

Felen:

1. Variabeln a måste omvandlas till datatypen int innan den används i räknesammanhang i satsen prod = a * b .

2. Variabeln b måste tilldelas ett värde innan den används i satsen prod = a * b till höger om = .

1307

Skriv ett program som kodar följande algoritm (tillvägagångssätt):

1. Läs in ett positivt heltal.
2. Multiplicera med 8
3. Lägg till 12
4. Dividera med 4
5. Dra av 3
6. Multiplicera med 2

Lagra varje steg i en variabel. Skriv ut slutvärdet (steg 6.). Testa programmet genom att läsa in startvärdena 2, 5, 8 och 10. Finns det något enkelt samband mellan start- och slutvärdet? I så fall beskriv det. Kommer andra startvärden att visa samma samband? Testa gärna!

Bevisa sambandet matematiskt.

```
# 1307.py

steg1 = int(input('\n\t Mata in ett pos. heltal:\t'))    # Startvärde
steg2 = steg1 * 8
steg3 = steg2 + 12
steg4 = steg3 // 4
steg5 = steg4 - 3
steg6 = steg5 * 2                                        # Slutvärde

kvot = steg6 // steg1                    # Samband: steg6 = 4 * steg1

print('\n\t Algoritmens slutvärde:\t       ', steg6, '\n',
         '\n\t Kvoten mellan start- och slutvärde:\t', kvot, '\n')
```

Samband: Om startvärdet är x blir slutvärdet 4 x .

Bevis:

1.	Startvärde	x
2.	Multiplicera med 8	8 x
3.	Lägg till 12	8 x + 12
4.	Dividera med 4	(8 x + 12) / 4 = 2 x + 3
5.	Dra av 3	2 x
6.	Multiplicera med 2	4 x

1.4 Delbarhet – övn sid 52

1401

Försök att i Interactive mode beräkna följande uttryckets värde för $x = -9$. Tolka Pythons svar.

$$\frac{87+13}{(x+9)/5}$$

Python:

```
>>> x = -9
>>>
>>> (87 + 13) / ((x + 9) / 5)

Traceback (most recent call last):
  File "<pyshell#3>", line 1, in
<module>
    (87 + 13) / ((x + 9) / 5)
ZeroDivisionError: float division
by zero
   >>>
```

Tolkning:

För x = -9 blir uttryckets nämnare 0 därför att (-9 + 9) / 5 = 0 / 5 = 0.

Division med 0 är inte definierad och kan inte genomföras.

Därför ger Python ett felmeddelande: 'ZeroDivisionError'

1402

Ta över beräkningen av uttryycket i övn **1401** till ett program i en fil som beräknar uttryckets värde när man matar in ett värde för x. Bygg in i ditt program `if`-satser som förhindrar division med 0 om man matar in □9 för x på liknande sätt som i programmet **Division_0** (sid 42).

```
# 1402.py

x = int(input('\n\tMata in ett heltal:       '))

if x != -9 :
   y = (87 + 13) / ((x + 9) / 5)
   print('\n\tFör x =', x, 'blir y =', y, '\n')

if x == -9 :
   print('\n\tOBS!\n\tDu har matat in ett tal för vilket',
                 '\n\tuttrycket inte är definierat.   \n')
```

1403

Modifiera lösningen till övn **1402** ovan genom att ersätta `if`-satserna men en `if-else`-sats. Annars ska det nya programmet göra samma sak och ge samma utskrift som det gamla.

```
# 1403.py

x = int(input('\n\tMata in ett heltal:       '))

if x != -9 :
   y = (87 + 13) / ((x + 9) / 5)
   print('\n\tFör x =', x, 'blir y =', y, '\n')
else :
   print('\n\tOBS!\n\tDu har matat in ett tal för vilket',
                 '\n\tuttrycket inte är definierat.   \n')
```

1404

a) Marcus som är 1,75 m stor och väger 76 kg vill veta om han är överviktig. Enligt Body Mass Index (BMI) anses man vara överviktig om BMI > 25. Testa i Interactive mode om Marcus är överviktig med formeln:

$$\text{BMI} = \frac{\text{Vikt i kg}}{\left(\text{Längd i m}\right)^2}$$

```
a)   >>>
     >>> 76 / 1.75 ** 2
     24.816326530612244
     >>>
     >>> # Marcus är inte överviktig: BMI < 25
     >>>
```

b) Skriv ett program (BMI Calculator) som läser in vikten i kg och längden i cm och skriver ut Överviktig om BMI > 25 annars OK. Som kontroll skriv även ut BMI-värdet.

```
# 1404 b).py

vikt  = int(input('\n\tAnge vikt i hela kg:     '))
längd = int(input('\n\tAnge längd i hela cm:    ')) # Inläsning i cm

BMI = vikt / (längd/100) ** 2                      # Omvandling
print('\n\tBMI =', BMI)                            # till meter

if BMI > 25 :
   print('\n\tÖverviktig!\n')
else :
   print('\n\tOK!\n')
```

c) Gör samma sak som i b), men läs in längden i meter som decimaltal. För typomvandling använd funktionen **float()**, sid 37.

```
# 1404 c).py

vikt  = int(input('\n\tAnge vikt i hela kg:\t\t   '))   # Inläsning
                                                       # i meter
längd = float(input('\n\tAnge längd i m (OBS! Decimaltal):  '))

BMI = vikt / längd ** 2                                # Beräkning
print('\n\tBMI =', BMI)                                 # i meter

if BMI > 25 :
   print('\n\tÖverviktig!\n')
else :
   print('\n\tOK!\n')
```

1405

Skriv ett program som läser in tre heltal, hittar och skriver ut det största av dem. Vilken ändring i koden leder till det minsta talet?

```
# 1405.py

tal1 = int(input('\n\tMata in tal1:\t'))
tal2 = int(input('\n\tMata in tal2:\t'))
tal3 = int(input('\n\tMata in tal3:\t'))

max = tal1
if tal2 > max :
   max = tal2
if tal3 > max :
   max = tal3

print('\n\tDet största talet är', max, '\n')
```

```
Genom att byta ut alla > mot < får man det minsta talet.
```

1406

a) Testa i Python om 4592 är jämnt delbart med 7. Motivera.
b) Skriv en pythonkod som avgör om a är jämnt delbart med b.

a)

```
>>> 4592 % 7
0
>>>
>>> # 4592 % 7 = 0 innebär att 4592 delat med 7 ger resten 0.
>>> # Rest 0 vid division betyder att 4592 är jämnt delbart med 7.
```

```
>>> # Generellt:
>>> # a % b = 0 innebär att a är jämnt delbart med b.
```

b)

```
if a % b == 0 :
   print('a är jämnt delbart med b')
else :
   print('a är inte jämnt delbart med b')
```

1407

Testa följande algoritm i Pythons Interactive mode för flera startvärden:

1. Välj ett positivt heltal n .
2. Bilda n :s siffersumma och ev. siffersummans siffersumma.
3. Beräkna n % 9 i Pythons Interactive mode.

Beskriv dina observationer.

Python:

```
>>> n = 314
>>> # Siffersumman:
>>> 3 + 1 + 4
8
>>> 314 % 9
8
>>> n = 5326
>>> # Siffersumman:
>>> 5 + 3 + 2 + 6
16
>>> # Siffersummans siffersumma:
>>> 1 + 6
7
>>> 5326 % 9
7
```

```
>>> n = 101
>>> # Siffersumman:
>>> 1 + 0 + 1
2
>>> 101 % 9
2
>>> n = 34879
>>> # Siffersumman:
>>> 3 + 4 + 8 + 7 + 9
31
>>> # Siffersummans siffersumma:
>>> 3 + 1
4
>>> 34879 % 9
4
```

Observation:

```
Modulo 9 ger talets siffersumma resp. siffersummans siffersumma.
```

1408

Skriv ett program som kodar följande algoritm:

1. Läs in ett positivt heltal n .
2. Beräkna $d = (n+1)(n-1)$.
3. Om d är jämnt delbart med 8 skriv ut YES annars NO.

Testa programmet med 6 udda och 6 jämna startvärden. Beskriv resultatet.
Kan resultatet generaliseras? Om ja, bevisa det.

```
# 1408.py

n  = int(input('\n\tAnge ett positivt heltal:\t'))

d = (n + 1) * (n - 1)

if d % 8 == 0 :                  # Delbarhet med 8
   print('\n\tYES\n')
else :
   print('\n\tNO\n')
```

```
Resultat:     Udda startvärden ger YES, jämna startvärden ger NO.
```

Generellt: (n+1)(n-1) är alltid jämnt delbart med 8 om n är udda.

Bevis:
Om n är udda är, är både (n+1) och (n-1) jämna. Det minsta pos. heltalet som är udda är 1. Då blir n+1 = 2 och n-1 = 0. 2*0 = 0 och 0 % 8 = 0 → YES.
Nästa udda pos. heltal är 3. Då blir n+1 = 4 och n-1 = 2. 4*8 = 8 → YES.
Nästa udda pos. heltal är 5. Då blir n+1 = 6 och n-1 = 4. 6*4 = 24 → YES.
Efter det kommer alltid faktorn 2*2*2 = 8 vara med i (n+1)(n-1) när n är udda.

1409

Vilka värden kommer variablerna tal och mod att ha efter följande kod?

```
tal = 32
d = 6
mod = tal % d
tal = tal - d - d - d - d - d
```

Svara först utan Python. Testa sedan i Interactive mode. För vilken matematisk kunskap är koden ett exempel på? Svara först. Testa sedan i Python.

Utan Python:

mod = 32 % 6 = 2

tal = 32 – 6 – 6 – 6 – 6 – 6 = 2

Slutsats: $32 \% 6 = 32 - 5 \cdot 6$

Koden är ett exempel på:

a % b är upprepad subtraktion av *b* från *a* , så många gånger som b ryms i *a* .

Interactive mode

```
>>> tal = 32
>>> d = 6
>>> mod = tal % d
>>> tal = tal - d - d - d - d - d
>>>
>>> tal
2
>>> mod
2
```

1410

Idag är det onsdag. Julia vill träffa sin kompis om 13 dagar och vill veta vilken veckodag det blir. Lös problemet generellt:

Skriv ett program som frågar efter aktuell veckodag. Mata in en siffra för veckodagen. Numrera veckans dagar stigande från 1-7 med början på måndag. Sedan ska programmet fråga, när användaren vill träffa din kompis och få som svar ett antal dagar. Beräkna och skriv ut den planerade träffens veckodag som nummer.

```
# 1410.py

idag = int(input('\n\tVilken veckodag har vi idag?\n\n' +
                 '\tSvara med en siffra (mån = 1, ..., sön = 7):\t '))

antalDagar = int(input(
                 '\n\tOm hur många dagar vill du träffa din kompis?    '))

träff = (idag + antalDagar) % 7

print('\n\tDu kommer att träffa din kompis veckodag\t', träff, '\n')
```

1411

Koda följande algoritm:

1. Läs in tre heltal till timmar, minuter och sekunder.
2. Omvandla allt till totalsekunder. Skriv ut totalsek.
3. Ta sekunderna från steg 2.
4. Omvandla tillbaka till timmar, minuter och sekunder.
5. Skriv ut tim, min, sek.

Gör utskriften användarvänlig. Testa ditt program t.ex. för 7 timmar, 58 minuter och 34 sekunder.

```
# 1411.py

   tim = int(input('\n\t Ange antal timmar:\t\t'))
   min = int(input('\n\t Ange antal minuter:\t\t'))
   sek = int(input('\n\t Ange antal sekunder:\t\t'))

   totalsek = 3600 * tim + 60 * min + sek

   print('\n\t', tim, 'timmar,', min, 'minuter och', sek,
         'sekunder är', totalsek, 'sekunder.')

   tim =   totalsek // 3600
   min =  (totalsek % 3600) // 60
   sek = ((totalsek % 3600) % 60) % 60

   print('\n\t', totalsek, 'totalsekunder är', tim,
         'timmar,', min, 'minuter,', sek, 'sekunder.\n')
```

1412

Koda följande algoritm:

1. Läs in tre heltal till år, månader, veckor och dagar.
2. Omvandla allt till totaldagar och skriv ut resultatet.
3. Ta totaldagarna från steg 2.
4. Omvandla tillbaka till år, månader, veckor och resterande dagar.
5. Skriv ut år, månader, veckor samt resterande dagar.

Gör utskriften användarvänlig. Testa ditt program t.ex. för 2 år, 11 månader, 3 veckor och 6 dagar.

```
# 1412.py

år      = int(input('\n\t Ange antal år:\t\t'))
månader = int(input('\n\t Ange antal månader:\t'))
veckor  = int(input('\n\t Ange antal veckor:\t'))
dagar   = int(input('\n\t Ange antal dagar:\t'))

totaldagar = 365 * år + 30 * månader + 7 * veckor + dagar

print('\n\t', år, 'år,', månader, 'månader,', veckor, 'veckor och',
      dagar, 'dagar är', totaldagar, 'dagar totalt.')

år        =   totaldagar // 365
månader   =  (totaldagar % 365) // 30
veckor    = ((totaldagar % 365) % 30) // 7
restdagar = ((totaldagar % 365) % 30) % 7

print('\n\t', totaldagar, 'dagar år', år, 'år,', månader,
      'månader,', veckor, 'veckor och', restdagar, 'dagar.\n')
```

1413

Skriv ett program som läser in begynnelsebokstaven till en veckodag, med `if-elif-else`-satsen (sid 49) bestämmer vilken veckodag det är och skriver ut den.

Fixa problemet med tisdag/torsdag genom att bygga in en `if-else`-sats ifall det matas in `'t'`. Läs in och bearbeta den andra bokstaven. Ta även hand om felaktig inmatning både för den första och den andra begynnelsebokstaven.

```
# 1413.py

   bokstav1 = input('\n\tAnge en veckodags begynnelsebokstav:\t')

   if bokstav1 == 'm' :
      veckodag = 'måndag'
   elif bokstav1 == 't' :
        bokstav2 = input('\n\tAnge veckodagens andra bokstav:\t')
        if bokstav2 == 'i' :
           veckodag = 'tisdag'
        elif bokstav2 == 'o':
           veckodag = 'torsdag'
        else :
           veckodag = '?'
   elif bokstav1 == 'o' :
        veckodag = 'onsdag'
   elif bokstav1 == 'f' :
        veckodag = 'fredag'
   elif bokstav1 == 'l' :
        veckodag = 'lördag'


   elif bokstav1 == 's' :
        veckodag = 'söndag'
   else :
        veckodag = '?'
   if veckodag != '?' :
      print('\n\tDet är', veckodag, '.\n')
   else :
      print('\n\tDetta är ingen veckodag!\n')
```

1.5 Gissa tal – ett spel – övn sid 62

1501

a) Använd en loop med `while`-satsen för att skriva ut de första 10 positiva heltalen.
b) Vilken ändring i koden till a) måste göras för att få de första 20 pos. heltalen?

```
# 1501_a.py

tal = 0

while tal < 10 :
   tal = tal + 1
   print('\t', tal)
```

```
# 1501_b.py

tal = 0

while tal < 20 :
   tal = tal + 1
   print('\t', tal)
```

1502

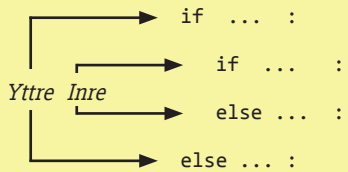
a) Skriv ett program som skriver ut de första 10 jämna talen.
b) Modifiera a) så att endast de första 10 udda talen skrivs ut.

```
# 1502_a.py

tal = 0

while tal < 20 :
   tal = tal + 1
   if tal % 2 == 0 :
      print('\t', tal)
```

```
# 1502_b.py

tal = 0

while tal < 20 :
   tal = tal + 1
   if tal % 2 == 1 :
      print('\t', tal)
```

1503

I programmet **GissaTal_1** (sid 54) används en `if-elif-else`-sats för att koda trevägsval. Ett sådant val kan även kodas med en nästlad `if-else`-sats som har följande struktur:



Modifiera programmet **GissaTal_1** så att `if-elif-else`-satsen ersätts av en nästlad `if-else`-sats.

```
# 1503.py

   secret = 17
   guess = int(input('\n\tGissa ett tal mellan 1 och 20:\t'))

   if guess <= secret :
      if guess == secret :
         print('\n\tGRATTIS, du har gissat rätt!\n')
      else :
         print('\n\tFel:', guess, '< hemliga talet.\n')
   else :
      print('\n\tFel:', guess, '> hemliga talet.\n')
```

1504

a) Skriv ett program som beräknar summan av de första 10 positiva heltalen.
b) Generalisera a) så att programmet beräknar summan av de första n positiva heltalen där n kan matas in. Testa för n = 100 och 1 000.
c) Skriv ett program som beräknar summan s av de första n positiva heltalen med hjälp av formeln $summa = n\,(n+1)\,/\,2$. Testa om du får samma svar i b) och c) för n = 1 000, 5 000 och 1 000 000.

```
   # 1504 a).py

   sum = 1
   term = 1
   while term < 10 :
      term = term + 1
      sum = sum + term
   print('\n\tSumman 1 + 2 + ... + 10 =', sum, '\n')
```

```
   # 1504 b).py

   n = int(input('\n\tHur långt vill du summera de positiva heltalen?\t'))

   sum = 1
   term = 1
   while term < n :
      term = term + 1
      sum = sum + term
   print('\n\tSumman 1 + 2 + ... +', n, '=', sum, '\n')
```

```
# 1504 c).py

  n = int(input('\n\tHur långt vill du summera de positiva heltalen?\t'))
  sum = n * (n + 1) // 2

  print('\n\tSumman 1 + 2 + ... +', n, '=', sum, '\n')

  print('\n\tSumman 1 + 2 + ... +', n, '=', sum, '\n')
```

1505

Koda följande algoritm till ett program:

Läs in ett positivt heltal
Skriv ut talet
Så länge talet ≠ 1 REPETERA:
    OM talet är udda
        multiplicera med 3, addera 1
    ANNARS
        dividera talet med 2
    Skriv ut talet

Testa programmet genom att läsa in heltalen 3, 6, 7, 13 och 50. Ange slutresultaten. Testa gärna fler startvärden. Studera de uppkomna talföljderna. Har du en förklaring för slutresultaten?

```
  # 1505.py

  tal = int(input('\n\t Mata in ett positivt heltal:\t'))
  print('\t', tal)
  while tal != 1 :
     if tal % 2 == 1 :
        tal = 3 * tal + 1
     else :
        tal = tal // 2
     print('\t', tal)
```

```
  Resultat:   Alla testade startvärden leder till 1 oavsett startvärde.
```

```
  Förklaring: Ingen. Det finns hittills inget matematiskt bevis för detta.
```

1506

Koda följande algoritm till ett program:
Testa för 48 och 60. Vad gör algoritmen?

Läs in två positiva heltal a och b
Så länge a ≠ b REPETERA:
    OM a > b
        a = a - b
    ANNARS
        b = b - a
Skriv ut a

```
# 1506.py

a = int(input('\n\t Mata in ett pos. heltal:\t'))
b = int(input('\n\t Mata in ett annat pos. heltal:\t'))

while a != b :
   if a > b :
      a = a - b                    # Upprepad subtraktion, se övn 1409
   else :
      b = b - a
print('\n\t', a, '\n')
```

Algoritmen beräknar den största gemensamma faktorn (gcd = greatest common divisor) av talen a och b (Euklides' algoritm).
Körresultatet för 48 och 60 blir 12. Dvs:
gcd(48, 60) = 12, vilket kan användas för att förkorta bråk: $\frac{48}{60} = \frac{48/12}{60/12} = \frac{4}{5}$

1.6 Hantering av slumptal – övn sid 69

1601

a) Skriv ett program som använder en loop med `for`-satsen för att skriva ut 10 slumptal mellan 0 och 1.
b) Skräddarsy Pythons funktion **`random()`** för att slumpa 20 heltal mellan 1 och 50. Bygg in det i ett program som skriver ut slumptalen.

```
# 1601_a.py

import random

for i in range(1, 11) :
    print('\t', random.random())
```

```
# 1601_b.py

import random

for i in range(1, 21) :
    print('\t',
    int(1 + random.random() * 50))
```

1602

Vi vill simulera tärningskast.

a) Generera 10 slumptal mellan 1 och 6 och skriv ut dem.
b) Skapa först en tom lista i Python, lägg de 10 slumptalen från a) i listan och skriv ut den.
c) Skriv ut en lista som innehåller 500 tärningskast.

```
# 1602_a.py

import random

for i in range(1, 11) :
    print('\t', int(1 + random.random() * 6))
```

```
# 1602_b.py

import random

tärningsLista = [ ]

for i in range(1, 11) :
    tärningsLista = tärningsLista + [int(1 + random.random() * 6)]

print('\n\t', tärningsLista, '\n')
```

```
# 1602_c.py

import random

tärningsLista = [ ]

for i in range(1, 501) :
    tärningsLista = tärningsLista + [int(1 + random.random() * 6)]

print('\n\t', tärningsLista, '\n')
```

1603

Skriv ett program som räknar förekomsten (frekvensen) av de 6 slumptalen i övn **1602** c):s lista över 500 tärningskast. Skapa 6 dellistor och lägg i var och en endast samma slumptal. Skriv ut dellistornas längder. Är de ungefär lika stora? Ange ett exempel. Avsluta programmet med en kontroll genom att summera alla dellistornas längder, vilket borde ge 500.

```
# 1603_c.py

import random

L = L1 = L2 = L3 = L4 = L5 = L6 = [ ]

for i in range(0, 500) :
    L = L + [int(1 + random.random() * 6)]
    if L[i] == 1 :
       L1 = L1 + [L[i]]
    if L[i] == 2 :
       L2 = L2 + [L[i]]
    if L[i] == 3 :
       L3 = L3 + [L[i]]
    if L[i] == 4 :
       L4 = L4 + [L[i]]
    if L[i] == 5 :
       L5 = L5 + [L[i]]
    if L[i] == 6 :
       L6 = L6 + [L[i]]

print('\n\t', L,
      '\n\n\tAntal 1-or =', len(L1),
      '\n\n\tAntal 2-or =', len(L2),
      '\n\n\tAntal 3-or =', len(L3),
      '\n\n\tAntal 4-or =', len(L4),
      '\n\n\tAntal 5-or =', len(L5),
      '\n\n\tAntal 6-or =', len(L6),
      '\n\n\tKontroll:\n\tSumman av alla dellistornas längder =',
      len(L1) + len(L2) + len(L3) + len(L4) + len(L5) + len(L6), '\n')
```

```
Dellistornas längder är ungefär lika stora, t.ex.: 81, 81, 81, 94, 88, 75
```

1604

Skriv ett program som skriver ut endast var 10:e tal i heltalsintervallet [1, 5 000]. Låt programmet läsa in steget 10 som en variabel. Om steget är *n* ska var *n*:e tal skrivas ut. Testa för olika *n*. Använd lista för att kunna se utskriften på skärmen.

```
# 1604.py

steg = int(input('\n\tMata in ett heltal för steget:\t'))

L = [ ]
for i in range(1, 5001) :
    if i % steg == 0 :
       L = L + [i]
print('\n', L, '\n')
```

1605

Den inbyggda pythonfunktionen **`ord()`** returnerar heltalskoden till en bokstav medan funktionen **`chr()`** returnerar bokstaven till en heltalskod. T.ex. är `ord('a')` = 97 och `chr(97)` = `'a'`. Använd dessa funktioner för att med en `for`-sats skriva ut det engelska alfabetets stora bokstäver A-Z. Mer info om **`ord()`** och **`chr()`** finns på sid 82.

```
# 1605.py

for i in range(ord('A'), ord('Z') + 1) :
    print('\t', chr(i))
```

1606

Familjen Pettersson tänker plundra sina tre spargrisar för att gå till Gröna Lund. De vill uppskatta hur mycket de kommer att få ihop. Troligen finns det mellan 90-120, 70-85 och 35-50 kr i varje spargris.

Den yngste sonen Max som läst *Koda matte med Python* vill simulera plundringen och tänker skriva ett program som använder sig av slumptal för de uppskattade grisvärdena för att beräkna ett närmevärde till den totala spargrisförmögenheten. Skriv programmet åt Max.

```
# 1606.py

import random

gris = [ ]                                                  # Tom lista
a1 = 90; b1 = 120
gris = gris + [a1 + int(random.random() * (b1-a1+1))]       # 1:a grisens
                                                            # pengar läggs
                                                            # till listan
a2 = 70; b2 = 85
gris = gris + [a2 + int(random.random() * (b2-a2+1))]       # 2:a grisens
                                                            # pengar läggs
                                                            # till listan
3 = 35; b3 = 50
gris = gris + [a3 + int(random.random() * (b3-a3+1))]       # 3:e grisens
                                                            # pengar läggs
                                                            # till listan
total = 0
for i in range(0, 3) :                                      # 3 grisar
    total = total + gris[i]                                 # Pengarna adde-
print('\n\tI spargrisarna finns ca.', total, 'kr.\n')       # ras, skrivs ut
```

1607

En borrutrustning för bergvärme kan borra 25 m i en viss tomtmark under den första timmen. Under de följande timmarna minskar borrdjupt med uppskattningsvis 10- 20% för varje timme.

Skriv ett program som anger ett närmevärde till det totala borrdjupet om borren går oavbrutet i 8 timmar. Programmet ska använda slumptal för den uppskattade minskningen av utrustningens prestation efter den första timmen.

```
# 1607.py

import random

totalDepth = 0
hDepth = 25                                         # 1:a timmens borrdjup
a = 10
b = 20
procent = a + int(random.random() * (b-a+1))        # Mellan 10 och 20 %
FF = 1 - procent / 100                              # Förändringsfaktorn

for h in range(1, 9) :                              # 8 timmar
    totalDepth = totalDepth + hDepth                # Varje timmes totaldjup
    hDepth = FF * hDepth                            # Varje timmes borrdjup
                                                    # efter 1:a timmen
print('\n\tHålet för bergvärmen är ca.', int(totalDepth), 'meter djupt.\n')
```

1.7 Funktioner i programmering – övn sid 76

1701

Funktionen $y = f(x) = x^2$ kan i Python definieras så här:

```
def f(x) :
    return x**2
```

a) Inkludera funktionen i ett program som anropar den för att skriva ut följande värdetabell för alla heltal x i intervallet [-5, 5]:
Markera med kommentar definitionen och anropen.

b) Utöka värdetabellen till x-intervallet [-10, 10].

x	y
-5	25
-4	16
-3	9
-2	4
-1	1
0	0
1	1
2	4
3	9
4	16
5	25

```
# 1701_a.py

def f(x) :              # Definition
    return x**2

print('\n\t', 'x\t', 'y', '\n')

for i in range(-5, 6) : # Anrop
    print('\t', i, '\t', f(i))
```

```
# 1701_b.py

def f(x) :
    return x**2

print('\n\t', 'x\t', 'y', '\n')

for i in range(-10, 11) :
    print('\t', i, '\t', f(i))
```

1702

Definiera funktionen $y = f(x) = x^3$ i Python.

a) Inkludera funktionen i ett program som anropar den för att skriva ut en värdetabell för alla heltal x i intervallet [-5, 5] på samma sätt som i övn 1701 a).

b) Skriv värdena från a) i listor: Skapa en lista för alla x och en för alla y. Lägg in i dem resp. värdena och skriv ut listorna. Markera med kommentar funktionens definition och anrop.

```
# 1702_a.py

def f(x) :
    return x**3

print('\n\t', 'x\t', 'y', '\n')

for i in range(-5, 6) :
    print('\t', i, '\t', f(i))
```

```
# 1702_b.py

def f(x) :              # Definition
    return x**3

x = y = [ ]             # Listorna

for i in range(-5, 6) :
    x = x + [i]
    y = y + [f(i)]  # Anrop

print('\n\tx ='  , x,
      '\n\n\ty =', y, '\n')
```

1703

Skriv de fyra räknesätten och heltalsdivisionen samt modulo som funktioner i Python. Läs in två heltal. Anropa funktionerna och skriv ut resultaten så att du får t.ex. följan-de utskrift när du läser in 5 och 3:

```
5  +  3  ger  8
5  -  3  ger  2
5  *  3  ger  15
5  /  3  ger  1.6666666666666667
5  // 3  ger  1
```

Ditt program ska bli en modularisering av övn **1305**.

```
# 1703.py

tal1 = int(input('\n\t Mata in ett heltal:       '))
tal2 = int(input('\n\t Mata in ett heltal till:  '))

def sum(a, b) :
    return a + b

def diff(a, b) :
    return a - b

def prod(a, b) :
    return a * b

def div(a, b) :
    return a / b

def intDiv(a, b) :
    return a // b

def mod(a, b) :
    return a % b


print('\n\t', tal1, ' + ' , tal2, ' ger ', sum(tal1, tal2)   , '\n\t',
              tal1, ' - ' , tal2, ' ger ', diff(tal1, tal2)  , '\n\t',
              tal1, ' * ' , tal2, ' ger ', prod(tal1, tal2)  , '\n\t',
              tal1, ' / ' , tal2, ' ger ', div(tal1, tal2)   , '\n\t',
              tal1, ' //' , tal2, ' ger ', intDiv(tal1, tal2), '\n\t',
              tal1, ' % ' , tal2, ' ger ', mod(tal1, tal2)   , '\n  ')
```

1704

Modularisera lösningen till övn **1411** (sid 127) genom att skriva beräkningen av totalsek som en funktion. Välj olika variabler för de formella och de aktuella parametrarna. Anropa funktionen. Testa ditt program t.ex. för 7 timmar, 58 minuter och 34 sekunder. Får du samma resultat som i övn **1411**?

```
# 1704.py

def totalsek(t, m, s) :
    return 3600 * t + 60 * m + s

tim = int(input('\n\t Ange antal timmar:\t\t'))
min = int(input('\n\t Ange antal minuter:\t\t'))
sek = int(input('\n\t Ange antal sekunder:\t\t'))
t   = totalsek(tim, min, sek)

print('\n\t', tim, 'timmar,', min, 'minuter och', sek,
                        'sekunder är', t, 'sekunder.')

tim =   t // 3600
min =  (t % 3600) // 60
sek = ((t % 3600) % 60) % 60

print('\n\t', t, 'totalsekunder är', tim, 'timmar,',
                  min, 'minuter,', sek, 'sekunder.\n')
```

1705

Modularisera programmet **GissaTal_3** (sid 58) genom att generera slumptalet **secret** (rad **8**) med funktionen **myRand()** (sid 72). Bibehåll **myRand()** i en separat fil och importera den tillhörande modulen i ditt program.

```
# RandFkt.py

import random

def myRand(a, b) :
    if a < b :
       return a + int(random.random() * (b-a+1))
    else :
       return b + int(random.random() * (a-b+1))
```

```
# 1705.py

import RandFkt                              # Importerar modulen RandFkt

secret = RandFkt.myRand(1, 100)             # Anropar funktionen myRand()
                                            # i modulen RandFkt
print('\n\tProgrammets hemliga tal är mellan 1 och 100.')

guessNo = 0
wrongGuess = True

while wrongGuess :
   guess = int(input('\n\tGissa vilket heltal (Avsl. med 0):  '))
   if guess == 0 :
      print('\n\tAvbrott: Programmets hemliga tal var\t',secret, '\n')
      wrongGuess = False
   elif guess == secret :
      print('\n\t\tGRATTIS, du har gissat rätt efter',
                                                 guessNo, 'försök.\n')
      wrongGuess = False
   elif guess < secret :
      print('\n\t\t', guess, '< hemliga talet.\tFörsök igen!')
   else :
      print('\n\t\t', guess, '> hemliga talet.\tFörsök igen!')
   guessNo = guessNo + 1
```

1.8 Kryptering – övn sid 85

1801

Experimentera med programmet **Char2int** (sid 83) för att ta reda på differensen mellan gemeners och versalers ASCII-koder. Skriv ett program som läser in en gemen och skriver ut dess versal och sedan läser in en versal och skriver ut dess gemen. Använd Pythons funktioner **ord()** och **chr()** (sid 82).

```
# 1801.py

codeDiff = ord('a') - ord('A')

gemen = input('\n\t Mata in en gemen:\t ');
print('\n\t', gemen, ':s versal är\t\t', chr(ord(gemen) - codeDiff))

versal = input('\n\t Mata in en versal:\t ');
print('\n\t', versal, ':s gemen är:\t\t', chr(ord(versal) + codeDiff),
                                                        '\n')
```

1802

Skriv ett program som läser in ett tecken och förskjuter det i teckentabellen med ett visst antal steg som en slags krypteringsnyckel. Skriv ut både det inlästa och det förskjutna tecknet på ett användarvänligt sätt.

```
# 1802.py

step = int(input('\n\tMata in ett heltal (krypteringsnyckel):\t'))
letter = input('\n\tMata in ett tecken: \t\t\t')

output = '\n\t' + letter                    # Inläst originaltecken
letter = chr(ord(letter) + step)            # Krypterat tecken

print(output, 'har förskjutits med', step, 'steg och är nu:', letter,
                                                        '\n')
```

1803

Skriv ett program som läser in fem tecken och skriver ut dem förskjutna med steget 1 i teckentabellen så att t.ex. inmatningen **Kalle** ger utskriften **Lbmmf**. Återställ sedan det krypterade ordet. Vidareutveckla programmet genom att öka steget och läsa in krypteringsnyckeln.

```
# 1803.py
# Tecknet \ följt av Enter är radfortsättningstecknet i pythonkod
# Tecknet \ används i rader som inleds med newText och restoredText

# key = 1
key = int(input('\n\tMata in ett heltal (krypteringsnyckel):  '))

oldText = input('\n\tMata in 5 tecken utan mellanslag:\t ')
ch1 = oldText[0]
ch2 = oldText[1]
ch3 = oldText[2]
ch4 = oldText[3]
ch5 = oldText[4]

newText = chr(ord(ch1) + key) + chr(ord(ch2) + key) + chr(ord(ch3) + \

          key) + chr(ord(ch4) + key) + chr(ord(ch5) + key)

print('\n\tDet krypterade ordet:\t\t\t', newText)

ch1 = newText[0]
ch2 = newText[1]
ch3 = newText[2]
ch4 = newText[3]
ch5 = newText[4]

restoredText = chr(ord(ch1) - key) + chr(ord(ch2) - key) + \
               chr(ord(ch3) - key) + chr(ord(ch4) - key) + \
               chr(ord(ch5) - key)

print('\n\tDet återställda ordet:\t\t\t', restoredText, '\n')
```

1804

Vidareutveckla lösningen till övn **1803** genom att utöka och läsa in antalet tecken. Mata in text av godtycklig längd, kryptera den och återställ sedan utgående från den krypterade texten.

```
# 1804.py

key = int(input('\n\tMata in ett heltal (krypteringsnyckel):  '))

oldText = input('\n\tMata in text:\t\t\t ')
newText = ''                          # Tom sträng

for n in range(0, len(oldText)) :
    ch = oldText[n]                   # Tar tecknen från oldText
    ch = chr(ord(ch) + key)           # Krypterar tecknen
    newText = newText + ch            # Lägger tecknen i newText
```

```
    print('\n\tDen krypterade texten:\t\t\t', newText)

    restoredText = ''

    for n in range(0, len(newText)) :
        ch = newText[n]                  # Tar tecknen från newText
        ch = chr(ord(ch) - key)          # Återställer tecknen
        restoredText = restoredText + ch # Lägger tecknen i restoredText

    print('\n\tDen återställda texten:\t\t\t', restoredText, '\n')
```

1805

Modularisera lösningen till övn **1804** genom att skriva koden för både kryptering och återställning som en funktion kallad **`myEncrypt()`**. Anropa funktionen med den inlästa nyckeln för att kryptera texten och med den negativa nyckeln för att återställa den.

```
# 1805.py

def myEncrypt(oldText, k) :
    newText = ''                         # Tom sträng
    for n in range(0, len(oldText)) :
        ch = oldText[n]                  # Tar tecknen från oldText
        ch = chr(ord(ch) + k)            # Krypterar tecknen
        newText = newText + ch           # Lägger tecknen i newText
        return newText                   # Reurnerar krypterad text

key = int(input('\n\tMata in ett heltal (krypteringsnyckel):  '))
text = input('\n\tMata in text:\t\t\t\t ')

EncryptedText = myEncrypt(text, key)   # Krypterar text
print('\n\tDen krypterade texten:\t\t\t', EncryptedText)

restoredText = myEncrypt(EncryptedText, -key) # Återställer text
print('\n\tDen återställda texten:\t\t\t', restoredText, '\n')
```

1806

Modifiera lösningen till övn **1805** genom att generera krypteringsnyckeln med slumptal. Använd funktionen **`myRand()`** (sid 72) för detta och välj själv intervallet för val av slumptal. Låt **`myRand()`** ligga i en separat fil och importera den som en modul i ditt program.

```
# RandFkt.py

import random

def myRand(a, b) :
    if a < b :
       return a + int(random.random() * (b-a+1))
    else :
       return b + int(random.random() * (a-b+1))
```

```
# 1806.py

 import RandFkt                              # Importerar modulen RandFkt

 def myEncrypt(oldText, k) :
     newText = ''                            # Tom sträng
     for n in range(0, len(oldText)) :
         ch = oldText[n]                     # Tar tecknen från oldText
         ch = chr(ord(ch) + k)               # Krypterar tecknen
         newText = newText + ch              # Lägger tecknen i newText
     return newText                          # Reurnerar krypterad text

 key = RandFkt.myRand(1, 100)                # Anropar funktionen myRand()
 text = input('\n\tMata in text:\t\t ')

 EncryptedText = myEncrypt(text, key)     # Krypterar text
 print('\n\tDen krypterade texten:\t', EncryptedText)

 restoredText = myEncrypt(EncryptedText, -key)  # Återställer text
 print('\n\tDen återställda texten:\t', restoredText, '\n')
```

1807

Skriv ut med en loop en teckentabell i ASCII-intervallet [33, 256]. Utskriften ska visa varje tecken bredvid sin ASCII-kod i en tabell med 8 kolumner, dvs gör radbyte var 8:e utskrift. För att undvika radbyten med **`print()`**-satser i loopen gör så här: Skapa en tom sträng före loopen, lägg i den alla utskrifter i loopen och skriv ut den efter loopen med en enda **`print()`**-sats.

```
    # 1807.py

    output = ''                                # Tom utskriftssträng

    for code in range (33, 257) :
        output = output + str(code) + ' ' + chr(code) + '\t'
        if code % 8 == 0 :                     # Var 8:e utskrift:
           output = output + '\n'              # Radbyte
        code = code + 1

    print(output)
```

1808

Skriv ett program som skapar slumplösenord bestående av 8 tecken med följande policy: 3 små bokstäver, 2 siffror och 3 stora bokstäver. Experimentera med ASCII-tabellen från övn **1807** för att få reda på kodintervallerna till lösenordens olika delar. Programmet ska fråga efter önskat antal lösenord och skriva ut dem i två kolumner: I den första ska stå: **user1**, **user2**, ... I den andra ska till varje användare stå ett slumpvis genererat lösenord.

```
# RandFkt.py

import random

def myRand(a, b) :
    if a < b :
       return a + int(random.random() * (b-a+1))
    else :
       return b + int(random.random() * (a-b+1))
```

```
# 1808.py
# Genererar slumplösenord med policyn:
# 8 tecken = 3 små bokstäver: ASCII-intervall (97, 122) +
#            2 siffror                       (48,  57) +
#            3 stora bokstäver               (65,  90)

import RandFkt

antal = int(input('\nHur många användarnamn med lösenord' +
                                          ' vill du ha?  '))
print()

for n in range(1, antal+1) :
    passwd = ''

    for i in range(0, 3) :                # 3 små bokstäver
        passwd = passwd + chr(RandFkt.myRand(97, 122))
    for i in range(3, 5) :                # 2 siffror
        passwd = passwd + chr(RandFkt.myRand(48, 57))

    for i in range(5, 8) :                # 3 stora bokstäver
        passwd = passwd + chr(RandFkt.myRand(65, 90))

    print('\tuser' + str(n), '\t' + passwd)

print()
```

1.9 Primtal – övn sid 97

1901

Matematikern *Fermat* trodde på 1600-talet att talet $2^{32} + 1 = 4\ 294\ 967\ 297$ var ett primtal. På 1700-talet fann *Euler* att det var sammansatt.

Kör programmet **PrimtalsTest** (sid 87) för att visa att talet är sammansatt. Ange talets primfaktorer genom att köra programmet **PrimFaktorer** (sid 95).

```
PrimtalsTest:

                Mata in ett positivt heltal:    4294967297

                Det inmatade talet 4294967297 är sammansatt.
```

```
PrimFaktorer:

                 Mata in ett positivt heltal:   4294967297

                 Talets primfaktorer:    [641, 6700417]
```

1902

Skriv ett program som kodar följande algoritm – en metod för att avgöra om ett tal är primtal *(Eratosthenes såll)*:

1. Läs in ett positivt heltal mellan 1 och 100.
2. Om talet är 1, är det varken ett primtal eller ettsammansatt tal.
3. Om talet är 2, 3, 5 eller 7 är det ett primtal, annars:
4. Om talet är delbart med 2, är det ett sammansatt tal, annars:
5. Om talet är delbart med 3, är det ett sammansatt tal, annars:
6. Om talet är delbart med 5, är det ett sammansatt tal, annars:
7. Om talet är delbart med 7, är det ett sammansatt tal, annars:
8. Är det ett primtal.

Testa programmet för olika startvärden. Kommer programmet att ge korrekt svar om man matar in startvärden som är > 100? Testa för 121 och 169. Är programmets svar korrekt? Avgör

med programmet **PrimtalsTest** (sid 87) om 121 och 169 är primtal eller ej. Förklara.

```
# 1902.py

   tal = int(input('\n\t Mata in ett tal mellan 1 och 100:\t'))

   if tal == 1 :
      print('\n\t', tal, 'är varken primtal eller sammansatt tal.\n')
   elif tal == 2 or tal == 3 or tal == 5 or tal == 7 :
      print('\n\t', tal, 'är ett primtal.\n')
   elif tal % 2 == 0 :
      print('\n\t', tal, 'är sammansatt.\n')
   elif tal % 3 == 0 :
      print('\n\t', tal, 'är sammansatt.\n')
   elif tal % 5 == 0 :
      print('\n\t', tal, 'är sammansatt.\n')
   elif tal % 7 == 0 :
      print('\n\t', tal, 'är sammansatt.\n')
   else :
      print('\n\t', tal, 'är ett primtal.\n')
```

För startvärden > 100 ger programmet ovan inte alltid korrekt svar.

T.ex. svarar programmet att 121 och 169 är primtal vilket är fel:

$$121 = 11 \cdot 11 \qquad \text{och} \qquad 169 = 13 \cdot 13$$

Men programmet kan inte upptäcka det därför att det endast testar delbarhet med upp till 7.

1903

Talet 1 uppfyller definitionen för primtal. Ändå sägs det i algoritmen i övn **1902**, punkt 2, att 1 varken är ett primtal eller ett sammansatt tal. Varför? Försök att hitta en förklaring.

Förklaring:

1 är endast delbart med 1 och med sig själv. Så primtalskraven är uppfyllda. Ändå får 1 inte vara med bland primtalen eftersom Aritmetikens fundamentalsats (sid 86) inte vore sann om 1 vore ett primtal. Då skulle t.ex. gälla:

$$12 = 2 \cdot 2 \cdot 3 = 1 \cdot 2 \cdot 2 \cdot 3 = 1 \cdot 1 \cdot 2 \cdot 2 \cdot 3 = \cdots$$

Dvs uppdelningen av ett tal i en produkt av primtal vore inte längre entydig.

1904

Vidareutveckla programmet i övn **1902** så att man med säkerhet kan avgöra om tal mellan 1 och 200 är primtal eller ej. Testa igen för 121 och 169. Hur går det för tal > 200? Testa för 529. Avgör med **PrimtalsTest** (sid 87) om 529 är primtal. Motivera din algoritm i det nya programmet.

```
# 1904.py

tal = int(input('\n\t Mata in ett tal mellan 1 och 200:\t'))

if tal == 1 :
   print('\n\t', tal, 'är varken primtal eller sammansatt tal.\n')
elif tal == 2 or tal == 3 or tal == 5 or tal == 7 or tal == 11 \
                                                  or tal == 13 :
print('\n\t', tal,'är ett primtal.\n')
elif tal % 2 == 0 :
   print('\n\t', tal, 'är sammansatt.\n')
elif tal % 3 == 0 :
   print('\n\t', tal, 'är sammansatt.\n')
elif tal % 5 == 0 :
   print('\n\t', tal, 'är sammansatt.\n')
elif tal % 7 == 0 :
   print('\n\t', tal, 'är sammansatt.\n')
elif tal % 11 == 0 :
   print('\n\t', tal, 'är sammansatt.\n')
elif tal % 13 == 0 :
   print('\n\t', tal, 'är sammansatt.\n')
else :
   print('\n\t', tal, 'är ett primtal.\n')
```

Nu svarar programmet korrekt, nämligen att 121 och 169 är sammansatta tal. För startvärden > 200 ger programmet 1904.py inte alltid korrekt svar. T.ex. svarar programmet att 529 är primtal vilket är fel:

$$529 = 23 \times 23$$

Men programmet kan inte upptäcka det därför att det endast testar delbarhet med upp till 13.

Om man med säkerhet vill avgöra om tal mellan 1 och 200 är primtal eller ej räcker det att testa delbarhet med upp till heltal

$$< \sqrt{200} = 14, \cdots,$$ dvs 13.

1905

Programmet **AllaPrimtal** skriver ut alla primtal i ett givet positivt heltalsintervall (sid 91). Modifiera det så att det nya programmet frågar efter antalet n primtal och listar ut de första n primtalen i en lista som börjar med 2. Importera modulen **PrimFkt** och anropa funktionen **primtest()** för att avgöra om ett tal är primtal (sid 90).

```
# 1905.py

import PrimFkt                    # Modulen PrimFkt som innehåller
                                  # funktioen primtest()
antal = int(input('\n\tHur många primtal vill du ha?  '))
print('\n\tHär listas de första', antal, 'primtalen:')

primtal = [ ]
tal = 2
while len(primtal) < antal :
   if PrimFkt.primtest(tal) :     # Anrop av funktioen primtest()
      primtal = primtal + [tal]
   tal = tal + 1
print('\n', primtal, '\n')
```

1906

Antalet primtal $\leq n$ brukar man beteckna med $\pi(n)$. T.ex. är $\pi(10) = 4$ eftersom 2, 3, 5 och 7 är de enda primtalen ≤ 10 .

Skriv en funktion antalPrim(n) som beräknar $\pi(n)$. Anropa funktionen i ett program för att skriva ut antalet primtal $\leq n$ för: $n = 10, 10^2, 10^3, 10^4$

```
# 1906.py

import PrimFkt

def antalPrim(n) :                          # Funktionen antalPrim()
    primtal = [ ]
    for tal in range(1, n) :
        if PrimFkt.primtest(tal) and tal != 1 :
           primtal = primtal + [tal]
    return len(primtal)


print()
for i in range (1, 5) :                     # Anrop av funktionen:
    print('\tAntalet primtal <=', 10**i, ' är ', antalPrim(10**i))
print()
```

1.10 Rekursion – övn sid 102

1011

Följande algoritm beräknar summan av de första n positiva heltalen som en rekursiv funktion:

```
Funktion sum(n)
    OM n = 1
        returnera 1
    ANNARS
        returnera n + sum(n-1)
```

a) Varför är funktionen rekursiv?
b) I vilken ordning adderar algoritmen de pos. heltalen, fram- eller baklänges? Förklara varför den gör så.
c) Skriv algoritmen som en funktion i Python. Bygg in funktionen i ett program och anropa den för n = 10 och för n = 100.

a) Funktionen är rekursiv därför att den anropar sig själv med sum(n-1) i sista raden.

b) Algoritmen summerar de positiva heltalen baklänges, t.ex. om n = 10:

10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1

Det framgår av uttrycket n + sum(n-1) som returneras i sista raden. Algoritmen gör så för att få stopp på rekursionen när n har nått 1.

```
# 1011_c.py

def sum(n) :
    if n == 1 :
       return 1
    else :
       return n + sum(n-1)

for i in range(1, 3) :
    print('\n\tSumman 1 + 2 + ... +', 10**i, '=', sum(10**i))

print()
```

1012

Följande algoritm beräknar siffersumman för ett positivt heltal:

```
Funktion siffSum(n)
   OM n < 10
      returnera n
   ANNARS
      rest = n % 10
      m = (n – rest) // 10
      returnera rest + siffSum(m)
```

a) Gå igenom algoritmen steg för steg för $n = 385$ och förklara vad den gör.
b) Skriv ett program som definierar funktionen siffSum() och anropar den för n = 385 och andra heltal som läses in.
c) Kontrollera ditt programs resultat genom att bilda n % 9 vilket ger n:s siffersumma resp. siffersummans siffersumma, jfr. övn **1407**.

a) siffSum(385): 385 > 10 → rest = 385 % 10 = 5 → m = (385 – 5) // 10 = 38

siffSum(385) = 5 + siffSum(38)

siffSum(38): 38 > 10 → rest = 38 % 10 = 8 → m = (38 – 8) // 10 = 3

siffSum(38) = 8 + siffSum(3)

siffSum(3): 3 < 10 → siffSum(3) = 3

siffSum(385) = 5 + siffSum(38) = 5 + 8 + siffSum(3) = 5 + 8 + 3 = 16

```
# 1012_b.py

tal = int(input('\n\tFör vilket pos. heltal vill du ha siffersumman?\t'))

def siffSum(n) :
    if n < 10 :
       return n
    else :
       rest = n % 10
       m = (n - rest) // 10
       return rest + siffSum(m)

  print('\n\tSiffersumman till', tal, 'är', siffSum(tal), '.\n')
```

```
c)    >>> n = 385
      >>> n % 9
      7
      >>> # Siffersumman:
      >>> 3 + 8 + 5
      16
      >>> # Siffersummans siffersumma:
      >>> 1 + 6
      7
```

1013

Euklides algoritm beräknar den största gemensamma faktorn (**gcd** = greatest common divisor) av de positiva heltalen a och b. Här är en rekursiv beskrivning av den:

```
Funktion gcd(a, b)
   OM b = 0
      returnera a
   ANNARS
      rest = a % b
      returnera gcd(b, rest)
```

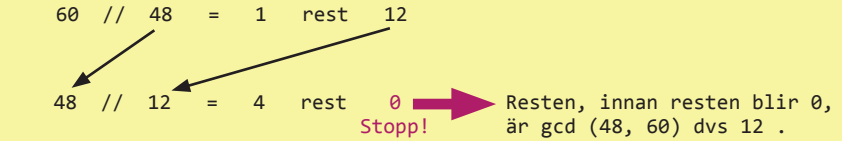
a) Ställ upp ett räkneschema för manuellt genomförande av algoritmen och demonstrera den för a = 60 och b = 48.

b) Skriv en funktion **gcd()** i Python och anropa den för a = 60 och b = 48 från ett program som kan läsa in även andra heltal.

c) Jämför ditt programs resultat med körresultat från övn **1506**:s lösning som kodar Euklides algoritm på ett iterativt sätt. Förklara varför man får identiska resultat fast **1506** använder upprepad subtraktion medan funktionen **gcd()** räknar med modulooperatorn?

d) Skriv en annan version av funktionen **gcd()** som använder upprepad subtraktion istället för modulo.

a)



```
# 1013_b.py

def gcd(a, b) :
    if b == 0 :
       return a
    else :
       rest = a % b
       return gcd(b, rest)

a = int(input('\n\tMata in ett positivt heltal:\t   '))
b = int(input('\n\tMata in ett annat pos. heltal:     '))

print('\n\tDen största gemensamma faktorn av', a, 'och', b,'är',
                                                 gcd(a, b), '.\n')
```

c) Alla körningar av programmen 1013_b.py och 1506.py ger identiska resultat. De gör båda samma sak dvs beräknar gcd(a, b). Detta beror på att modulooperationen är upprepad subtraktion (lösning till övn **1409**), närmare bestämt: a % b är upprepad subtraktion av b från a, så många gånger som b ryms i a.

```
# 1013_d.py

def gcd(a, b) :
    if a == b :
       return a
    else :
       if a > b :
          a = a - b
          return gcd(b, a)
       else :
          b = b - a
          return gcd(a, b)

a = int(input('\n\tMata in ett positivt heltal:\t   '))
b = int(input('\n\tMata in ett annat pos. heltal:     '))

print('\n\tDen största gemensamma faktorn av', a, 'och', b,'är',
                                                 gcd(a, b), '.\n')
```

Appendix

En mobil pythonmiljö

En pythonmiljö är en yta där man kan skriva pythonkod och utföra den direkt. Bakom ytan finns en programvara, *Python interpretatorn*, som tolkar koden och exekverar den. I vår app *Mattekollen* finns redan en sådan miljö inbyggd så att man slipper installera Pythons programmeringsmiljö i datorn (sid 14). Appen har en matte- och en programmeringsdel och följer Skolverkets kursplaner. Varje avsnitt har quiz, övningar och genomgångar, vars innehåll och omfång ständigt uppdateras. Det räcker med att ladda ner appen eller köra den som webbapp i webbläsaren på för att testa pythonkod.

Appen Mattekollen

1. Gå till **www.mattekollen.se** och välj:

   Kör som Webbapp | GET IT ON Google play | Download on the App Store

2. Öppna appen antingen som webbapp (på datorn eller mobilen) eller ladda ner den till din mobil. Du får bilden som visas på förra sidan. Under rubriken Programmering i matematik klicka på:

   En mobil pythonmiljö

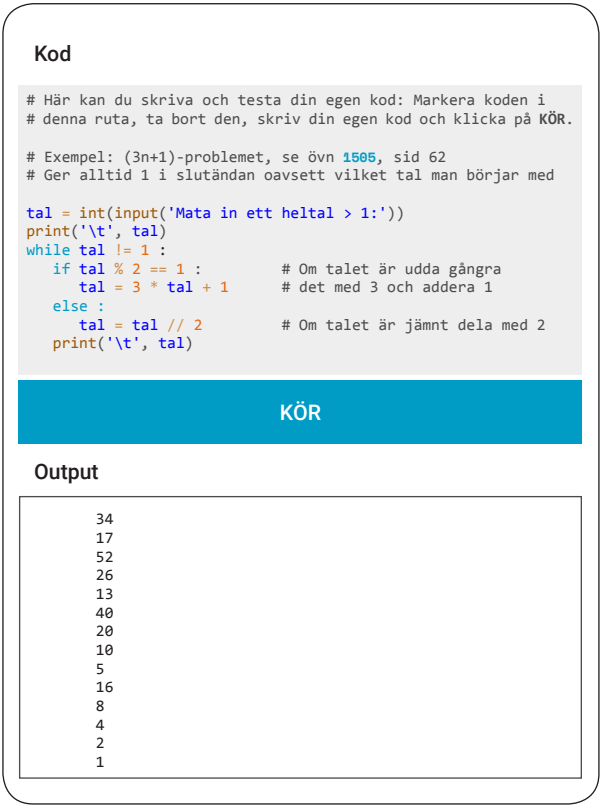
   Mattekollens pythonmiljö öppnas.

Appen Mattekollen är fortfarande en betaversion. Den vidareutvecklas permanent med avseende på både matematik- och programmeringsdelen. Du kan alltid få tag i den senast uppdaterade versionen genom att använda den som *webbapp* i webbläsaren via länken:

app.mattekollen.se

Mattekollens pythonmiljö

Appen Mattekollen innehåller en enkel programmeringsmiljö för Python, för att snabbt kunna testa vilken pythonkod som helst, men också lösa appens övningar. Detta ger användaren möjligheten att utan installation av programvara i datorn kunna mata in kod i mobilen och få svar direkt. Efter att du klickat på **En mobil pythonmiljö,** gör så här:



1. I det övre fönstret med rubriken **Kod** kan du mata i din kod som även kan bestå av flera rader. Du kan bryta rad genom att trycka på Enter.
   Enter-tryckningen exekverar inte raden. Vissa ord i koden får automatiskt en färg, för att underlätta kodningen: Miljön har stöd för s.k. *syntax highlighting* liknande andra utvecklingsmiljöer.
2. När du är klar med koden och vill exekvera den tryck på den blå knappen **KÖR**.
3. Du kan se kodens körresultat i det nedre fönstret med rubriken **Output**.

Exemplet på bilden ovan är det s.k. *(3n+1)-problemet*, en iterativ algoritm som alltid slutar med 1 oavsett startvärde – ett av matematikens hittills obevisade problem. Python ger oss möjligheten att på ett empiriskt sätt testa algoritmen med olika startvärden. Så får man en uppfattning om att den verkligen ger samma resultat, nämligen 1. Återstår att fundera över *varför* den gör så. Problemet finns även som en övning i boken (övn **1505**, sid 62).

Visual Studio

Sedan Pythons tillkomst har många utvecklingsmiljöer skapats för Python. En av de mest effektiva, kraftfulla och användarvänliga är *Visual Studio* som beskrivs här i detalj. Varianten vi presenterar här är gratis (freeware) och fungerar under Windows.

Visual Studio skapades år 1997 av *Microsoft* som, för att underlätta utvecklingsarbetet, integrerade verktyg för olika språk i EN miljö. Sådana program heter *Integrated Development Environments (IDEs)*. En IDE är ett grafiskt gränssnitt som bl.a. innehåller en editor, kompilatorer, interpretatorer, debuggers för olika språk och andra verktyg för programutveckling i en samlad miljö. En IDE har många fördelar: En av dem är att man slipper byta miljö mellan editering och körning, en annan är att man inte behöver bry sig om sökvägar i datorns filsystem, en tredje att man har stödverktyg för språket osv. Visual Studio är skapat för professionella utvecklare och därför väldigt stort och komplext. För att koncentrera oss på själva *språket* Python kommer vi att begränsa beskrivningen av miljön till ett minimum som är absolut nödvändigt.

En annan fördel av Visual Studio är att det är en professionell IDE för all sorts programutveckling som har kompilatorer för många språk (C#, Visual Basic, C++, F#, ...). Har man lärt sig miljön med ett språk blir det lätt att byta språk.

Installation av Visual Studio

1. Gå i din webbläsare till adressen:

   www.visualstudio.com/vs/python

   Visual Studio:s Python-sida visas.
   Gå med musen över den ljusblå knappen

   Dowload Visual Studio

   En dropplista dyker upp. Välj Community 2017.

2. Installationsfilen vs_community_….exe laddas ner. Dubbelklicka på den just hämtade installationsfilen. Svara Ja på frågan om du ska tillåta att den

här appen får göra ändringar på din dator. Klicka på Continue om det dyker upp rutan Visual Studio Installer.

3. Det tar ett tag tills Visual Studio Installer öppnar ett stort vitt fönster dyker upp med den lilla rubriken Installing – Visual Studio … och den blåmarkerade fliken **Workloads**. I den finns ett antal rutor. Bland dem finns följande:

4. Markera bland rutorna i fliken Workloads den 4:e rutan till vänster med rubriken:

Python development
genom att bocka den lilla blå rutan i det övre högra hörnet. Klicka sedan i det stora vita fönstrets nedre högra hörn på knappen **Install**. Detta kan ta ett tag, ev. ganska länge – beroende på din dators prestation.

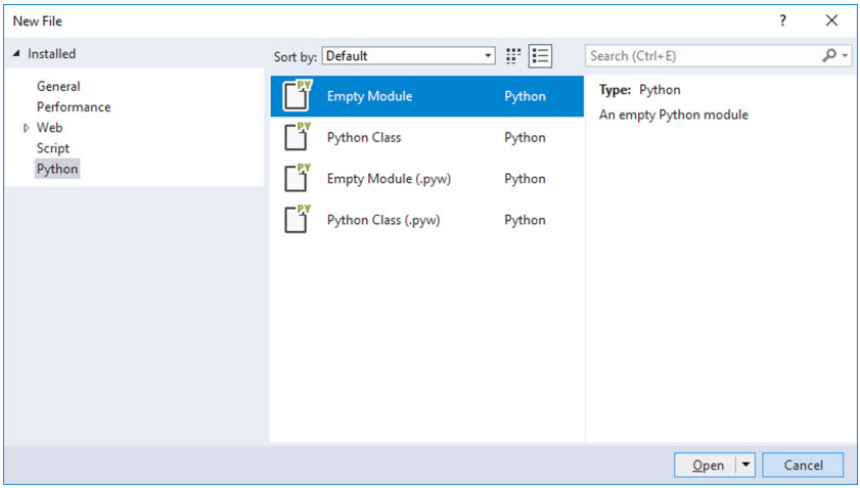
När du får upp ett fönster med bl.a. detta → har du lyckats med att installera programvaran. Klicka på knappen Launch. Om du uppmanas att skapa ett konto (Sign in) klicka antingen på Not now, may be later eller skaffa ett Microsoft-konto. Det är gratis, går fort och är inte problematiskt. Om du får upp en ruta med bl.a. dropplistan Development Settings välj Python. Om alternativet inte finns låt General stå där. Klicka sedan på knappen Start Visual Studio.

Python i Visual Studio

Öppna Visual Studio och klicka i menyraden på:

File → New → File…

Följande dialogrutan New File kommer upp:



Expandera i den vänstra kolumnen Installed om det inte redan är gjort och markera Python. Markera i den mellersta kolumnen knappen Empty Module . Klicka på knappen Open längst ner till höger. Du återvänder till Visual Studio. Om de små fönstren Solution Explorer, Properties eller andra dyker upp till höger, stäng dem. Klicka i menyraden längst upp på:

File → Save Selected Items As…

Dialogrutan Save File As dyker upp. Navigera i din dators fil- och mappsystem till den mapp som du vill lägga dina pythonfiler i. Döp filen till **Welcome.py** i den undre delen av dialogrutan, i textrutan Filnamn: . Klicka på Save. Du återvänder till Visual Studio.

Skriv i det vita fönstret följande pythonkod (utan radnumren):

```
1    # Welcome.py
2    # Skriver ut text
3
4    print('\n\t Välkommen till \n'                    '
5          '\n\t Koda matte med Python! \n'           '
6          '\n\t Programmering i matematik \n'        '
7          '\n\t En handbok för lärare och elever \n')
```

För att exekvera programmet `Welcome` klicka i Visual Studios menyrad längst upp på:

Project → Start Without Debugging

Om allt gått bra bör du se nu följande utskrift:

```
Välkommen till

Koda matte med Python!

Programmering i matematik

En handbok för lärare och elever
```

Är din utskriftskonsol fortfarande svart och du vill kanske byta bakgrundsfärg på den, gå till den lilla svarta ikonen i konsolfönstrets övre vänstra hörn för att välja Egenskaper. Följ instruktionerna på sid 15 under rubriken Bakgrundsfärg.

KODA MATTE MED PYTHON

En lärobok som kompletterar den klassiska matematikundervisningen med inslag av programmering. Den vägleder, stöttar och utmanar både lärare och elever genom att kombinera teori med praktiska övningar och fullständiga lösningar.

I framtiden kommer programmeringens relevans knappast avta. Enligt Skolverkets nya riktlinjer ska därför alla elever som läser matematik i Sverige förstå och kunna använda programmeringens fundament. Avsikten med denna bok är att erbjuda ett hjälpmedel som garanterar en smidig övergång till en matematikundervisning som genomsyras av programmeringens tankesätt.

Oavsett om du är lärare eller elev, nybörjare på högstadiet eller semiproffsig kodare på gymnasiet, så lämpar sig boken för din undervisning. Utförliga genomgångar vävs ihop med utmanande övningar som kan testas i bokens egna app-baserade pythonmiljö.