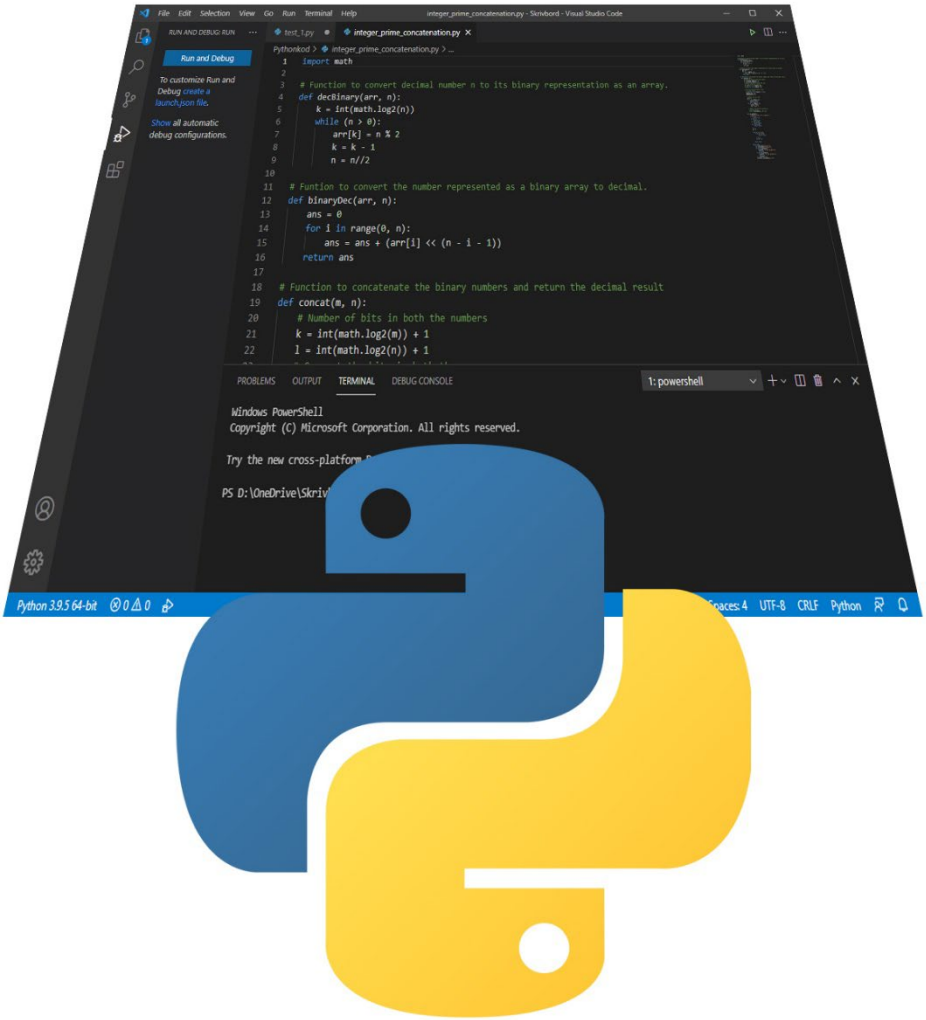
En ultrakort introduktion till Python

Simon K. Jensen



Göteborgs
Tekniska College
Utveckla & Utvecklas. Nå din fulla potential.

Innehåll

1. Introduktion 5
1.1. Tolkade programspråk 5
1.2. Installation av Python 6
1.2.1. Ladda ner från *www.python.org* 6
1.2.2. Ladda ner *Visual Studio Code* 7
1.2.3. Andra Pythonmiljöer 7
1.3. Kod i *Python Shell* 8
1.4. Spara kod i *.py*-filer 9
1.5. Kodord i språket Python 10
1.6. Pythonvarianter 10
2. Variabler 11
2.1. Namngivning av variabler 11
2.2. Skapa en variabel 11
2.3. Skapa flera variabler samtidigt 11
2.4. Få fram värdet av en variabel 11
2.5. Variablers ID 11
2.6. Ta bort en variabel 12
3. Typer 13
3.1. Lite om begreppet *uttryck* 13
3.2. De inbyggda numeriska typerna 13
3.2.1. Heltalstypen **`int`** 13
3.2.2. Flytalstypen **`float`** 14
3.2.3. Typen **`complex`** 14
3.3. Den inbyggda typen för hantering av text 14
3.3.1. Indexering av textsträngar 14
3.3.2. Metoder i **`str`** 15
3.4. Den inbyggda logiska typen 15
3.5. Typen **`Decimal`** (från modulen **`decimal`**) 16
3.6. Den inbyggda sekvenstypen **`range`** 16
4. Operatorer 17
4.1. Aritmetiska operatorer 17
4.2. Operatorer för typen **`str`** 17
4.3. Jämförelseoperatorer 18
4.4. Logiska operatorer 18
4.5. Tilldelningsoperatorer 18
5. Satser 19
5.1. Enkla satser 19

5.1.1. Uttryckssatser 19
5.1.1. Andra enkla satser 19
5.2. Sammansatta satser 19
5.2.1. **if**-sats 19
5.2.2. **while**-sats 20
5.2.3. **for**-sats 21
5.2.4. Kodordet **else** i en loop 22
5.2.5. Indentering (indrag) 22
5.2.6. Satsen **pass** 22
6. Fler typer 23
6.1. Sekvenstypen **list** 23
6.1.1. Metoder i typen **list** 23
6.1.2. Ta bort ett element i en lista 23
6.2. Sekvenstypen **tuple** 23
6.3. Att använda *dellistor* (*slices*) 24
6.4. Typen **set** 24
6.4.1. Metoder i typen **set** 25
6.5. Typen **dict** 25
6.6. *Oföränderliga* och *föränderliga* typer 25
7. Funktioner 27
7.1. Inbyggda funktioner 27
7.2. Funktionsanrop 27
7.3. Att skapa egna funktioner 27
7.3.1. Funktionsdefinitioner 27
7.3.2. Satsen **return** 28
7.4. Lokala variabler och globala variabler 29
7.5. Dokumentsträngar i funktioner 29
8. Moduler 30
8.1. Kodorden **import**, **from** och **as** 30
8.2. Moduler i standardbiblioteket 30
8.3. Andra moduler 31
8.3.1. Installation av modulen **matplotlib** med *pip* 31
2. I kommandotolken som öppnas skriv följande kommando: 32
8.4. Ett exempel med modulen **matplotlib** 32
8.5. Skriva egna moduler 33
8.5.1. Variabeln **__name__** (och värdet **"__main__"**) 33
9. Läsa från filer och skriva till filer 34
9.1. Den inbyggda funktionen **open** 34

9.2. Metoderna **read** och **readline** ... 34
9.3. Metoderna **write** och **close** ... 35
9.4. Modulen **os** ... 35
10. Tillämpningar ... 36
10.1. Förslag på vidare studier ... 36
10.1.1. Grafana ... 36
10.1.2. Node-RED ... 36
10.1.3. Cisco Packet Tracer ... 36
10.2. Generellt om industriella tillämpningar ... 36
11. Övningar ... 37
11.1. Övningar till Avsnitt 1 ... 37
11.2. Övningar till Avsnitt 2 ... 37
11.3. Övningar till Avsnitt 3 ... 37
11.4. Övningar till Avsnitt 4 ... 37
11.5. Övning till Avsnitt 5 ... 37
11.6. Övningar till Avsnitt 6 ... 37
11.7. Övningar till Avsnitt 7 ... 38
11.8. Övningar till Avsnitt 8 ... 38
11.9. Övningar till Avsnitt 9 ... 38

Detta verk är skyddat av Lag (1960:729) om upphovsrätt till litterära och konstnärliga verk.
Redigering av dokumentet (härunder utvidgning och/eller minskning av dokumentets omfattning) är ej tillåtet.
Dokumentet får fritt skrivas ut i sin helhet för användning i undervisningssammanhang.

Version 1.1. Uppdaterat 10 december 2021. Utgivningsdatum: 20 juni 2021.

© Simon K. Jensen och Göteborgs Tekniska College (2021)
Göteborgs Tekniska College AB, Göteborg (www.goteborgstekniskacollege.se)

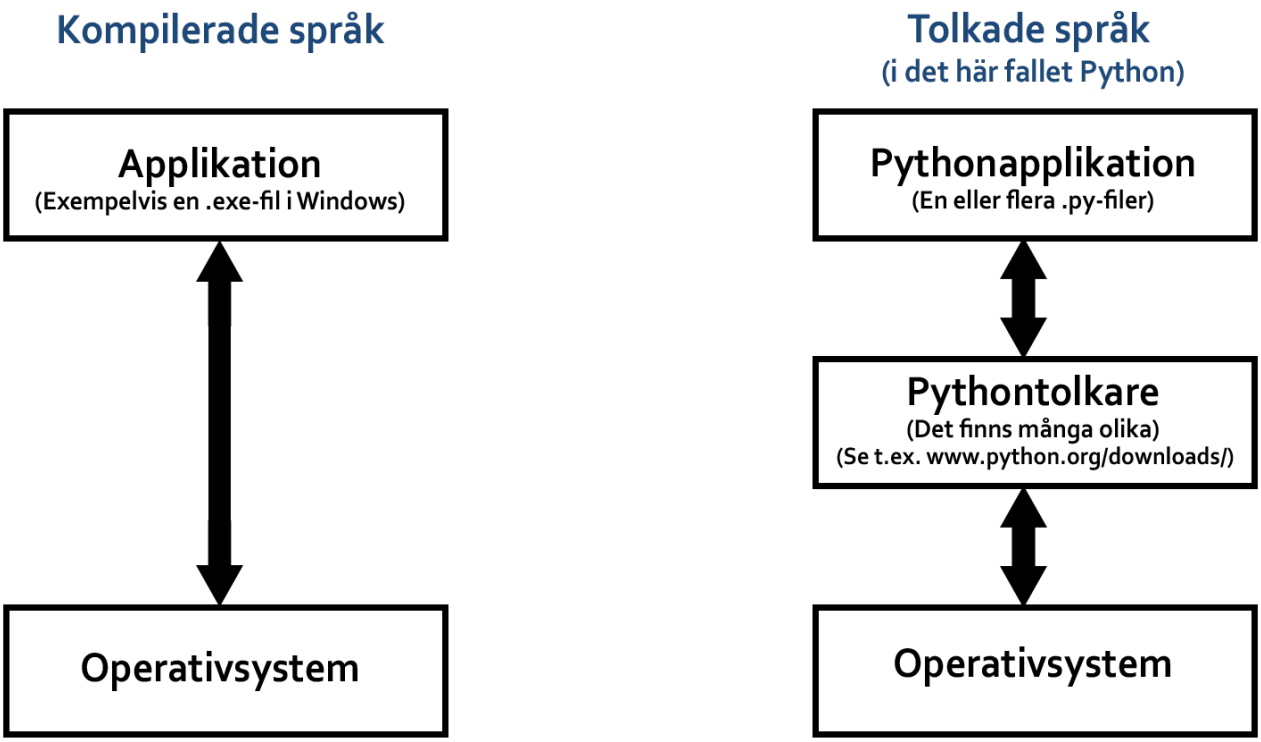
Produktion och layout: OffCircle AB, Göteborg (www.offcircle.com)
Omslag och bilder: Simon K. Jensen

1. Introduktion

Programpråket *Python* släpptes 1991 och har sedan dess vuxit och förändrats mycket i takt med att dess spridning ökade. Idag är Python ett av de mest populära programspråken, och det används i stor skala både inom industrin och den akademiska sektorn.

1.1. Tolkade programspråk

I motsats till *kompilerade* språk (till exempel C++) är Python är ett *tolkat* språk, vilket innebär att det finns ett extra lager (en *tolkare*) mellan operativsystemet och de applikationer (program) som är skrivna i Python. Tolkaren är i sig själv en applikation som finns installerad på den dator där Python-applikationerna körs. För att kunna köra Python måste det finnas en tolkare.



Figur 1. Skillnaden mellan *kompilerade språk* och *tolkade språk*.

Operativsystemet är det program som kommunicerar med hårddiskar, skärmar, skrivare och alla övriga *yttre enheter* (i bilden ovan visas inga yttre enheter). För att kunna köra en applikation måste det vanligtvis finnas ett operativsystem (det finns undantag från denna regel, exempelvis så kallade *inbyggda system*).

Lite kortfattat kan man säga att när en applikation *kompileras* så översätts programkoden till maskinkod som kan köras på det aktuella systemet. Men när en applikation *tolkas* så skapas aldrig någon fristående fil med maskinkod som kan startas direkt av operativsystemet. I stället skapar tolkaren ett antal instruktioner, en i taget, som därefter skickas till operativsystemet och utförs, en i taget.

Detta innebär att tolkade språk kan användas på ett sätt som skiljer sig från de kompilerade språken. Man kan nämligen skriva in instruktioner (kommandon), en i taget, i en så kallad *terminal* (även kallad *konsoll*). Terminalen är i ett litet program som gör det möjligt för oss att kommunicera direkt med tolkaren. En Pythontolkare som körs i terminalen kallas ett *skal* (*Pythonskal*), men oftast används de engelska benämningarna *shell* eller *Python shell*, och de instruktioner vi skriver i terminalen kallas *shell-kommandon*.

Att skiva kod i terminalen är väldigt användbart när man håller på att lära sig språket och när man felsöker, men det används även i professionella sammanhang för att exempelvis utföra fristående kommandon i stora system, till exempel sådant som vanliga användare inte kan göra från det vanliga gränssnittet (konfigurera en databas, utföra diagnostiska test samt övriga tekniska/administrativa arbetsuppgifter).

1.2. Installation av Python

Python kan användas i flera olika sammanhang, och det är fritt fram att välja den eller de tolkare (med tillhörande terminal och utvecklingsmiljö) man helst vill använda för att lära sig språket. Härunder följer lite information om två av de vanligaste. Installera dessa (börja med den förstnämnda nedan).

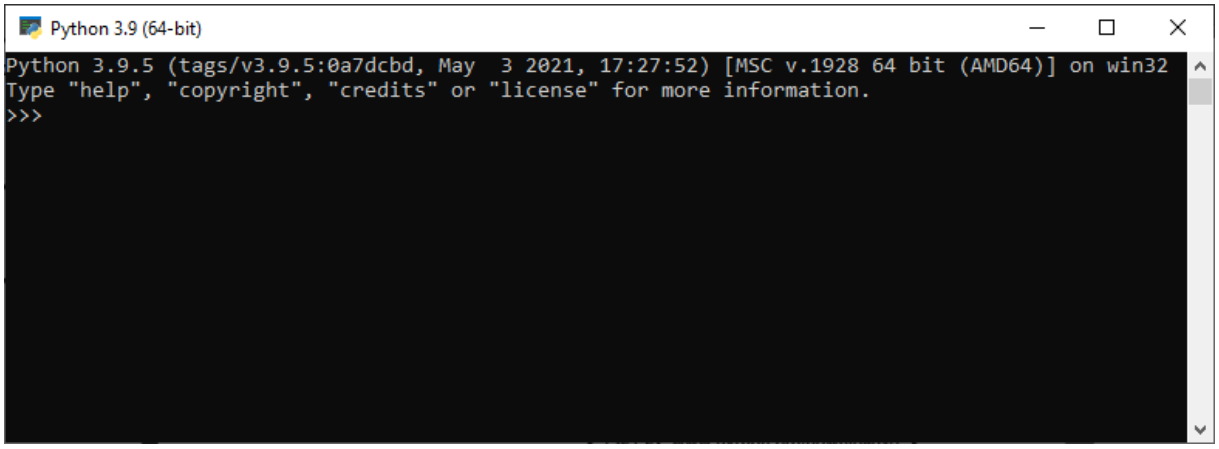
1.2.1. Ladda ner från *www.python.org*

På www.python.org/downloads hittar du en stor gul knapp som startar nerladdningen (i skrivande stund står det **Download Python 3.9.5**). Klicka och följ instruktionerna för att slutföra installationen, se dock nedan.

VIKTIGT: När installationsprogrammet startar, se till att rutan **Add Python to PATH** är *ifylld* innan du fortsätter.

När installationen är klar har 4 nya punkter lagts in på din startmeny i Windows:

Python 3.9 (64-bit)
Detta är terminalen som kör Python Shell. Här kan man direkt kommunicera med tolkaren, skriva kommandon, testa program och även skriva pyttesmå program. Men om man ska skriva längre program behövs en *textredigerare* av något slag (se avsnittet *Python IDLE (3.9 64-bit)* nedan).

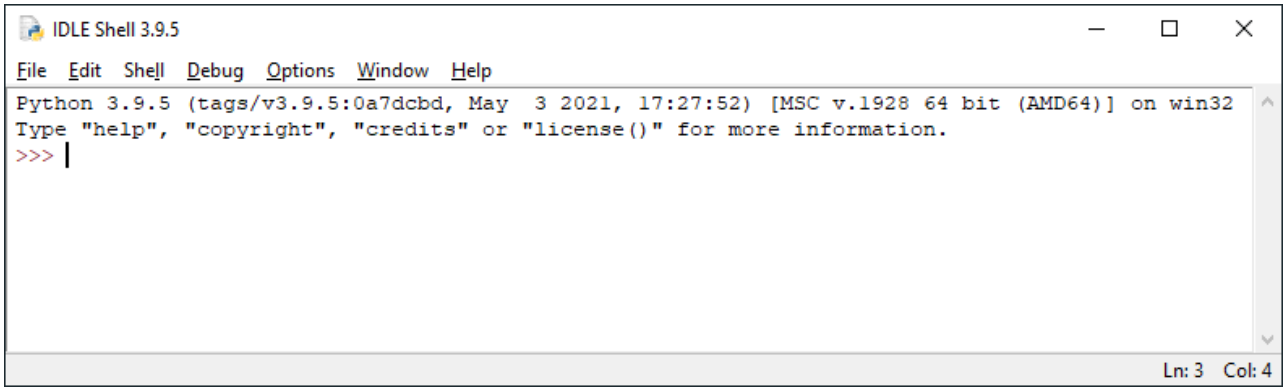


Figur 2. Programmet *Python Shell* (version 3.9.5) på en vanlig Windowsdator.

Python 3.9 Manuals (64-bit)
Detta är dokumentation om Python. Finns även på docs.python.org/3/

Python 3.9 Module Docs (64-bit)
Dokumentation av mer teknisk natur (med information om exempelvis vilka DLL-filer som används för varje så kallad *modul* i Python m.m., vilket inte är så viktigt för nybörjare).

Python IDLE (3.9 64-bit)
IDLE är en förkortning av *Integrated Development and Learning Environment*. Python IDLE är en mer interaktiv terminal som innehåller en textredigerare för språket Python. Det är här man vanligtvis arbetar med sina program (som kan öppnas/ändras/sparas som *.py*-filer).

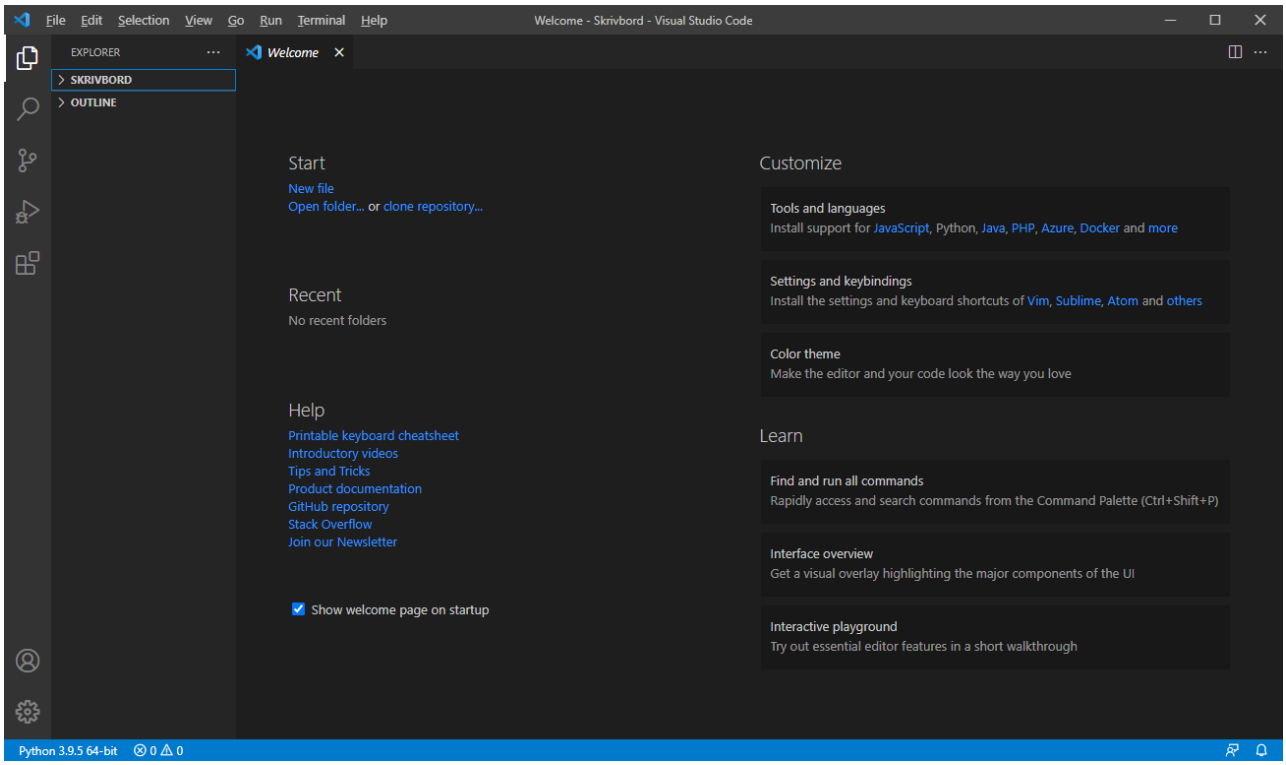


1.2.2. Ladda ner *Visual Studio Code*

Man kan komma långt med Python IDLE. Men för att kunna arbeta med stora projekt, eventuellt tillsammans med andra, krävs en så kallad *utvecklingsmiljö* med mer funktionalitet än vad som finns i Python IDLE.
Det finns ett flertal utvecklingsmiljöer på marknaden. En av de mest utbredda heter *Visual Studio Code* (eller bara *VS Code*) som kan laddas ner gratis från code.visualstudio.com.

För nybörjare kan VS Code verka lite väl komplicerad eftersom det finns många inställningar som kan göras. Några korta introduktionsfilmer finns här: code.visualstudio.com/docs/introvideos/basics (under varje film finns en länk till nästa film).

Visual Studio Code kan användas för många olika programspråk. För att vi ska kunna bygga, testa och köra just Python, så måste Python installeras. Detta görs enkelt direkt från Visual Studio Code (som automatiskt meddelar oss att vi behöver det när vi börjar skriva kod), men det går också bra att installera direkt från marketplace.visualstudio.com/items?itemName=ms-python.python.



Figur 4. Välkomstskärmen i *VS Code* på en vanlig Windowsdator (i den nedersta blå raden ses att Python har installerats och är valt som språk).

1.2.3. Andra Pythonmiljöer

Det finns Pythontolkare för nästan alla plattformar (*Windows*, *Linux*, *Unix*, *macOS*, *iOS*, *Android* med flera). Har man en iPad, kan man exempelvis använda apparna *Pythonista* eller *Python Box*. Dessutom finns ett antal bra webbplatser för utveckling *online* (testa till exempel replit.com, se nedan).

Figur 5. Utveckling i Python online på webbplatsen *replit.com*.

1.3. Kod i *Python Shell*

När Python Shell startas, antingen separat (se Figur 2) eller som en del av IDLE (se Figur 3), är det första man ser den s.k. *primära prompten* som består av tecknen >>> vilket markerar att tolkaren är redo för att ta emot instruktioner. All inmatning måste vara giltig Pythonkod, och varje inmatning avslutas med **Enter**.

Vissa instruktioner sträcker sig över flera separata rader, och då ser man den *sekundära prompten* som består av tecknen `...` (tre punkter).

En av de viktigaste sakerna att börja med är kommandot `help()`, vilket startar Pythons inbyggda hjälpfunktion `help`. När denna funktion körs byts den primära prompten ut mot prompten `help>` och då kan man få information om i princip allt som finns tillgängligt i den aktuella Pythoninstallationen. Hjälpfunktionen avslutas med kommandot `quit`, vilket gör att vi får tillbaka den primära prompten.

Det går även bra att direkt starta ett visst hjälpavsnitt med `help(text)`, där `text` beskriver det man vill ha information om. Testa till exempel att skriva `help(complex)` för att se information om Pythons inbyggda typ för hantering av komplexa tal, eller `help(id)` för information om *identitetsfunktionen* `id` (mer om denna funktion senare).

Om hjälpen i Python Shell innehåller mycket information delas denna upp i sidor som avslutas med texten `-- More --` (och mellanrumstangenten eller **Enter** visar nästa sida, medan tangenten **Q** avslutar).

Python Shell är praktisk när man vill testa en del av en kod eller helt enkelt vill använda Python som en sort avancerad miniräknare. Testa att skriva in följande (tryck **Enter** efter varje rad):

```
a = 2
b = 3
c = 3.5
a + b * c / 2
```

Du bör rimligen få resultatet `7.25` (i all programmering används *punkt* i stället för *komma* som decimaltecken).

Sista inmatningen ovan tolkas $a + \frac{b \cdot c}{2}$ som, i enlighet med de vanliga aritmetiska prioriteringsreglerna.

Symbolerna `a`, `b` och `c` i koden ovan kallas *variabler* (ibland påträffas benämningarna *etiketter* och *labels*).

Låt oss testa en annan beräkning, nämligen omvandling av en temperatur från °F to °C (alltså från Fahrenheit till Celsius):

```
F = 93.2
C = (F - 32) * 5 / 9
print("Grader Celsius: ", C)
```

Då får vi veta att 93,2°F motsvarar 34°C, och här har vi använt funktionen `print` för att mata ut lite extra information på skärmen tillsammans med resultatet av beräkningen.

I språket Python finns ett antal inbyggda *funktioner* (några exempel är `help`, `id`, `complex` och `print` som alla har nämnts tidigare i detta avsnitt). Men väldigt ofta behöver en massa andra funktioner som *inte* är inbyggda i Python utan i stället finns i så kallade *moduler*. Som exempel, låt oss beräkna periodtiden för en harmonisk rörelse av något som är fäst vid en fjäder. Det finns en välkänd förenklad formel för detta:

$T = 2\pi\sqrt{\frac{m}{k}}$, där T är periodtiden, m är massan av det som hänger i fjädern och k är fjäderkonstanten.

Låt oss säga att massan är 0,5 kg och fjädern har fjäderkonstanten 1,9 N/m.
För att kunna använda *roten ur* samt värdet π (pi) måste vi *importera* en modul som heter `math`. Så här ser det ut:

```
import math
2 * math.pi * math.sqrt(0.5/1.9)
```

Då får vi resultatet `3.223206053652882`, och ser att periodtiden är dryga tre sekunder.

När väl modulen `math` har importerats är den tillgänglig tills vi stänger ner terminalen, och vi kan därmed göra flera beräkningar i samma fönster utan att behöva importera igen.

För att se en lista med alla matematikfunktionerna i modulen `math`, kan `help` vara nyttig:

```
help(math) # Detta startar hjälpavsnittet som handlar om modulen math.
```

På raden ovan har vi använt `#`-tecknet för att skriva en *kommentar* i koden. Kommentarer ignoreras av tolkaren och används enbart av den (eller de) som skriver koden. Varje gång tolkaren påträffar tecknet `#` på en rad så tolkas resten av raden som en kommentar (dock inte om `#`-tecknet finns inuti s.k. *textsträngar*).

1.4. Spara kod i *.py*-filer

I förra avsnittet skrev vi några *instruktioner* (eller *kommandon* om man tycker att detta ord passar bättre). Dessa kommandon skrevs in, en i taget, och utfördes efterhand som de skrevs in, och den korrekta datalogiska termen för dem är *satser*. En sats är en instruktion som kan läsas och översättas av antingen en *kompilator* eller en *tolkare* (i fallet Python rör det sig om en tolkare). En *applikation* (ett *program*) är en mängd satser som utförs i en viss ordningsföljd (uppifrån och ner, från vänster till höger).

Programkod sparas i vanliga textfiler. För att Pythontolkaren ska känna igen filerna har de filändelsen *.py*, men de kan öppnas i ett valfritt textredigeringsprogram (till exempel Notepad).

För att *skriva* ett Pythonprogram behöver vi endast skriva ett antal satser i ett textdokument. Men för att *köra* programmet behövs ju tolkaren, och därför är det praktiskt att använda den textredigerare som finns inbyggd i IDLE (eller VS Code) tillsammans med tolkaren. Ytterligare en fördel med detta är att denna textredigerare *förstår* Python, ungefär som att Word har en inbyggd stavnings- och grammatikkontroll.

För att skapa en ny *.py*-fil i IDLE, välj *File* > *New File* från menyn eller klicka **Ctrl** + **N**. Då öppnas ett nytt textfönster i vilket koden kan skrivas. När programmet har skrivits färdigt kan det startas med tangenten **F5** (eller *Run* > *Run Module* från menyn). Om man vill kontrollera koden *utan* att köra den kan man göra detta med **Alt** + **N** (eller *Run* > *Check Module* från menyn). När programmet körs sker in- och utmaningen i det första fönstret, alltså terminalfönstret i IDLE (det som visar prompten **>>>**).

För att skapa en ny *.py*-fil i VS Code, gör samma sak: Välj *File* > *New File* från menyn eller klicka **Ctrl** + **N**. Då öppnas en ny flik. Om inget programspråk har valts innan, så visas följande text:
Select a language to get started. Start typing to dismiss, or don't show this again.

Här kan man klicka på *Select a language* och på ett enkelt sätt välja språket Python. Sedan är det bara att börja skriva. När programmet ska köras, klicka **F5** eller välj *Run* > *Start Debugging* från menyn (spara filen innan). Programmet körs i en terminal som är inbyggd i VS Code, oftast visas den i bottensektionen. Det finns alltså inget separat terminalfönster som i IDLE (om Pythonprogrammet i sin tur använder några av de många moduler som finns för att skapa *grafiska* användargränssnitt, så gäller inte detta).

I förra avsnittet räknade vi ut att 93,2 °F motsvarar 34 °C. Låt oss skapa ett litet program utifrån detta. Skapa en ny fil och skriv följande:

```
F = input("Grader Fahrenheit? ")
F = float(F)
C = (F - 32) * 5 / 9
print("Grader Celsius: ", C))
```

På första raden ovan använder vi Pythons inbyggda `input`-funktion. När denna funktion används skriver den ut texten `Grader Fahrenheit?` och sedan väntar programmet på att en inmatning ska ske. Denna inmatning placeras i variabeln `F`. Men `input` returnerar alltid en *text*, och vi vill ha ett *decimaltal* eftersom vi ju ska använda detta tal i beräkningen på tredje raden. Därför använder vi funktionen `Float` för att omvandla det som finns i `F` till ett decimaltal (allt detta förklaras närmare i nästa avsnitt).

Nu kan programmet köras om och om igen, och för varje körning kan vi ange ett *nytt* temperaturvärde.

1.5. Kodord i språket Python

Ett programspråk är, precis som ett vanligt skriftspråk, uppbyggt av ett antal *ord*. I programspråk kallas dessa *kodord*, *reserverade ord* eller *nyckelord* (efter engelskans *keywords*). Efterhand som språket Python utökas kommer med största sannolikhet fler kodord att läggas till. Men Python har relativt få kodord jämfört med andra programspråk (C++ har till exempel strax över hundra). I skrivande stund innehåller den senaste versionen (Python 3.9) följande 35 kodord, varav de 3 `gröna` är de två *logiska värdena* samt det *tomma värdet* (de 12 `röda` hinner vi inte prata mer om detta kompendium, men de är inte speciellt svåra att lära sig när man väl har kommit en bit på vägen med språket):

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

Tabell 1. Samtliga kodord i Python (juni 2021).

1.6. Pythonvarianter

Den ursprungliga tolkaren för *Python*, den *originala* versionen (som även kallas ***CPython***), är skriven i språket C, och det är den man kan hämta från www.python.org. Men eftersom Python bygger på *open source* är det fritt fram att utveckla andra varianter. Och det har gjorts.

Cython är namnet på en variant som gör det möjligt att skriva egna *utvidgningar* (*extensions*) till språket Python (de skrivs i språket C). Cython består av CPython *plus* en del C-funktionalitet. För att utnyttja kraften i Cython bör man vara en erfaren C-programmerare.

En Pythontolkare översätter vanligtvis Pythonkod till så kallad *bytekod*. Bytekod påminner om maskinkod, och det går att *köra* på en dator (om man har en så kallad *bytekodtolkare* installerad). Maskinkod är specifik för en viss typ av dator, medan bytekod är *portabel*, vilket kortfattat innebär att den kan flyttas mellan olika sorters maskiner.

Språket Java används, precis som språket C, i stor utsträckning, speciellt vid industriella tillämpningar. Och Java har en bytekodtolkare med namnet *JVM* (*Java Virtual Machine*). Därför har det byggts en variant av Python som använder Java för att skapa sin bytekod (i stället för C) så koden kan köras på JVM. Denna Pythonvariant har namnet ***Jython***.

PyPy är en variant som är skriven i Python (ja, en Pythontolkare skriven i Python). Denna variant använder en teknik som heter *JIT-kompilering* (*Just-in-Time Compilation*) för att kompilera vissa delar av Pythonkoden till den aktuella maskinens *maskinkod* (i stället för bytekod), vilket ofta ger ökad prestanda. Detaljerna är relativt komplicerade och utelämnas här, men läs gärna mer på en.wikipedia.org/wiki/Just-in-time_compilation och en.wikipedia.org/wiki/Interpreter_(computing).

Det finns fler Pythonvarianter, men de fyra ovanstående är de vanligaste i professionella sammanhang.

2. Variabler

En *variabel* är (lite förenklat) en *behållare* som kan innehålla data. Variabler finns i alla programspråk. I Python kallas de ibland *labels* eller *etiketter*, vilket strängt taget är mer korrekt (se Avsnitt 2.5).

2.1. Namngivning av variabler

En variabel måste ha ett *namn*, och det väljs av den programmerare som skriver programmet. Välj alltid namn som är relevanta i sammanhanget.

Namnet får bara innehålla *små bokstäver* (`a` till `z`), *stora bokstäver* (`A` till `Z`), *siffror* (`0` till `9`) och *underscore* (alltså tecknet `_`). Namnet får inte *börja* med en siffra. Namnet får inte vara ett av kodorden i Tabell 1.

Namn på variabler är *skiftlägeskänsliga*, vilket innebär att exempelvis `Age`, `age` och `AGE` är tre *olika* variabelnamn.

Ett par exempel (efter varje variabel står en kommentar som börjar med tecknet `#`):

```
voltage = 35      # En variabel med namnet voltage och heltalsvärdet 35.
brand = "Volvo"   # En variabel med namnet brand och textvärdet "Volvo".
x = 12.5364       # En variabel med namnet x och värdet 12.5364.
```

2.2. Skapa en variabel

För att skapa en ny variabel behöver man bara hitta på ett lämpligt namn och sedan *tilldela* ett *värde* till namnet. Man tilldelar ett värde till en variabel med *tilldelningsoperatorn* `=` (likhetstecknet). Om variabeln redan finns, så får den ett nytt värde. Om variabeln inte finns, så skapas den.

2.3. Skapa flera variabler samtidigt

På grund av att det finns något som heter *tuplar* (se Avsnit 6.2) så är det möjligt att skapa flera variabler på samma rad:

```
a, b, c = 2, 1.5, -4
print(a + b + c)
```

Antalet variabelnamn till vänster om likhetstecknet måste vara exakt samma som antalet värden på högra sidan. I exemplet ovan får vi resultatet `-0.5` utskrivet i terminalen.

2.4. Få fram värdet av en variabel

För att få fram värdet av en variabel behöver man bara skriva namnet på variabeln. Om variabeln inte har något värde betyder det att den inte har skapats ännu, och då får man ett felmeddelande om man försöker använda den.

2.5. Variablers ID

I Python är variabler lite annorlunda än i många andra språk, eftersom de egentligen inte *innehåller* ett värde (fast vi säger att de gör det) utan de *refererar* till ett värde. Så om två variabler innehåller samma värde, så *är* de egentligen samma variabel. För att se detta kan man använda den inbyggda funktionen `id`.

Varje värde som används i ett program måste lagras i datorns minne (åtminstone under den tiden då det används). Datorns minne är uppbyggt av *byte* (små minnespositioner som kan lagra en liten mängd data), och dessa byte ligger i en numrerad ordningsföljd (byte nummer 0, efterföljt av byte nummer 1, efterföljt av byte nummer 2, och så vidare). Numret som används i denna ordningsföljd kallas en *adress*.

Ofta är det värde som ska lagras för stort för att få plats i en enda byte, och då behövs flera byte för att lagra värdet. Då gäller alltid att dessa byte ligger *efter varandra*, i följd. I Python kallas adressen till *den första byten* i ett värde för värdets *ID*, och vi kan få fram detta med funktionen `id`. Om två variabler refererar till samma värde har de samma *ID* (ofta ett väldigt stort heltal eftersom datorn har miljardtals byte).

Ett par exempel (testa i Python Shell):

```
id(10)
id(10.0)
a = 7
id(a)
b = 7
id(b)
c = 7.0
id(c)
```

Här ser vi att `id(a)` och `id(b)` ger samma resultat medan `id(c)` ger ett annat resultat. Orsaken är att även om innehållet i `c` också är *sju*, så har värdet `7.0` (ett *decimaltal*) en annan *typ* än värdet `7` (ett *heltal*). Detta förklaras i nästa avsnitt.

Om två variabler (`a` och `b`) har samma värde med samma typ gäller nästan alltid att `id(a) == id(b)`, vilket kan skrivas kortare som `a is b` (både `==` och `is` är s.k. *operatorer*, se Avsnitt 4).

2.6. Ta bort en variabel

Om en variabel har skapats kan den tas bort med följande sats (vi går igenom begreppet *sats* i avsnitt 5):

```
del variabelnamn
```

Vi tar ett exempel:

```
a = 5
b = 5
print(a)
del a
print(b)
print(a)
```

Här skapas variablerna `a` och `b` (de refererar till samma värde). Sedan skickas värdet av `a` till funktionen `print`, och på nästa rad tas `a` bort. Sedan skickas värdet av `b` till funktionen `print`, men därefter blir det stopp och tolkaren visar oss följande felmeddelande:

Exception has occurred: NameError
name 'a' is not defined

Orsaken till felet är att i sista raden ovan används variabeln `a` som inte längre finns.

3. Typer

En *typ* (eller *datatyp*) är ett sätt för tolkaren att avgöra på vilket sätt den ska tolka den data som finns lagrad i programmet. Detta är viktigt eftersom olika typer behandlas på olika sätt av tolkaren. I Python finns det ett antal *inbyggda typer*, men därutöver finns det många typer som vi kan importera från de *moduler* som finns i Pythons *standardbibliotek* (se docs.python.org/3/library). Dessutom kan vi skapa egna typer utifrån behov.

3.1. Lite om begreppet *uttryck*

Alla värden i ett program har en typ. Allt som variabler kan referera till har en typ. Allt som vi matar in när programmet körs har en typ. Det känns som att vi behöver ett ord för ”sådant som har en typ”. Och det ordet är *uttryck* (*expression* på engelska). Ett *uttryck* har ett *värde* och en *typ*.

Exempel på uttryck är *namn på variabler*, *heltal*, *decimaltal*, *textsträngar*, *funktioner* och *operationer* (vi återkommer till dessa begrepp).

Vissa uttryck är värden som har skrivits in direkt i koden (och alltså inte lagrats i exempelvis en variabel). Sådana uttryck kallas *literaler*. Följande uttryck är exempel på literaler:

```
7
5.8
"No"
"5"
```

För att få reda på typen för ett visst uttryck används funktionen `type`:

```
type(uttryck) # Denna funktion ger oss typen för uttrycket
```

För en komplett lista med alla inbyggda typer samt typerna i standardbiblioteket hänvisas till docs.python.org/3/library. I de efterföljande avsnitten går vi igenom några av de vanligaste typerna.

3.2. De inbyggda numeriska typerna

De numeriska typerna används för tre olika sorters *tal*, nämligen *heltal*, *flyttal* och *komplexa tal*.

3.2.1. Heltalstypen **`int`**

Ett *heltal* (*integer* på engelska) är ett tal utan decimaler, positivt eller negativt (eller noll). I Python hanteras heltal med typen `int`, vilket innebär att varje gång vi skriver ett heltal i vår kod får det automatiskt denna typ.

Om vi vill konstruera ett heltal utifrån en annan typ kan det göras med den inbyggda funktionen `int` (denna funktion kallas en *`int`-konstruktor*). Ett litet exempel:

```
a = 15          # Variabeln a får typen int och värdet 15
b = int(15.7)   # Variabeln b får typen int och värdet 15 (omvandlat från float)
c = int(-3.6)   # Variabeln c får typen int och värdet -3 (omvandlat från float)
d = int("23")   # Variabeln d får typen int och värdet 23 (omvandlat från str)
e = 123456789012345678901234567890123456789012345 # Variabeln e har 45 siffror
```

Det framgår av exemplet att `int`-konstruktorn aldrig utför någon matematisk avrundning uppåt vid omvandling från decimaltal (`float`) till heltal (`int`). I stället tas decimalerna helt enkelt bort.

Typen `int` har ingen övre gräns för hur många siffror ett tal får innehålla. Tolkaren reserverar precis den plats i minnet som är nödvändig för att heltalet ska få plats. Vill man veta hur många *bitar* som krävs för att lagra ett visst heltal i minnet kan man använda funktionen `bit_length` som finns inbyggd i typen `int`.
Ett par exempel:

```
print(a.bit_length())     # Skriver 4 på skärmen (a ovan har 4 bitar)
print(e.bit_length())     # Skriver 164 på skärmen (e ovan har 164 bitar)
print((27).bit_length()) # Skriver 5 på skärmen (talet 27 har 5 bitar)
```

Eftersom funktionen `bit_length` är inbyggd i en *typ* (i det fallet `int`) så innebär det att vartenda heltal har tillgång till funktionen. Man kallar funktionen en *metod* (och säger ibland att den är *medlem* i `int`).

Utöver `bit_length` innehåller typen `int` även metoderna `to_bytes`, `from_bytes` och `as_integer_ratio` som kan vara bra att känna till, men vi går inte in i detalj kring dessa. I avsnitt 4 ska vi i stället titta de så kallade *operatorerna* som kan användas för värden med typen `int`.

3.2.2. Flytalstypen **`float`**

Men vi kan ju inte programmera uteslutande med hjälp av heltal. Vi behöver *decimaltal*, och sådana kan vi enkelt få med den inbyggda typen `float` (eller med typen `Decimal` som ofta är ett bättre alternativ, se Avsnitt 3.5). Namnet `float` har valts på grund av den teknik som används för att lagra värden med denna typ, en teknik som kallas *flyttal* (*floating-point numbers*). För mer information, börja med en.wikipedia.org/wiki/Floating-point_arithmetic.

Om vi vill konstruera ett flyttal utifrån en annan typ kan det göras med den inbyggda funktionen `float` (funktionen kallas en *`float`-konstruktor*). Ett par exempel:

```
a = 15.0          # Variabeln a får typen float och värdet 15.0
b = float(5)      # Variabeln b får typen float och värdet 5.0 (omvandlat från int)
c = float("23.4") # Variabeln c får typen float och värdet 23.4 (omvandlat från str)
```

Observera att typen `float` *inte* har obegränsad precision. Vid tal med många decimaler, använd `Decimal`.

Utöver de operatorer som vi går igenom i nästa avsnitt finns en metod för typen `float` som är bra att känna till. Vi visar med ett exempel från Python Shell:

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

Metoden `is_integer` ger resultatet `True` om värdet är ett *heltal* av typen `float`. Typen `float` har även metoderna `as_integer_ratio`, `hex` och `from_hex`, men vi går inte in i detalj kring dessa.

3.2.3. Typen **`complex`**

Denna typ används för *komplexa tal*. I matematiken skriver vi till exempel *5+4i* för att beskriva ett komplext tal med den reella delen 5 och den imaginära delen 4. Men i Python används bokstaven *j* för den imaginära delen och vi skriver `5 + 4j`. Med konstruktorn `complex` kan vi konstruera samma komplexa tal, `complex(5,4)`, och det finns även ett flertal metoder för denna typ, men dessa går vi inte in på här.

3.3. Den inbyggda typen för hantering av text

En *textsträng* (*string* på engelska) är en följd av tecken. I Python används typen `str` för att lagra text. Inmatning av text sker med funktionen `input`. Utmatning sker med `print` (vid utmatning kan de flesta typer automatiskt omvandlas till typen `str`).

Textsträngar kan skapas med konstruktorn `str` eller som literaler på följande sätt:

Enkla citattecken: `'härinne kan vi ha "dubbla" citattecken'`
Dubbla citattecken: `"härinne kan vi ha 'enkla' citattecken"`
Trippelcitattecken: `'''så här'''` *eller* `"""så här"""`

3.3.1. Indexering av textsträngar

De tecken som finns i en textsträng står i en *ordnad* följd (texten `"ABC"` är inte identisk med `"BCA"` även om det är samma tecken). För att hålla koll på ordningen använder de flesta programspråken sig av en helt vanlig heltalsnumrering. Första tecknet har nummer 0 i ordningsföljden, andra tecknet har nummer 1, tredje tecknet har nummer 2, och så vidare. Dessa nummer kallas *index*. Om man vill få fram det tecknet som motsvarar ett visst index kan man använda *slice*-operatorn (mer om *slicing* i Avsnitt 4.2 och 6.3).
Den skrivs med fyrkantparanteserna `[` och `]` inom vilka man skriver index. Så här ser det ut:

```
a = "Fast and slow"
print(a[0])  # Tecknet F skrivs ut
```

```
print(a[1])  # Tecknet a skrivs ut
print(a[2])  # Tecknet s skrivs ut
print(a[4])  # Ett mellanslag skrivs ut
print(a[12]) # Ett w skrivs ut
print(a[13]) # Ett felmeddelande skrivs ut eftersom detta index är för stort
```

I tabellen nedan presenteras metoden `find` (i flera utgåvor) som kan ge oss index för en sökt delsträng.

3.3.2. Metoder i **`str`**

Typen `str` har många metoder (alltså *funktioner* som kan användas på en text).
Nedan visas en lista på några av de mest användbara.
För en komplett lista med förklaringar, se docs.python.org/3/library/stdtypes.html#text-sequence-type-str.

Metod	Beskrivning
`st.capitalize()`	Ger kopia av strängen ***st*** med första tecknet som *versal*, resten som *gemener*.
`st.count(part)`	Ger antalet icke-överlappande förekomster av strängen ***part*** i strängen ***st***.
`st.endswith(end)`	Ger `True` om strängen ***st*** slutar med det som finns i strängen ***end***, annars `False`.
`st.find(part)`	Ger *index* för första förekomsten av strängen ***part*** i strängen ***st***. Om ***part*** inte förekommer i ***st*** ges värdet **`-1`**.
`st.find(part, start)`	Ger *index* för första förekomsten av strängen ***part*** i strängen ***st*** från och med index `start`. Om ***part*** inte förekommer ges **`-1`**.
`st.find(part, start, end)`	Ger *index* för första förekomsten av ***part*** i ***st*** från och med index `start` fram till (men ej med) index *end*. Om ***part*** ej finns ges **`-1`**.
`st.isdecimal()`	Ger `True` om strängen ***st*** enbart består av *decimala siffror* (`0` till `9`) samt vissa arabiska siffror (med mera), annars `False`.
`st.isdigit()`	Ger `True` om strängen ***st*** enbart består av *siffror* och vissa *specialtecken* (till exempel tecknet ², annars `False`.
`st.islower()`	Ger `True` om alla tecken i strängen ***st*** är *gemener*, annars `False`.
`st.isnumeric()`	Ger `True` om strängen ***st*** enbart består av *siffror* och vissa *specialtecken* (t.ex. ⅓ och ⅔), annars `False`.
`st.isupper()`	Ger `True` om alla tecken i strängen ***st*** är *versaler*, annars `False`.
`st.join(x)`	Motsatsen till `split`. Ger ny sträng där ***st*** har satts in som *separator* mellan varje element i ***x*** (som kan vara en `str`, `list` , `tuple` eller `dict`).
`st.lower()`	Ger kopia av strängen ***st*** där samtliga tecken har konverterats till *gemener* (små bokstäver).
`st.replace(old, new)`	Ger ny sträng där alla förekomster av delsträngen ***old*** i strängen ***st*** har byts mot delsträngen ***new***.
`st.split(sep)`	Motsatsen till `join`. Ger en lista (typen `list`) där elementen är de delar av ***st*** som uppkommer när ***sep*** används som *separator*.
`st.startswith(start)`	Ger `True` om strängen ***st*** börjar med tecknen i strängen ***start***, annars `False`.
`st.strip()`	Ger kopia av strängen ***st*** där inledande och avslutande så kallat *whitespace* tagits bort.
`st.upper()`	Ger kopia av strängen ***st*** där samtliga tecken har konverterats till *versaler* (stora bokstäver).

Tabell 2. Metoder tillhörande typen `str`.

3.4. Den inbyggda logiska typen

Vissa uttryck är *sanna* (`True`) och vissa är *falska* (`False`), till exempel är uttrycket `5>4` sant medan `5<4` är falskt. I Python är det typen `bool` som håller reda på sådana logiska värden. Typen `bool` har endast två möjliga värden, `True` och `False`. Ett exempel:

```
a = 5 > 4   # Variabeln a får värdet True eftersom 5 är större än 4
```

3.5. Typen **Decimal** (från modulen **decimal**)

I Pythons standardbibliotek finns en modul som heter `decimal`. För att få tillgång till denna skriver vi
`from decimal import *`
och när modulen har importerats kan vi skapa och använda uttryck av typen `Decimal`.

Med `getcontext().prec = 10` kan vi ställa in önskad precision (i det här fallet 10 *signifikanta siffror*).
Testa följande:

```
print(5/9)                    # Division int / int (resultatet är float)
print(5.0/9.0)                # Division float / float (resultatet är float)
from decimal import *
getcontext().prec = 20
print(Decimal(5/9))           # Division int / int (resultatet omvandlas till Decimal)
print(Decimal(5)/Decimal(9))  # Division Decimal / Decimal (resultatet är Decimal)
```

Resultatet blir kanske lite överraskande? Vi får ungefär 15 decimalers noggrannhet vid utskrift av en `float` dividerad med en `float`, och 20 decimaler vid en `Decimal` dividerad med en `Decimal`. Det är som det bör vara. Men `Decimal(5/9)` ger oss ett märkligt resultat (eftersom varken `5` eller `9` är `Decimal`). Från detta kan vi dra lärdomen att samtliga deluttryck bör ha typen `Decimal` om de ska ingå i en beräkning där resultatet är `Decimal`, åtminstone om vi vill ha fler korrekta decimaler än vad typen `float` erbjuder.

Det finns flera fördelar med `Decimal` framför `float` (för att se dessa fördelar, ta gärna en titt på docs.python.org/3/library/decimal.html#module-decimal). Men samtidigt är det ju mer besvärligt att använda `Decimal` eftersom denna typ inte är inbyggd utan måste importeras.

3.6. Den inbyggda sekvenstypen **range**

Nästan alla typer vi har tittat på hittills (`int`, `float`, `Decimal`, `complex` och `bool`) används för att lagra *ett* värde i taget. En variabel med en av dessa typer kan referera till *ett* värde (och endast ett värde).

Men vi har även tittat lite på typen `str` som används för att lagra *flera* tecken (i följd).
Och då uppstår naturligtvis frågan hur vi kan lagra flera värden som *inte* är tecken?

Svaret är enkelt. Med hjälp av *sekvenstyperna*. En *sekvens* är en ordnad följd data.

I Python finns det tre inbyggda sekvenstyper (`list`, `tuple` och `range`). I avsnitt 6 tittar vi på `list` och `tuple`, men för att kunna ta oss igenom Avsnitt 5 behöver vi känna till `range` redan nu.

För att skapa ett så kallat `range`-*objekt* använder vi funktionen `range`:

`range(stopp)` *eller* `range(start, stopp)` *eller* `range(start, stopp, steg)`

Detta skapar en sekvens bestående av *heltal*.

Om endast `stopp` anges börjar sekvensen med talet `0` och slutar strax innan talet `stopp`.
Om endast `start` och `stopp` anges börjar sekvensen med talet `start` och slutar strax innan talet `stopp`.
Om både `start`, `stopp` och `steg` anges kommer det att finnas ett hopp mellan varje tal i sekvensen.

Ett par exempel:

```
range(6)        # Skapar en sekvens med talen 0, 1, 2, 3, 4, 5
range(6, 14)    # Skapar en sekvens med talen 6, 7, 8, 9, 10, 11, 12, 13
range(6, 14, 3) # Skapar en sekvens med talen 6, 9, 12
```

Utöver de tre *sekvenstyperna* finns det flera andra så kallade *datasamlingstyper* i Python.
Mer om detta i Avsnitt 6.

4. Operatorer

Saker som kan *räknas ut* kallas *uttryck* i programmeringssammanhang. När ett uttryck har beräknats får man dess *värde* (uttryckets *resultat*), och alla värden som finns i datorns minne har, som tidigare nämnt, en viss *typ* (typen beror på vilket värde det rör sig om).

Vi har tidigare sett en del så kallade *simpla uttryck*, exempelvis *literaler* och namn på variabler. Men vi behöver gå igenom hur man skapar så kallade *sammansatta uttryck*.

Ett *sammansatt uttryck* består av ett eller flera *simpla uttryck* som har satts ihop med en eller flera *operatorer*. De uttryck som används av en operator kallar vi *operander*.

Med andra ord, en *operator* är något som sätter ihop ett eller flera uttryck (som kallas *operander*) till ett nytt uttryck (som kallas en *operation*). Detta nya uttryck har, precis som alla andra uttryck, ett *värde* och en *typ* (ofta en annan typ än operanderna).

Varje operator har en egen *symbol*. Nedan går vi igenom de vanligaste operatorerna.

4.1. Aritmetiska operatorer

De aritmetiska operatorerna kan användas på uttryck av typen `int`, `float`, `complex` och `Decimal` (samt ytterligare typer som inte är med här).

I Python finns det *sju* aritmetiska operatorer:

Operator	Namn på operatorn	Exempel
`+`	Addition	`1.2 + 3  # Resultat: en float med värdet 4.2` `1 + 5  # Resultat: en int med värdet 6`
`-`	Subtraktion	`3.2 - 1.3  # Resultat: en float med värdet 1.9` `1 - 5  # Resultat: en int med värdet -4`
`*`	Multiplikation	`2.0 * 4  # Resultat: en float med värdet 8.0` `2 * 4  # Resultat: en int med värdet 8`
`/`	Division	`3.0 / 4  # Resultat: en float med värdet 0.75` `3 / 4  # Resultat: en float med värdet 0.75`
`%`	Modulus	`17 % 3  # Resultat: en int med värdet 2` `17 % 2.5  # Resultat: en float med värdet 2.0`
`**`	Exponentiering (upphöjning)	`2 ** 3  # Resultat: en int med värdet 8` `4 ** 3.0  # Resultat: en float med värdet 64.0`
`//`	Heltalsdivision (eng: *floor division*)	`17 // 3  # Resultat: en int med värdet 5` `17 // 2.5  # Resultat: en float med värdet 6.0`

Tabell 3. De sju aritmetiska operatorerna.

4.2. Operatorer för typen `str`

Typen `str` har följande operatorer (det finns ett par så kallade *formateringoperatorer* som ej är med här):

Operator	Namn på operatorn	Exempel
`+`	Sammanfogning (eng: *concatenation*)	`"ab" + "bc"  # Resultat: en str med värdet "abbc"`
`*`	Repetition	`"abc" * 2  # Resultat: en str med värdet "abcabc"` `"5" * 4  # Resultat: en str med värdet "5555"`
`[]`	Slice (eller *indexering*)	`"abcde"[3]  # Resultat: en str med värdet "d"`

`[ b : e ]`	Slice (eller *range slice*)	`"abcde"[2:4]  # Resultat: en str med värdet "cd"`
		`"abcde"[2:]  # Resultat: en str med värdet "cde"`
		`"abcde"[:4]  # Resultat: en str med värdet "abcd"`

`in`	*in* (motsatsen till *not in*)	`"bc" in "abcde" # Resultat: True (typen bool)` `"bc" in "abdce" # Resultat: False (typen bool)`
`not in`	*not in* (motsatsen till *in*)	`"bc" not in "abcde"  # Resultat: False (typen bool)` `"bc" not in "abdce"  # Resultat: True (typen bool)`

Tabell 4. Operatorer för typen `str`.

4.3. Jämförelseoperatorer

Att *jämföra* uttryck med varandra är en viktig del av all programmering. I Python finns åtta operatorer för jämförelse, och nästan alla typer går att jämföra med varandra (men vissa jämförelseoperatorer är begränsade till typer med inbyggd *ordningsföljd*, vilket till exempel typen `set` inte har).

Operation	Operationens resultat
`x < y`	`True` om `x` är mindre än `y`, annars `False`
`x <= y`	`True` om `x` är *mindre än eller lika med* `y`, annars `False`
`x > y`	`True` om `x` är *större än* `y`, annars `False`
`x >= y`	`True` om `x` är *större än eller lika med* `y`, annars `False`
`x == y`	`True` om `x` är *lika med* `y`, annars `False`
`x! = y`	`True` om `x` är *skilt från* `y`, annars `False`
`x is y`	`True` om `x` är *identisk med* `y`, annars `False`
`x is not y`	`True` om `x` *inte är identisk med* `y`, annars `False`

Tabell 5. De åtta jämförelseoperatorerna.

4.4. Logiska operatorer

I Python har vi tre logiska operatorer (`and`, `or` och `not`). Dessa operatorer måste antingen ha operander av typen `bool` eller operander som kan *tolkas* som `True` eller `False`. Uttryck av typen `bool` kan till exempel vara resultatet av en jämförelseoperation eller någon annan operation som ger `True` eller `False`.

När uttryck av någon annan typ tolkas som `bool` gäller följande:
Ett värde av *sekvenstyp* (`list`, `tuple`, `set`, `dictionary`) tolkas som `True` med mindre sekvensen är *tom*.
En *textsträng* (`str`) blir `True` med mindre strängen är *tom*.
Ett *tal* (`int`, `float`, `complex`) blir `True` med mindre värdet är 0 (alltså *noll*).

Operation	Operationens resultat
`x or y`	Om `x` tolkas som `False`, så `y`, annars `x`
`x and y`	Om `x` tolkas som `False`, så `x`, annars `y`
`not x`	Om `x` tolkas som `False`, så `True`, annars `False`

Tabell 6. De tre logiska operatorerna.

Vi tar ett par exempel:

```
a = 3
b = 5
print (a > 0 and a < 5)  # True
print (b < 0 or a < 5)   # True
print( not a is 0 )      # True
print( a is not 0 )      # True
print( (not a) is 0 )    # False
print( not a )           # False
print( a or b )          # Värdet 3
print( a and b )         # Värdet 5
```

4.5. Tilldelningsoperatorer

Den vanligaste tilldelningsoperatorn skriver vi med likhetstecknet `=` som symbol. Den *vänstra* operanden refererar till en *plats* i minnet och den *högra* operanden är ett *uttryck*.

För varenda av de aritmetiska operatorerna finns det en motsvarande tilldelningsoperator som kan användas på *tal* (några av dem kan dessutom användas på *sekvenser* och *textsträngar*). De skrivs med följande symboler:

```
+=        -=        *=        /=        //=        %=        **=
```

Härunder visas med ett exempel hur de två första fungerar för vanliga variabler:

```
a = 2    # variabeln a skapas och sättas lika med 2
a += 3   # nu är variabeln a lika med 5 (den har ökat med 3)
a -= 4   # nu är variabeln a lika med 1 (den har nu minskat med 4)
```

5. Satser

En sats (*statement* på engelska) är en instruktion som ska tolkas och utföras. Alla program består av satser som tolkas i tur och ordning uppifrån och ner. I Python finns det två huvudkategorier av satser. Den ena sortens satser (*enkla satser*) skrivs på *en* så kallad *logisk rad* i Python, medan *sammansatta satser* i regel sträcker sig över *flera logiska rader* och i sig själva innehåller en eller flera satser.

5.1. Enkla satser

Alla satser vi hittills har sett är *enkla satser* (*simple statements* på engelska). Vi har skrivit *tilldelningssatser*, till exempel `a=2` eller `x-=4`, och vi har skrivit så kallade *funktionsanrop*, till exempel `float("23.4")` och `print(a)`. Det är exempel på enkla satser.

5.1.1. Uttryckssatser

Funktionsanrop är i själva verket *uttryck*. I Python skapas en så kallad *uttryckssats* (*expression statement* på engelska) helt enkelt genom att man skriver ett uttryck (på en egen rad i koden). Uttryckssatser är en underkategori inom det vi kallar enkla satser. Det som händer när en uttryckssats körs är att uttryckets resultat räknas ut (om man kör direkt i Python Shell eller någon annan terminal, så omvandlas dessutom resultatet till en text som skrivs ut på skärmen).

Vi nämnde tidigare slice-operatorn (se Avsnitt 4.2). Med den kan man få fram *delar av sekvenser*, och sådana är också uttryck. Och det finns ännu mer komplicerade uttryck (bland annat så kallade *displays*), så även om uttryckssatserna är enkla satser innebär detta inte någon begränsning för komplexiteten i dem.

5.1.1. Andra enkla satser

Uttryckssatserna är de vanligaste av de enkla satserna. Några andra är `del`-satser (se Avsnitt 2.6) och `import`-satser (se Avsnitt 8). I nästa avsnitt ska vi även använda de enkla satserna `continue` och `break`.

Det finns även en sats som inte gör något. Ingenting alls. Den heter `pass`, och i nästa avsnitt ser vi varför den ibland behövs.

5.2. Sammansatta satser

En *sammansatt sats* (*compound statement* på engelska) innehåller andra satser och styr utförandet av dessa satser. Nedan går vi igenom de tre viktigaste och i Avsnitt 7 går vi igenom så kallade *funktionsdefinitioner* som också räknas som sammansatta satser.

5.2.1. **`if`**-sats

En if-sats är en så kallad *villkorssats* (även kallad *selektionssats*) som gör det möjligt för oss att få programmet att välja *en* av flera olika vägar utifrån logiska villkor. Ett *logiskt villkor* är antingen ett uttryck vars typ är `bool` eller ett uttryck som kan tolkas som `bool` (se Avsnitt 4.4). I nedanstående exempel visas används *`lila text`* för att markera logiska villkor.

Så här ser en `if`-sats ut i sin kortaste form:

```
if logiskt villkor:
    satser
```

Satsen består av kodordet `if` efterföljt av ett *`logiskt villkor`* och tecknet *kolon* (`:`), därefter ett *radbyte* och ett *indrag* och slutligen de *satser* som ska utföras ifall det logiska villkoret är `True`. Kort sagt, om villkoret är uppfyllt (sant), så körs satserna. Därefter fortsätter programmet vidare. Det finns inga begränsningar på vilka satser som kan stå i en `if`-sats (man kan till exempel ha `if`-satser inuti en `if`-sats). Ett exempel:

```
serienummer = input("Serienummer: ")
if serienummer[0] == "A":
    print("Detta är A-serien.")
    print("Modellen finns i sortimentet")
print("Programmet avslutas.")
```

I det lilla programmet ovan frågas användaren om en text. Ifall texten börjar med ett stort `A` skrivs två textsträngar ut. Därefter skrivs textsträngen `"Programmet avslutas."` ut.

Låt oss titta `if`-satsen i sin utvidgade form:

```
if första logiska villkoret:
    första satsgruppen
elif andra logiska villkoret:
    andra satsgruppen
elif tredje logiska villkoret:
    tredje satsgruppen
elif fjärde logiska villkoret:
    fjärde satsgruppen
:
:
:
else:
    sista satsgruppen
```

Kodordet `elif` är en sammanslagning av orden `else if`, och `elif` måste placeras efter det inledande `if`. De tre lodräta punkterna ovan indikerar att det kan finnas hur många `elif` som helst (eller inga alls).

Om inte det första logiska villkoret är `True` så beräknas andra logiska villkoret. Om det ej heller är `True` så beräknas tredje villkoret, och så vidare. Om inga villkor uppfylls, så utförs satserna i den sista satsgruppen (efter `else`).

Det är frivilligt att ha med `elif` och `else`. Men `if`-satser måste alltid börja med ordet `if`.

Vi bygger vidare på exemplet från förut:

```
serienummer = input("Serienummer: ")
if serienummer[0] == "A":
    print("Detta är A-serien.")
    print("Modellen finns i sortimentet")
elif serienummer[0] == "B":
    print("Detta är B-serien.")
    print("Modellen finns i sortimentet")
elif serienummer[0] == "C":
    print("Detta är C-serien.")
    print("Modellen finns i sortimentet")
elif serienummer[0] == "D":
    print("Detta är D-serien.")
    print("Modellen finns i sortimentet")
else:
    print("Ogiltigt serienummer.")
    print("Modellen finns inte i sortimentet")
print("Programmet avslutas.")
```

När vi har gått igenom *funktioner* (Avsnitt 7) kan detta göras mycket enklare. Ovanstående kodexempel har den nackdel att samma text (exempelvis `"Modellen finns i sortimentet"`) finns med på *flera* ställen. Om denna text vid ett senare tillfälle behöver ändras måste koden redigeras på flera ställen vilket ökar risken för att man skriver fel någonstans. Vi återkommer till detta exempel i nästa avsnitt samt i Avsnitt 7.

Med `if`-satsen kan se till att programkörningen att välja mellan flera olika vägar. Men vi behöver ett sätt att *upprepa* vissa delar av ett program. För detta ändamål används en *loop* (även kallad *snurra*, *repetitionssats* eller *iteration*). I Python finns två olika sorters för detta. Vi börjar med den enklaste.

5.2.2. **while**-sats

En `while`-sats liknar på många sätt (den kortaste formen av) en `if`-sats. Så här ser den ut:

```
while logiskt villkor:
    satser
```

Om `logiskt villkor` tolkas som `True`, så körs de efterföljande satserna (precis som i en `if`-sats). Men i stället för att fortsätta vidare (vilket `if`-satsen gör), så går tolkaren *tillbaka* upp till villkoret och tittar på det *en gång till*. Om det *fortfarande* tolkas som `True`, så utförs satserna igen, och tolkaren hoppar återigen upp och tolkar villkoret … och så håller programmet på så länge som `logiskt villkor` blir `True`. Först när `logiskt villkor` tolkas som `False` stannar loopen (och satserna inuti `while`-satsen körs inte när detta sker).

Nedan används `while` för att räkna ut summan av ett antal inmatade tal *fram till dess* att summan överstiger 100, och därefter skrivs summan ut på skärmen.

```
summa = 0.0
while (summa < 100.0):
    summa += float(input("Tal? "))
print("Summan är ", summa)
```

I ovanstående exempel kommer *alla* tal vi matar in att adderas till variabeln `summa`. Om vi vill att endast *positiva* tal ska med i summan kan vi använda en `continue`-sats. När den körs hoppar programmet direkt upp till `while`-satsens logiska villkor (alltså vidare till nästa *varv* i snurran). Så här ser det ut:

```
summa = 0.0
while (summa < 100.0):
    tal = float(input("Tal? "))
    if tal < 0.0:
        continue
    summa += tal
print("Summan är ", summa)
```

Om vi vill att programmet med serienummer från förra avsnittet ska köras *om och om* kan vi använda en `while`, men då behöver vi ett sätt att avbryta loopen. Det gör vi med en `break`-sats. Så här:

```
while(True):
    serienummer = input("Serienummer (avsluta med Q): ")
    if serienummer == "Q" or serienummer == "q":
        break
    if serienummer[0] == "A":
        print("Detta är A-serien.")
        print("Modellen finns i sortimentet")
    elif serienummer[0] == "B":
        print("Detta är B-serien.")
        print("Modellen finns i sortimentet")
    elif serienummer[0] == "C":
        print("Detta är C-serien.")
        print("Modellen finns i sortimentet")
    else:
        print("Ogiltigt serienummer.")
        print("Modellen finns inte i sortimentet")
print("Programmet avslutas.")
```

5.2.3. **for**-sats

En `for`-sats används för att genomlöpa en så kallad *genomlöpningsbar typ* (*iterable* på engelska).

En *genomlöpningsbar typ* kan till exempel vara en *datasamling* av typen `list`, `tuple`, `set` eller `dict`, eller ett objekt av typen `range`, eller en *textsträng* av typen `str`. I exemplen härunder skrivs den genomlöpningsbara typen med `grön text`.

Genomlöpning (*iteration* på engelska) av är en mycket vanlig teknik i all programmering, och i språket Python har detta gjorts synnerligen enkelt:

```
for variabelnamn in genomlöpningsbar typ:
    satser
```

Satserna i en `for`-sats utförs *en gång för varje element* i den genomlöpningsbara typen.

Ett exempel:

```
for i in "abc":
  print (i)
```

Ovanstående genomlöpning har 3 *varv* (eftersom `"abc"` innehåller 3 tecken).

Låt oss mata ut 571 rader med ett plustecken (`+`) i varje rad:

```
for x in range(571):
    print("+")
```

Låt oss mata ut alla udda tal från och med 11 till och med 99:

```
for x in range(11,100,2):
    print(x)
```

5.2.4. Kodordet **else** i en loop

En `while`-sats eller en `for`-sats kan avslutas med en `else`-sats som alltid utförs så snart genomlöpningen har avslutats (men *inte* om genomlöpningen har *avbrutits* med `break`).

I nedanstående program frågas efter två heltal, ett startvärde (variabeln `start`) och ett stoppvärde (variabeln `stopp`). Med hjälp av `while` genomlöpas sedan intervallet från `start` till `stopp`. Om inga värden i intervallet är *delbara* med `17` (vilket avgörs med modulusoperatorn `%`) så utförs `else`-satsen.

```
start = int(input("Startvärde: "))
stopp = int(input("Stopptvärde: "))
i = start
while i < stopp:
    if (i % 17):
        i += 1
        continue
    break
else:
    print("Det finns inga tal i intervallet ", start, " till ", stopp, " som är delbara med 17.")
```

5.2.5. Indentering (indrag)

Samtliga satser som ingår i en `if`-, `while`- eller `for`-sats måste *indenteras* för att tolkaren ska veta att de *tillhör* en viss sats. Normalt används ett indrag som består av 2 mellanslag eller 4 mellanslag (men ett tabulatortecken går också bra). I detta kompendium har vi valt ett indrag på 4 mellanslag.

Det är viktigt att vara konsekvent i användandet av indrag. Satser tillhörande samma *block* (till exempel en `if`-, `while`- eller `for`-sats) måste indenteras med *samma* typ av mellanrum för att tolkaren ska kunna förstå det.

För `if`-satser kan man helt skippa indenteringen ifall det endast finns *en* sats efter villkoret. Då skrivs hela `if`-satsen på samma rad:

```
if a > b: print("a är större än b")
```

Det går även bra att ha med `else` på samma rad (då används inget kolon i satsen). Så här:

```
print("a är störst") if a > b else print("b är större än eller lika med a")
```

5.2.6. Satsen **pass**

I avsnitt 5.1.1 nämnde vi satsen `pass` (som inte gör något). Nu när vi har gått igenom sammansatta satser ser vi att dessa *kräver* en eller flera satser för att fungera. Då kan `pass` användas för att få klart den överordnade strukturen även om vissa satser inte är klara ännu.

```
a = 2
if a < 0:
    print(a)
elif a < 10:
    pass # Vi har ännu inte skrivit koden som ska vara här (men en sats behövs här).
else:
    print (a**3)
```

6. Fler typer

I Python finns det *många* typer och på grund av att språket är i ständig utveckling kommer det löpande nya. Sådana nya typer måste *importeras*, antingen från *standardbiblioteket* (som vi gjorde med typen `Decimal` i Avsnitt 3.5) eller från andra *moduler* som går att hämta.

Men de *inbyggda typerna* (*built-in types*) är relativt få. Vi har nämnt `int`, `float`, `complex`, `str`, `bool` och `range`. Nedan presenteras kortfattat fyra typer som används väldigt ofta. Dessa fyra typer kallas *datasamlingar* (*collections* på engelska).

Det finns några få andra inbyggda typer som vi inte hinner vi med i detta kompendium.

6.1. Sekvenstypen **`list`**

Typen `list` används för att lagra *ett antal* värden i en variabel. Värdena som lagras kan ha vilken typ som helst och de behöver inte ha samma typ. Värdena lagras i ordning och kan indexeras med `[]`-operatorn (första värdet har index `0`, nästa har index `1` och så vidare). Observera att indexering börjar med `0`.

Värdena i en lista kallas (ibland) *element*. Elementen kan ändras efter att listan har skapats. Man kan även *ta bort* element och *lägga till* element efter att listan har skapats. En lista kan innehålla *dubbletter* av element.

Ett objekt av typen `list` skapas enkelt med fyrkantparenteser inom vilka elementen separeras med komma:

```
variabelnamn = [första_värdet, andra_värdet, tredje_värdet, och så vidare, sista_värdet]
```

Alternativt kan man använda `list`-konstruktorn (se tredje kodraden nedan). Följande är exempel på listor:

```
tio_primtal = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
brands = ["Volvo", "Ford", "Opel", "Kia"]
a = list(range(5))
```

Antalet element i en lista fås med funktionen `len`, exempelvis `len(brands)` som ger oss resultatet 4.

6.1.1. Metoder i typen **`list`**

Typen `list` har ett flertal metoder (alltså funktioner som kan användas på en lista). Nedan visas en lista.

Metod	Beskrivning
`s.append(x)`	Lägger till elementet ***x*** i slutet av listan ***s***.
`s.clear()`	Tar bort samtliga element i listan ***s***.
`s.copy()`	Skapar en kopia av listan ***s***.
`s.count(e)`	Ger antalet förekomster av elementet ***e*** i listan ***s***.
`s.extend(g)`	Lägger till alla element i ***g*** (av genomlöpningsbar typ) i listan ***s***.
`s.index(e)`	Ger index för första förekomsten av elementet ***e*** i listan ***s***.
`s.insert(p, e)`	Sätter in elementet ***e*** på position ***p*** i listan ***s***.
`s.pop(p)`	Tar bort elementet på position ***p*** från listan ***s***.
`s.remove(e)`	Tar bort första förekomsten av elementet ***e*** i listan ***s***.
`s.reverse()`	Byter ordning i listan ***s*** till *omvänd ordningsföljd*.
`s.sort()`	Sorterar listan ***s*** (det går att ange *parametrar* för sorteringen). För mer information se t. ex. w3schools.com/python/ref_list_sort.asp

Tabell 7. Metoder tillhörande typen `list`.

6.1.2. Ta bort ett element i en lista

Om `s` är en lista kan man enkelt ta bort elementet med index `i` så här: `del s[i]`
Alternativt kan man skriva `s[i]=[]`, och detta även kombineras med så kallade *dellistor* (se Avsnitt 6.3).

6.2. Sekvenstypen **`tuple`**

En *tupel* är ett begrepp som hämtats från matematiken. I Python används tuplar som en sorts listor, dock med en stor skillnad: Tuplar kan *inte* ändras när de väl har skapats. Man skapar en `tuple` genom att ange

dess element (värden) i följd, separerade med komma. Nedan skapar vi en tupel med namnet `T`:

```
T = "Test", 15.0, 82      # Tillåtet sätt att skapa en tupel
```

Men för att de inte ska uppstå förvirring är det vanligt att sätta parenteser kring tupelns innehåll:

```
T = ("Test", 15.0, 82)    # Bättre sätt att skapa en tupel
```

Om man vill skapa en tupel med endast ett element, så måste man sätta ett avslutande komma efter elementet:

```
T = ("Test",)             # En tupel med endast ett element
```

Tuplar kan användas som element i andra typer (exempelvis list), de kan *indexeras* och de har metoderna `count` och `index` (se tabellen i avsnitt 6.1.1).

```
print((1,3,3,5,5,6,4,1,1,1,2,2,3,2,3,4,5).count(3))   # Skriver 4 på skärmen
```

Dessutom kan variabeldefinitioner effektiviseras med hjälp av tuplar, vilket är orsaken till att vi kan skriva till exempel `a, b, c = 1, 2, 3` för att skapa 3 variabler samtidigt (`1, 2, 3` är ett objekt av typen `tuple`).

Antalet element i en tupel fås med funktionen `len`, precis som för listor.

6.3. Att använda *dellistor* (*slices*)

En *slice* kan enkelt skapas med konstruktionen `[start : stopp : steg]`, och detta fungerar på bland annat strängar (`str`), listor (`list`) och tuplar (`tuple`). Det skapar en *delsträng*, en *dellista* eller en *deltupel*.

Om `steg` utelämnas blir steglängden 1. Om `start` eller stopp utelämnas räknas från *början* respektive *till slutet* av strängen (eller listan eller tupeln). Det går även att använda *negativa* värden och på så sätt välja slices uträknade med omvänd ordning. Några exempel:

```
(1,3,5,6,4,1,1,2,2,3,4,5)[2:10:2]  # Ger tupeln (5,4,1,2)
"abcdef"[-2::]                     # Ger strängen "ef"
```

```
a = [1, 2, 3, 4, 5, 6, 7, 8]
a[-1]          # Ger 8
a[1:4]         # Ger listan [2, 3, 4]
a[2:-1]        # Ger listan [3, 4, 5, 6, 7]
a[2:7:2]       # Ger listan [3, 5, 7]
a[2:-1:2]      # Ger listan [3, 5, 7]
a[:]           # Ger listan [1, 2, 3, 4, 5, 6, 7, 8]
a[::2]         # Ger listan [1, 3, 5, 7]
a[::-1]        # Ger listan [8, 7, 6, 5, 4, 3, 2, 1]
```

6.4. Typen **`set`**

Typen `set` används för *mängder*, och typen har konstruerats för att kunna användas på samma sätt som begreppet mängd används inom både matematiken och i vårt vanliga talspråk. Ett objekt av typen `set` kan innehålla flera olika värden (och värden av olika typ), men i motsats till tuplar och listor kan mängder *inte* innehålla dubbletter (lägger man in dubbletter, så ignoreras de).

Mängder är inte *ordnade*, vilket innebär att elementen inte ligger i en bestämd följd. Mängder är *föränderliga* (man kan lägga till och ta bort element), men elementen i en mängd är *oföränderliga* (se Avsnitt 6.6).

Mängder kan skapas med måsvingarna `{` och `}` eller med hjälp av `set`-konstruktorn. Några exempel:

```
a = {1,3,5,5,6,4,1,1,2,7,3,4,5}  # En mängd med dubbletter (som ignoreras)
b = [-3,4,-5]                    # En lista med tre element
c = set(b)                       # En mängd skapas utifrån listan b
d = set()                        # En tom mängd skapas (enda sättet att skapa en sådan)
for element in a:
    print(element) # De sju elementen i a skrivs ut en i taget (ingen fast ordning)
for element in c:
    print(element) # De tre elementen i c skrivs ut en i taget (ingen fast ordning)
```

För att ta reda på om ett element finns i en mängd (eller inte) används `in`-operatorn: `3 in c` blir `False`.
Antalet element i en mängd fås med funktionen `len`, precis som för listor och tuplar.

6.4.1. Metoder i typen **set**

Den matematiska mängdläran innehåller en del begrepp (*snitt*, *union, differens* och så vidare). Dessa begrepp har byggts in i typen `set`. Dessutom innehåller `set` metoder för att lägga till element eller ta bort element ur en mängd. Nedan visas ett urval. För mer information, se till exempel w3schools.com/python/python_ref_set.asp.

Metod	Beskrivning
`s.add(e)`	Lägger till elementet ***e*** i mängden ***s***.
`s.clear()`	Tar bort samtliga element i mängden ***s***.
`s.copy()`	Skapar en kopia av mängden ***s***.
`s.difference(t)`	Skapar en ny mängd med de element som finns i ***s*** men *inte* i ***t***.
`s.discard(e)`	Tar bort elementet ***e*** från ***s*** (om ***e*** inte finns i ***s***, händer inget).
`s.issubset(t)`	Ger `True` om alla elementen i ***s*** även finns i mängden ***t***, annars `False`.
`s.pop()`	Tar bort ett *slumpmässigt valt* element från mängden ***s***.
`s.remove(e)`	Tar bort elementet ***e*** från ***s*** (om ***e*** inte finns i ***s***, får man ett *fel*).
`s.union(t)`	Skapar en ny mängd med elementen i både ***s*** och ***t*** (utan dubbletter).

Tabell 8. Några metoder tillhörande typen `set`.

Mängder är mycket mer än ett matematiskt verktyg. Många datalogiska problem kan lösas med hjälp av mängder som kan förenkla vissa *algoritmer* avsevärt. Det faktum att en mängd antingen innehåller ett visst element eller inte samt att det inte finns dubbletter kan utnyttjas i många sammanhang.

Objekt av typen `set` kan även behandlas med följande matematiska operatorer (`A` och `B` är mängder nedan):

```
A == B      # Mängden A är lika med mängden B (True eller False)
A != B      # Mängden A är inte lika med mängden B (True eller False)
A <= B      # A är en delmängd av B
A < B       # A är en äkta delmängd av B
A >= B      # A är en övermängd av B
A > B       # A är en äkta övermängd av B
A | B       # Unionen av A och B (en ny mängd)
A & B       # Snittet av A och B (en ny mängd)
A - B       # Differensen mellan A och B (en ny mängd)
A ^ B       # Symmetriska differensen mellan A och B (en ny mängd)
```

6.5. Typen **dict**

Det som i vissa andra programspråk kallas en *associativ array* heter *dictionary* i Python. I detta kompendium går vi inte i detalj kring detta, men begreppet ska nämnas eftersom `dict` är en inbyggd typ som används ofta. För mer information, se exempelvis docs.python.org/3/tutorial/datastructures.html#dictionaries.

Objekt av typen `dict` skapas med måsvingarna `{` och `}` och dess (kommaseparerade) värden är så kallade *nyckel*:*värde*-*par* som består av en *nyckel*, ett *kolon* och ett *värde*. I exemplet nedan är nycklarna av typen `int` medan värdena är av typen `str` (det går självklart att skapa `dict`-objekt med andra nycklar och värden):

```
nummer = {1:"ett", 2:"två", 3:"tre", 4:"fyra", 5:"fem"}
fingrar = int(input("Hur många fingrar ser du? "))
print ("OK, ", nummer[fingrar])
```

Nycklarna fungerar på samma sätt som *index* i en lista eller en tupel. Med indexoperatorn kan man komma åt värdena via nycklarna. Nedan är nycklarna textsträngar och värdena är mängder.

```
season = {"spring":{3,4,5}, "summer":{6,7,8}, "autumn":{9,10}, "winter":{11,12,1,2}}
print (season["spring"])
```

6.6. *Oföränderliga* och *föränderliga* typer

Internt är språket Python uppbyggt av så kallade *objekt*. Allt som lagras i minnet under programkörning lagras i form av objekt. Varje objekt har alltid ett *ID* (en *identitet*), en *typ* och ett *värde*. Vissa objekt har dessutom ett *namn*.

```
Answer = 42          # Definition av variabel med ett namn och ett värde.
print(id(Answer))    # Utmaning av variabelns ID.
print(type(Answer))  # Utmaning av variabelns typ (i det här fallet int).
print(Answer)        # Utmaning av variabelns värde (i det här fallet 42).
```

När väl ett objekt har skapats kan dess ID *aldrig* ändras. Det används internt av tolkaren för att hitta objektet när det ska användas av programmet. Typen ändras ej heller (den talar om för tolkaren vilka sorters operationer som är tillåtna för objektet).

Men *värdet* av ett objekt kan i vissa fall ändras, i andra fall inte ändras, beroende på vilken typ objektet har.
Om värdet *kan* ändras säger vi att objektet är *föränderligt* (*mutable* eller *changeable* på engelska).
Om värdet *inte kan* ändras säger vi att objektet är *oföränderligt* (*immutable* eller *unchangeable* på engelska).

Testa följande:

```
Answer = 42
print(id(Answer))  # Vi matar ut ID för det objekt variabeln refererar till.
Answer = 43
print(id(Answer))  # Vi matar ut ID för det nya objekt variabeln refererar till.
```

Här ser vi att det som ändras när vi tilldelar ett nytt värde till variabeln i själva verket är att variabeln efter tilldelningen refererar till ett *nytt* objekt. Värdet `42` ligger kvar i minnet med samma ID som förut, men `Answer` refererar inte längre till detta värde utan till det nya objektet `43` som har ett annat ID än objektet `42`. Värdena `42` och `43` har typen `int`, och `int` är en *oföränderlig* typ.

Låt oss nu titta på en *föränderlig* typ, nämligen `list`.

```
x = [10, 20, 30]  # Definition av en variabel som refererar till ett list-objekt.
print(x)          # [10, 20, 30] matas ut på skärmen.
print(id(x))      # Variabelns ID matas ut.

x.pop()           # Vi tar bort sista värde i listan.
print(x)          # [10, 20] matas ut på skärmen.
print(id(x))      # Variabelns ID matas ut. Det är samma ID som förut.
```

Följande typer är *föränderliga*: `list`, `set`, `dict`.
Följande typer är *oföränderliga*: *numeriska typer* (`int`, `float`, `complex`, `Decimal`), *strängar* (`str`), `range` och `tuple`.
Nedan följer en liten översikt (det finns fler typer i språket, men de får inte plats i detta kompendium).

Typ	Föränderlig?	Ordnad?	Använd när …
`int`	Nej	Enskilt värde	Du vill arbeta med ***heltal***.
`float`	Nej	Enskilt värde	Du vill arbeta med ***decimaltal*** (lagrade som *floating point*).
`Decimal`	Nej	Enskilt värde	Du vill arbeta med ***decimaltal*** med hög precision.
`complex`	Nej	Enskilt värde	Du vill arbeta med ***komplexa tal***.
`bool`	Nej	Enskilt värde	Du vill spara ***sanningsvärden*** (`True` och `False`).
`str`	Nej	Ja	Du vill arbeta med ***text*** eller generellt en *ordnad* följd av tecken. Funktionen `input` returnerar ett `str`-objekt.
`list`	Ja	Ja	Du vill arbeta med en **ordnad sekvens** som ska kunna uppdateras. Elementen i en lista kan vara vilken typ som helst.
`tuple`	Nej	Ja	Du vill arbeta med en ***ordnad sekvens*** som du vet *inte* kommer att ändras *eller* som ska användas som värde i en `set` eller som nyckel i en `dict`. Elementen i en `tuple` kan vara vilken typ som helst.
`range`	Nej	Ja	Du behöver en ***ordnad och sorterad sekvens av heltal*** som du kan använda med `in`-operatorn i en `for`-sats. Ett `range`-objekt kan omvandlas till en `list`, `tuple` eller `set`.
`set`	Ja	Nej	Du vill hålla koll på ***värden där ordningsföljd ej spelar roll*** och du inte vill ha dubbletter. Samtliga element i en mängd måste vara *oföränderliga*.
`dict`	Ja	Ja (sedan version 3.6)	Du vill ha en datasamling innehållande en uppsättning ***par*** bestående av *nycklar* som var för sig är knutna till ett *värde*. Samtliga nycklar i ett `dict`-objekt måste vara *oföränderliga*.

Tabell 9. Översikt över några *föränderliga* respektive *oföränderliga* typer.

7. Funktioner

En *funktion* är i programmeringssammanhang nästan aldrig samma sak som en *funktion* i matematiska sammanhang. Inom matematiken är en funktion en teoretisk konstruktion, en *formel* som kan användas för att räkna ut värden, till exempel $f(x)=4x-2$ som varje nytt x ger oss ett nytt värde, nämligen $4x-2$.

I de flesta programspråk är en funktion i stället ett *delprogram* (kallas även *procedur*, *subrutin*, *rutin*, *subprogram*, *underprogram*, *service*, kärt barn har många namn).

Ett sådant delprogram är helt enkelt *några rader med kod* som lagras i minnet så de kan användas senare och användas flera gånger. Orsaken till at detta kallas en *funktion* är att vi kan *skicka värden* till delprogrammet, och vi kan *få ut ett värde* från funktionen (precis som i matematiken). Men i motsats till matematiken kan vi ha funktioner som *inte* tar emot några värden alls och/eller *inte* ger något värde som resultat.

7.1. Inbyggda funktioner

I Python finns det ett antal så kallade *inbyggda funktioner* (*built-in functions*). Vi har redan använt flera av dessa (exempelvis `print`, `input`, `id`, `help` samt de så kallade *konstruktorerna* `int`, `float`, `str`, `range` med flera). För en komplett lista över alla inbyggda funktioner, se docs.python.org/3/library/functions.html.

7.2. Funktionsanrop

Att *använda* en funktion kallas att *anropa* funktionen och vi säger att vi utför ett *funktionsanrop* (*function call*). I Python (och långt de flesta andra programspråk) görs detta med *parenteser*, alltså `(` och `)`, inom vilka man skriver de värden funktionen ska ta emot:

```
abs(-3.2)     # Funktionen abs anropas och värdet -3.2 skickas med till funktionen.
id(5)         # Funktionen id anropas och värdet 5 skickas till funktionen.
help()        # Funktionen help anropas och inga värden skickas till funktionen.
print("A",2)  # Funktionen print anropas och värdena "A" och 2 skickas till funktionen.
```

De värden vi skickar till en funktion kallas *aktuella parametrar* (eller bara *parametrar*, och ibland *argument*).

Det värde vi får tillbaka kallas funktionens *returvärde* (*return value*). Vissa funktioner har inget returvärde, och då returnerar funktionen i stället `None` (som har typen `NoneType`). I exemplen ovan returnerar `help` och `print` uttrycket `None` medan `abs` returnerar `3.2` och `id` returnerar ett stort heltal.

Eftersom en funktion är ett delprogram kan i princip vad som helst ske inuti en funktion. Både `help` och `print` matar exempelvis ut text på skärmen, och `input` tar emot en text från tangentbordet. Det finns funktioner som öppnar filer på hårddisken, skickar e-post och mycket annat. Långt de flesta funktioner är inte inbyggda utan ligger i *moduler* som måste *importeras* innan vi kan anropa dem (se Avsnitt 8).

Om man vill veta vad en funktion gör, så använder man `help`, till exempel `help(abs)`. Testa gärna detta.

Så snart ett funktionsanrop har utförts fortsätter programmet med den sats som står efter anropet.

7.3. Att skapa egna funktioner

Innan man kan anropa en funktion måste den *definieras*. De inbyggda funktionerna *fördefinierade* och kan anropas direkt. För funktionerna i *moduler* (exempelvis de moduler som finns i *standardbiblioteket*) gäller att den modul som innehåller funktionen måste importeras innan en funktion kan anropas.

7.3.1. Funktionsdefinitioner

Vi kan definiera en *egen* funktion med en så kallad *funktionsdefinition*. Så här ser det ut:

```
def funktionsnamn(parametrar):
    satser
```

Definitionen börjar med kodordet `def` efterföljt av funktionens namn (kallad `funktionsnamn` ovan) som måste följa samma namngivningsregler som gäller för variabler. Sedan följer en *parentes* inom vilken de så kallade *formella parametrarna* anges (`parametrar` ovan). Dessa parametrar är variabelnamn separerade

med *komma* (se exempel nedan). Efter parentesen sätter vi ett *kolon*. Därefter *ny rad* och slutligen de satser som funktionen ska utföra när den anropas. Satserna i funktionen måste *indenteras* så tolkaren vet att de tillhör funktionen. Vi tar ett exempel:

```
# Definition av funktionen Fahrenheit_till_Celsius
def Fahrenheit_till_Celsius(F):
    C = (F - 32) * 5 / 9
    print("Grader Celsius: ", C)

# Anrop av funktionen Fahrenheit_till_Celsius
Fahrenheit_till_Celsius(100)
```

Här har vi skapat en liten funktion utifrån vårt tidigare exempel. Nu kan vi anropa `Celsius_till_Fahrenheit` om och om igen och även använda funktionen i andra program (när den har placerats i en modul, se Avsnitt 8).

7.3.2. Satsen **`return`**

Med en `return`-sats avslutas funktionen direkt och det värde som står efter kodordet `return` skickas tillbaka till det stället där funktionen anropades. Låt oss använda detta i vår temperaturomvandlingsfunktion:

```
# Definition av funktionen Fahrenheit_till_Celsius
def Fahrenheit_till_Celsius(F):
    return (F - 32) * 5 / 9

# Här anropas funktionen inuti en while-snurra
while True:
    F = input("Grader Fahrenheit (avsluta med Q):  ")
    if F=="Q" or F=="q":
        break
    C = Fahrenheit_till_Celsius(float(F))
    print("Grader Celsius: ", C)
```

Låt oss skriva om programmet med serienummer från Avsnitt 5. Så här såg det ut:

```
while(True):
    serienummer = input("Serienummer (avsluta med Q): ")
    if serienummer == "Q" or serienummer == "q":
        break
    if serienummer[0] == "A":
        print("Detta är A-serien.")
        print("Modellen finns i sortimentet")
    elif serienummer[0] == "B":
        print("Detta är B-serien.")
        print("Modellen finns i sortimentet")
    elif serienummer[0] == "C":
        print("Detta är C-serien.")
        print("Modellen finns i sortimentet")
    elif serienummer[0] == "D":
        print("Detta är D-serien.")
        print("Modellen finns i sortimentet")
    else:
        print("Ogiltigt serienummer.")
        print("Modellen finns inte i sortimentet")
print("Programmet avslutas.")
```

Med en funktion får vi lite bättre struktur i programmet:

```
def kontrollera_serienummer(nr):
    if nr[0]>="A" and nr[0]<="D":
        print("Detta är " + nr[0] + "-serien.")
        return True
    print("Ogiltigt serienummer.")
    return False

while(True):
    serienummer = input("Serienummer (avsluta med Q): ")
```

```
    if serienummer=="Q" or serienummer == "q":
        break
    if kontrollera_serienummer(serienummer):
        print("Modellen finns i sortimentet")
    else:
        print("Modellen finns inte i sortimentet")
print("Programmet avslutas.")
```

7.4. Lokala variabler och globala variabler

De *formella parametrarna* i en funktionsdefinition samt övriga variabler vi definierar *inne i* funktionen blir så kallade *lokala variabler*. De är endast åtkomliga inuti funktionen.

Variabler som definieras *utanför* funktioner kallas globala variabler. De kan avläsas (men inte ändras) inuti funktioner. Vill vi göra en global variabel *ändringsbar* inuti en funktion använder vi kodordet `global` (till höger):

```
a=5

def f():
    a=3
    print(a) # Värdet 3 skrivs ut

f()
print(a) # Värdet 5 skrivs ut
```

```
a=5

def f():
    global a
    a=3
    print(a) # Värdet 3 skrivs ut

f()
print(a) # Värdet 3 skrivs ut
```

7.5. Dokumentsträngar i funktioner

I professionella sammanhang skrivs funktioner på ett sätt så de kan ingå i färdiga moduler som används precis som alla andra moduler. Men då vill vi naturligtvis att funktionen `help` även ska kunna ge information om våra egna funktioner. Detta gör med hjälp av en så kallad *dokumentsträng* (*docstring*).

Dokumentsträngar kan vara ganska komplicerade. För en fullständig genomgång, se python.org/dev/peps/pep-0257. Nedan ges en liten introduktion.

En dokumentsträng används för att lägga in text i en funktion (eller i en *modul*) som kan läsas (och matas ut) av funktionen `help`. En dokumentsträng är en textsträng inom trippelcitattecken (`'''`). Sådana strängar ignoreras av tolkaren om de står för sig själva.

Lägger man in en sådan överst i en funktion kan man få information om funktionen med `help`. Låt oss göra detta för funktionen från förra avsnittet:

```
def kontrollera_serienummer(nr):
    '''Kontroll av om ett serienummer är giltigt eller ej.'''
    if nr[0]>="A" and nr[0]<="D":
        print("Detta är " + nr[0] + "-serien.")
        return True
    print("Ogiltigt serienummer.")
    return False
```

Nu kan vi skriva `help(kontrollera_serienummer)` och får då följande utmatning:

```
Help on function kontrollera_serienummer in module __main__:
kontrollera_serienummer(nr)
    Kontroll av om ett serienummer õr giltigt.
```

Här ser vi att vårt svenska tecken `ä` tolkas helt fel av funktionen `help` (i det här fallet blir det tecknet `õ`). Detta är ett bra argument för att *alltid* använda *engelska* som språk när man skriver program. Alltså *engelska variabelnamn*, *engelska funktionsnamn* och fram för allt *engelska dokumentsträngar*.

Det finns mycket mer att säga om detta. Det har skrivits hela böcker om de olika konventioner och regler som finns för att namnge de så kallade *identifierare* (variabler, klasser, funktioner med mera) vi skapar i Python (och andra programspråk).

I utmatningen vi får med `help(kontrollera_serienummer)` ovan nämns en modul med namnet `__main__`. I nästa avsnitt ges en förklaring till detta.

8. Moduler

Python är ett *stort* språk (och det växer ständigt). Så ifall vårt system skulle behöva läsa in *hela* språket varje gång vi skulle köra ett program skulle det bli oerhört ineffektivt eftersom vi oftast bara behöver tillgång till en liten avgränsad del av språket. Lösningen på detta problem kallas *moduler*. I de senaste utgåvorna av Python finns över två hundra moduler som medföljer vid installationen.

En *modul* är en mängd kod som först blir tillgänglig för oss när vi importerar den. Ofta innehåller moduler en mängd funktioner (som i sin tur först körs när vi anropar dem), men en modul kan även innehålla definitioner av *typer* samt alla möjliga sorters kod. Vi har till exempel sett modulen `math` som innehåller de vanligaste matematiska funktionerna, och modulen `decimal` som ger oss tillgång till typen `Decimal` (se avsnitt 3.5).

8.1. Kodorden **import**, **from** och **as**

En modul importeras med kodordet `import`. Detta kan göras på flera olika sätt. Vi visar med några exempel:

```
import math                          # Importerar modulen math
```

Användning av modulens funktioner görs med `math.`*`funktionsnamn`*, t.ex. `math.sin(7)` vilket ger oss sinus av 7.

```
from math import cos                 # Importerar endast funktionen cos från modulen math
```

För att använda funktionen `cos` kan vi nu skriva enbart `cos`, exempelvis `cos(6)` vilket ger oss cosinus av 6.

```
from math import sqrt as roten_ur # Importerar funktionen sqrt och döper den till roten_ur
```

För att använda funktionen `sqrt` kan vi nu skriva `roten_ur`, exempelvis `roten_ur(2)` vilket ger oss $\sqrt{2}$.

```
import decimal                       # Importerar modulen decimal
```

För att använda typen `Decimal` kan vi nu skriva `decimal.Decimal`.

```
from decimal import *                # Importerar allt från modulen decimal
```

För att använda typen `Decimal` kan vi nu skriva `Decimal` (i stället för `decimal.Decimal`).

När man har importerat en viss modul kan man få hjälp med `help`-funktionen:

```
import tkinter   # Importerar modulen tkinter
help(tkinter)    # Visar hjälpinformationen för tkinter
```

8.2. Moduler i standardbiblioteket

Nedan ses ett urval från standardbiblioteket (för en fullständig lista, se docs.python.org/3/py-modindex.html).

Modul	Kort beskrivning av modulens innehåll
`array`	Typen `array` (nästan samma som `list`, fast alla element måste ha samma typ).
`atexit`	Verktyg för att registrera funktioner som ska köras automatiskt när tolkaren avslutas.
`audioop`	Verktyg för att editera och utföra operationer på ljudfiler.
`calendar`	Verktyg för hantering av kalendrar.
`cmath`	Matematiska funktioner för komplexa tal, till exempel konvertering till polära koordinater.
`colorsys`	Omvandling mellan färgsystemen RGB, YIQ, HLS och HSV.
`datetime`	Olika typer för datum och tidpunkter.
`decimal`	Typen `Decimal` (se Avsnitt 3.5).
`doctest`	Hjälpverktyg för testkörning av *exempelkod* i dokumentsträngar.
`email`	Verktyg för att skapa och hantera e-mejl.
`filecmp`	Verktyg för att jämföra filer och mappar.
`fileinput`	Verktyg för att enkelt genomlöpa filer (eller *standard input*).
`fractions`	Typen `Fraction` som används för *bråk* (rationella tal).
`ftplib`	Typen `FTP` som kan användas för att bygga klientsidan av en FTP-applikation.
`getpass`	Verktyg för enkel hantering av lösenord och användarnamn.
`html`	Verktyg (med undermodulerna `html.parser` och `html.entities`) för hantering av HTML.
`http`	Paket med undermodulerna `http.client`, `http.server`, `http.cookies`, `http.cookiejar`.
`io`	Verktyg för hantering av *strömmar* (*text*, *binary*, *raw*). Se Avsnitt 9.
`itertools`	Mycket användbar modul med så kallade *iteratorer* för effektiv genomlöpning av data.
`json`	Verktyg för kodning och avkodning av JSON-strängar.

`keyword`	Verktyg för att avgöra om textsträngar är reserverade kodord eller ej.
`locale`	Verktyg för hantering av internationaliseringstjänster (decimaltecken, valutatecken m.m.).
`math`	Grundläggande matematikfunktioner.
`operator`	Operatorer som funktioner (exempelvis gör `operator.add(a,b)` samma som `a+b`).
`os`	Plattformsberoende (extra) verktyg för interaktion med operativsystemet.
`pdb`	Interaktiv källkodsfelsökare.
`platform`	Ger tillgång till information om den aktuella plattformen.
`random`	Verktyg för generering av *slumptal*.
`shelve`	Typen `shelve`. Se mer och fler liknande typer på docs.python.org/3/library/persistence.html
`statistics`	Matematiska statistikfunktioner.
`string`	Ett antal funktioner för hantering av textsträngar.
`sysconfig`	Information om Pythons konfigurering.
`time`	Funktioner för hantering av tid. Se även modulerna `datetime` och `calendar`.
`timeit`	Funktioner för att mäta hastigheten av ett kodfragment i Python.
`tkinter`	Förkortning av *Tk Interface*. Modul för att bygga grafiska användargränssnitt.
`turtle`	Ett ramverk för att bygga grafiska applikationer (i utbildningssyfte)
`unicodedata`	Verktyg för UCD (Unicode Character Database) och hantering av unicode-text.
`webbrowser`	Verktyg för att visa webbaserat innehåll.
`winreg`	Ger åtkomst till registerdatabasen i Windows.
`winsound`	Ger åtkomst till ljudsystemet i Windows (spela ljudfiler m.m.).
`wsgiref`	Verktyg för WSGI (Web Server Gateway Interface).
`xml`	Verktyg för bearbetning av data i XML-format.
`zipfile`	Verktyg för att skapa, öppna, läsa och redigera *.zip*-filer.

Tabell 10. Ett urval av Pythons standardmoduler.

8.3. Andra moduler

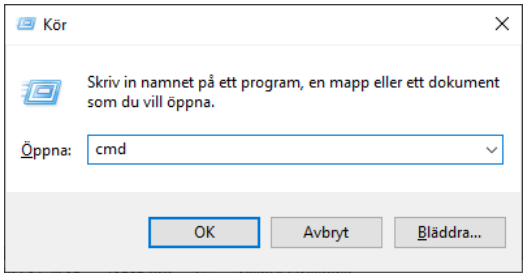
Förutom modulerna i standardbiblioteket finns det hundratals andra moduler som kan användas inom olika områden. Det finns moduler för ***grafiska gränssnitt*** (exempelvis *wxPython*, *PyGObject*, *Pmw*, *WCK*, *Tix*), ***databashantering*** (exempelvis *MySQLdb*, *PyGreSQL*, *Gadfly*, *SQLAlchemy*, *KInterbasDB*), ***webbutveckling*** (exempelvis *Beautiful Soup*, *scrape*, *mechanize*, *libgmail*, *Google Maps*, *Requests*, *Selenium*, *pyquery*), ***ljudbehandling*** (exempelvis *pySonic*, *PyMedia*, *PMIDI*, *Mutagen*), ***bild- och videobehandling*** (exempelvis *PIL (Python Imaging Library)*, *GDmodule*, *VideoCapture*, *MoviePy*, *pyscreenshot*), ***matematik och datavetenskap*** (exempelvis *SciPy*, *Matplotlib*, *Pandas*, *NumPy*, *SymPy*), ***spelutveckling*** (exempelvis *Pygame*, *Pyglet*, *pyOpenGL*), ***USB-portar och seriella portar*** (exempelvis *PyUSB*, *PySerial*, *USPP*) samt ***Windowsspecifika moduler*** (exempelvis *pywin32*, *PyRTF*, *WMI*, *Py2exe*) och ***MacOS-specifika moduler*** (exempelvis *py2app*, *PyObjC*) och *mycket* annat.

De flesta moduler finns i så kallade *paket* (som kan innehålla *undermoduler*). För att få tillgång till sådana paket måste vi *installera* dem, se nästa avsnitt.

8.3.1. Installation av modulen **`matplotlib`** med *`pip`*

Det finns flera olika sätt att installera paket (se webbplatsen för varje respektive paket), men det enklaste är att använda Pythonprogrammet `pip`. Nedan visas hur man gör i Windows.

1. Tryck ner Windowstangenten ⊞ tillsammans med tangenten **R** och skriv **cmd** i det fönstret som då öppnas:



Figur 6. Fönstret *Kör* (*Run*) i Windows 10.

2. I kommandotolken som öppnas skriv följande kommando:
python -mpip install -U pip

3. När kommandot ovan har slutförts är vi redo att installera moduler. Nedan installerar vi modulen `matplotlib`:
pip install matplotlib (om detta inte fungerar, testa **python -mpip install -U matplotlib**)

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.19042.1052]
(c) Microsoft Corporation. Med ensamrätt.

C:\Users\simon>python -mpip install -U pip
Requirement already satisfied: pip in c:\users\simon\appdata\local\programs\python\python39\lib\site-packages (21.1.1)
Collecting pip
  Downloading pip-21.1.2-py3-none-any.whl (1.5 MB)
     |████████████████████████████████| 1.5 MB 3.2 MB/s
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 21.1.1
    Uninstalling pip-21.1.1:
      Successfully uninstalled pip-21.1.1
Successfully installed pip-21.1.2

C:\Users\simon>python -mpip install -U matplotlib
Collecting matplotlib
  Downloading matplotlib-3.4.2-cp39-cp39-win_amd64.whl (7.1 MB)
     |████████████████████████████████| 7.1 MB 3.3 MB/s
Collecting numpy>=1.16
  Downloading numpy-1.20.3-cp39-cp39-win_amd64.whl (13.7 MB)
     |████████████████████████████████| 13.7 MB 6.8 MB/s
Collecting pyparsing>=2.2.1
  Downloading pyparsing-2.4.7-py2.py3-none-any.whl (67 kB)
     |████████████████████████████████| 67 kB 742 kB/s
Collecting cycler>=0.10
  Downloading cycler-0.10.0-py2.py3-none-any.whl (6.5 kB)
Collecting python-dateutil>=2.7
  Downloading python_dateutil-2.8.1-py2.py3-none-any.whl (227 kB)
     |████████████████████████████████| 227 kB 2.2 MB/s
Collecting kiwisolver>=1.0.1
  Downloading kiwisolver-1.3.1-cp39-cp39-win_amd64.whl (51 kB)
     |████████████████████████████████| 51 kB 199 kB/s
Collecting pillow>=6.2.0
  Downloading Pillow-8.2.0-cp39-cp39-win_amd64.whl (2.2 MB)
     |████████████████████████████████| 2.2 MB 3.2 MB/s
Collecting six
  Downloading six-1.16.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: six, python-dateutil, pyparsing, pillow, numpy, kiwisolver, cycler, matplotlib
Successfully installed cycler-0.10.0 kiwisolver-1.3.1 matplotlib-3.4.2 numpy-1.20.3 pillow-8.2.0 pyparsing-2.4.7 python
-dateutil-2.8.1 six-1.16.0
```

Figur 7. Installation av `pip` och `matplotlib` i Windows 10 (observera att Python måste finnas på datorn innan detta görs).

Programmet `pip` behöver bara installeras en gång. Därefter kan valfritt antal moduler installeras och när det är gjort kan de användas i flera olika program utan att man behöver installera dem varje gång.

8.4. Ett exempel med modulen **`matplotlib`**

När en modul har installerats kan *VS Code* öppnas och då finns modulen tillgänglig (likaså i *IDLE*). Modulen `matplotlib` används till exempel för att skapa (plotta) matematiska grafer (se matplotlib.org). Då används undermodulen `pyplot` (som vanligtvis importeras som `plt`).

Som exempel, låt oss plotta en graf med alla *kvadrattalen* (1^2, 2^2, 3^2, 4^2, 5^2 o.s.v.) upp till 100^2.

```
from matplotlib import pyplot as plt

x = list(range(1,101))          # En lista med värdena 1 till 100.
y = []                          # En tom lista.
for i in range(len(x)):
    y.append(x[i]*x[i])         # Kvadrattalen läggs in i listan y.
plt.plot(x, y, 'rs')            # Skapa graf med värdena i y som funktion av värdena i x.
plt.xlabel("Heltalsvärden (x)") # Text på x-axeln
plt.ylabel("Kvadrattal (y)")    # Text på y-axeln.
plt.show()                      # Visa grafen i ett separat fönster.
```

För att få information om funktionen `plot` (som finns i `plt`), skriv `help(plt.plot)`. Då ser vi att strängen `'rs'` som används är en formatsträng som talar om för funtionen att vi vill ha *red* (`'r'`) *square* (`'s'`) som grafpunkter.

När programmet ovan körs får vi en graf som vi kan zooma i, och även spara som en *.png*-fil på hårddisken.

8.5. Skriva egna moduler

Kortfattat kan man säga att en *modul* är en *.py*-fil. Om vi skapar en *.py*-fil har vi skapat en modul, och denna kan *importeras* i andra *.py*-filer (eller i Python Shell).

Låt oss anta att vi har sparat plot-exemplet från förra avsnittet i en fil med namnet *kvadrattal.py*. Då kan vi använda samma kod i ett annat program:

```
import kvadrattal
```

Om vi kör ovanstående får vi samma graf som vi fick i exemplet tidigare. Men låt oss ändra i filen *kvadrattal.py* så den i stället innehåller en *funktion*:

```
from matplotlib import pyplot as plt

def kvadrattalsgraf(n):
    '''This function plots n number of squares (x²), from 1 to n²'''
    x = list(range(1, n+1))        # En lista med värdena 1 till n.
    y = [] # En tom lista.
    for i in range(n):
       y.append(x[i]*x[i])         # Kvadrattalen läggs in i listan y.
    plt.plot(x, y, 'rs')           # Skapa graf med listan y som funktion av listan x.
    plt.xlabel("Heltalsvärden (x)") # Text på x-axeln
    plt.xlabel("Kvadrattal (y)")   # Text på y-axeln.
    plt.show()                     # Visa grafen i ett separat fönster.
```

Nu kan vi skapa en *ny* fil (eller flera nya filer) och använda funktionen där:

```
from kvadrattal import kvadrattalsgraf as graf

antal = int(input("Hur många tal? "))
graf(antal)
```

Sammanfattningsvis kan man säga att om vi ska använda en modul importerar vi den med `import` (eventuellt i kombination med `from` och `as`). Om det är en *modul i standardbiblioteket* fungerar det direkt. Om det är en *modul i ett paket* måste paketet installeras först. Om det är en *egen modul* måste den ligga i en *.py*-fil och denna fil måste ligga i den mapp där vi ska använda den.

8.5.1. Variabeln **`__name__`** (och värdet **`"__main__"`**)

Vi har tidigare sett att Python innehåller vissa inbyggda *funktioner*, och vi har även använt inbyggda *konstanter* (såsom `True`, `False` och `None`).

Det finns även inbyggda *variabler*. En av dessa heter `__name__` och den innehåller alltid namnet på den *importerade* modul som koden tillhör. Om vi importerar en modul från filen *test.py* så har `__name__` värdet `"test"` när koden vi har importerat körs. Om vi sedan importerar ytterligare en modul, så byts värdet i `__name__` ut mot namnet på denna nya modul (när koden i *den* körs).

Men om vi kör koden i filen *test.py* direkt, utan importering, då får `__name__` värdet `"__main__"`, och detta gäller för alla filer.

Kort sagt, när vi kör kod gäller `__name__ == "__main__"`, med mindre koden körs genom importering i en annan fil.

Om vi lägger till nedanstående rader i slutet av filen *kvadrattal.py* från tidigare säkerställer vi att koden i denna fil *både* kan importeras som modul *och* köras direkt:

```
if __name__ == "__main__":
    antal = int(input("Hur många tal? "))
    kvadrattalsgraf(antal)
```

9. Läsa från filer och skriva till filer

Ett datorprogram (alltså ett *program*, en *applikation*, en *app*) behöver vanligtvis någon typ av *input* för att få *in* data som programmet ska behandla. Dessutom ger programmet i regel någon typ av *output*, alltså data som vi får *ut* när vi kör programmet. Hittills har vi uteslutande använt funktionen `input` för att få *in* data i våra program (`input` läser en textsträng från tangentbordet) och funktionen `print` för att få *ut* data (`print` skickar en text till skärmen).

Både tangentbord och skärm är ett exempel på *yttre enheter*, och det finns många olika sorters yttre enheter, varav några används för *indata* (till exempel tangentbord, mus, kamera, mikrofon, pekskärm, hårddisk, skanner samt sensorer av olika slag), några används för *utdata* (till exempel skärm, pekskärm, högtalare, hårddisk, skrivare) och några används för både *in- och utdata* (till exempel hårddisk, modem, pekskärm).

I detta avsnitt ska vi använda hårddisken och *skapa* filer, *skriva till* filer, *läsa från* filer samt *ta bort* filer.
För fler detaljer än vad som finns med här, se till exempel docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files eller pythonforbeginners.com/files/reading-and-writing-files-in-python.

9.1. Den inbyggda funktionen **open**

Funktionen `open` har två parametrar, ett *filnamn* och ett *filläge* (*file mode*). Filnamnet är en textsträng innehållande en sökväg som beskriver filens *plats* och filens *namn* (inklusive *filändelse*). Filläget är en textsträng som beskriver på vilket sätt filen ska öppnas samt vilken sorts fil det är (*text* eller *rå data*).

Filläget kan ha följande värden (om inget filläge anges används det förvalda värdet `"r"+"t"`, alltså `"rt"`):

`"r"` (*Read*) öppnar en fil för *läsning* (ger ett fel om filen inte finns). **Förvalt värde.**
`"a"` (*Append*) öppnar en fil för att *lägga till innehåll* i den (skapar filen om den inte finns).
`"w"` (*Write*) öppnar en fil för att *skriva över innehållet* i den (skapar filen om den inte finns).
`"x"` (Create) skapar en fil (ger ett fel om filen redan finns).

Som tillägg (men inte *ensamt*) kan man specificera vilken typ av fil det rör sig om:

`"t"` (*Text*) betyder att filen innehåller text. **Förvalt värde.**
`"b"` (*Binary*) - betyder att filen innehåller övrig data (till exempel en bild).

Som returvärde från funktionen `open` får man ett *filobjekt*, även kallad *filhanterare* (*file handler* på engelska), som innehåller metoder för att läsa och skriva i filen. I exemplet nedan skapas tre filobjekt:

```
f1 = open("D:/OneDrive/Dokument/graf.png","rb")
f2 = open("Test.txt","at")
f3 = open("H:/test_sample.dat")
```

`f1` blir ett filobjekt för läsning av bildfilen *graf.png* i mappen *D:/OneDrive/Dokument*.
`f2` blir ett filobjekt som kan användas för att lägga till text i filen *Test.txt* (ligger i mappen i vilken Python körs).
`f3` blir ett filobjekt för läsning av textfilen *test_sample.dat* på roten av hårddisk *H*.

9.2. Metoderna **read** och **readline**

Funktionen `read` är en av metoderna i den filhanterare vi får från funktionen `open` ovan. Den läser från filen och returnerar en textsträng med innehållet (om det är en textfil). Vi kan välja att läsa enbart de första tecknen i filen, se exemplet nedan.

```
print(f2.read())     # Skriver ut hela innehållet i textfilen Test.txt från exemplet ovan.
print(f3.read(12))   # Skriver ut de första 12 tecken i textfilen test_sample.dat ovan.
```

Med metoden `readline` kan vi läsa *en* rad i taget. Nedan skrivs de två första raderna i *demo.txt* ut.

```
f = open("demo.txt", "r")
print(f.readline())
print(f.readline())
```

Man kan även använda en `for`-sats (filhanteraren kan betraktas som en genomlöpningsbar typ:

```
for a in f:
    print (a)
```

9.3. Metoderna **write** och **close**

För att kunna *skriva* till en fil måste den öppnas i filläget `"w"` eller `"a"` med funktionen `open`. När det har gjorts kan vi skriva till filen med funktionen `write`:

```
f = open("test.txt", "w")
f.write("Detta skriver över allt eventuellt tidigare text i filen!")
```

När vi är klara med att skriva (eller läsa) till en fil måste filen stängas. Detta görs med metoden `close`:

```
f.close()
```

Om vi vill *lägga till mer text* (filläget `"a"`) i filen *test.txt* ovan eller om vi vill *läsa text* (filläget `"r"`) från den måste den öppnas på nytt i rätt filläge, och även stängas på nytt:

```
f = open("test.txt", "r")
file_content = f.read()
f.close()
if file_content.find("tidigare") != -1:
    print('Filen innehåller ordet "tidigare"')
```

9.4. Modulen **os**

I modulen `os` finns verktyg för att kommunicera direkt med operativsystemet (till exempel hantera filer). Nedan visas hur vi kan ta bort filen *test.txt* ovan med funktionen `remove` från modulen `os` (först kontrolleras att det är rätt sökväg):

```
import os

if os.path.exists("test.txt"):
    os.remove("test.txt")
else:
    print("Filen existerar inte!")
```

Om vi vill ta bort en hel mapp använder vi funktionen `rmdir` (en förkortning av *remove directory*). Nedan tas mappen *Uppgifter* bort:

```
import os
os.rmdir("Uppgifter")
```

På samma sätt kan vi skapa en ny mapp med funktionen `mkdir` (en förkortning av *make directory*). Nedan skapas mappen *Uppgifter* på nytt:

```
import os
os.mkdir("Uppgifter")
```

Funktionen `os.path.exists` kan användas för att kontrollera om en mapp (eller en fil) existerar *innan* vi försöker ändra på den. Funktionen returnerar `True` eller `False`.

10. Tillämpningar

I detta kompendium har vi fokuserat på att gå igenom grunderna i språket Python. Men för att kunna jobba professionellt krävs kännedom om de situationer i vilka Python används. Det finns hundratals professionella system i vilka språket Python kan användas (varav en del kostar en rejäl slant medan andra är gratis).

Se gärna python.org/about/apps som startpunkt för vidare utforskning av ämnet.

10.1. Förslag på vidare studier

Nedan nämnas ett par system för industriella applikationer som inspiration och för vidare självstudier.

10.1.1. Grafana

Grafana (grafana.com) är ett system för att skapa och visualisera data genom så kallade *dashboards* (en sorts virtuella instrumentbrädor). Grafana är praktisk för att det kan hantera data från många olika källor samtidigt och på ett enkelt sätt presentera informationen. Webbplatsen innehåller bland annat kodexempel i Python.

10.1.2. Node-RED

Node-RED (nodered.org) är ett webbläsarbaserat onlinesystem som används för att koppla ihop extern utrustning, alltså hårdvara, och skapa så kallade *flöden* (*flows*) för att få kontroll över en given process.
Det primära språket här är JavaScript, men med ett litet tillägg (se flows.nodered.org/node/node-red-contrib-python-function) blir det möjligt att använda Python i stället. På webbplatsen finns tydlig information om hur Node-RED installeras (det kräver *node.js*, se nodejs.org/en) samt länkar till några utmärkta introduktionsvideor.

10.1.3. Cisco Packet Tracer

Cisco Packet Tracer (netacad.com/courses/packet-tracer) är ett simuleringsverktyg för IoT och nätverksteknologi som till stor del används i undervisningssammanhang (det är gratis att registrera sig och ladda ner verktyget). Det erbjuds en del onlinekurser (där själva kursintyget dock kostar några hundralappar).

10.2. Generellt om industriella tillämpningar

Enligt en rapport från 2019 gjort av ActiveState (activestate.com) är nedanstående de tio mest vanliga *industriella* områden där Python används (citat från rapporten):

Insurance: Creating business insights with machine learning.
Retail Banking: Flexible data transformation and manipulation.
Aerospace: Meeting software system deadlines.
Finance: Data mining to identify cross-sell opportunities.
Business Services: API access to financial information.
Hardware: Automating network administration.
Healthcare: Predicting disease prognosis.
Consulting Services: Bespoke web development.
InfoTech: IT modernization.
Software: Adding extensibility to legacy applications.

Inom *varenda* av dessa branscher (och många fler) finns ett antal professionella verktyg utvecklade för att så specifikt som möjligt uppfylla de krav som finns. Så när man som programmerare ställs in för ett problem som ska lösas består arbetet till stor del i att lära sig ett nytt system, och därefter skriva några rader med kod.

En annan väg att gå är naturligtvis att bli *Pythonexpert* och sedan jobba med utveckling av nya moduler (paket) vilka i sin tur används av tusentals andra utvecklare. Men låt oss nu inte gå händelserna i förväg …

I Avsnitt 11 finns ett antal övningar som bör göras för att träna på de grundläggande begreppen som har presenterats i detta kompendium. Lycka till!

11. Övningar

I detta avsnitt finns ett antal övningar till de föregående avsnitten.

11.1. Övningar till Avsnitt 1

1. Förklara kortfattat vad en *tolkare* är.

2. Installera den senaste versionen av Python från www.python.org/downloads på ditt system.
Öppna sedan *Python Shell* samt *IDLE* och bekanta dig med dessa program.
Installera därefter *Visual Studio Code* och eventuella tillägg som behövs för att kunna köra Python och bekanta dig med denna utvecklingsmiljö. Skapa ett par *.py*-filer och se till att du förstår hur de sparas, vart de hamnar samt hur de körs och felsöks.

3. Vad är periodtiden T för den harmoniska rörelsen av massan 2,2 kg i en fjäder med k=2,4 N/m, enligt formeln i Avsnitt 1.3?

4. Skriv en lista med så många *utvecklingsmiljöer* du kan hitta (ange *plattformen* inom parentes).

5. Skriv en lista med så många *Pythonvarianter* du kan hitta.

11.2. Övningar till Avsnitt 2

6. Är `_,a=0.3,5` giltig kod? Om så är fallet, vad gör koden? Om inte, *varför*?

7. Är `id(3)` samma som `id(a)` ifall `a` är en variabel av typen `float` med värdet `3.0`?

11.3. Övningar till Avsnitt 3

8. Förklara kortfattat begreppen *uttryck*, *literal* och *typ*.

9. Hur många *bitar* behövs för att lagra heltalet `12334534534534534534587889345476764 53`?

10. Skapa en `range`-sekvens med innehållet `13,17,21,25,29,33`.

11. Kan steglängden i ett `range`-objekt vara `0.5`?

12. Vad är skillnaden mellan `float` och `Decimal`?

13. Vad blir resultatet av `"ABCDEFGHIJ".find("def",2,8)`? Vilken *typ* har resultatet?

14. Skriv ett litet program som kollar om ett inmatat tal är ett *kvadrattal* (alltså ett tal x så att $\sqrt{x}$ är ett *heltal*). Det finns flera olika sätt att lösa det, men om `sqrt` ska användas så glöm inte att inkludera modulen `math`.

11.4. Övningar till Avsnitt 4

15. Vad blir resultatet av (`"ABCDEFGHIJ"*2).find("DEF",2,8)`? Vilken *typ* har resultatet?

16. Förklara vad *heltalsdivision* är. Vilken operator används för detta?

17. Om vi skriver `a = 5 or 3`, vilket *värde* och vilken *typ* får variabeln `a` då?

18. Skapa variabel som innehåller en sekvens med alla *udda tal* från `1` till `999999`.

11.5. Övning till Avsnitt 5

19. Skriv ett program som läser in en textsträng på formen `"Series sss Measurement mmm Value vvv"` där ***sss***, ***mmm*** och ***vvv*** är *decimala* värden med valfritt antal decimaler. Programmet ska extrahera dessa tre värden, placera dem i var sin `Decimal`-variabel och slutligen skriva ut dem var för sig.

11.6. Övningar till Avsnitt 6

20. Skapa en tom *lista* med namnet `x`.
Lägg in värdena `-4` och `5.6` som element i listan med metoden `append`.
Lägg till en *tom lista* som element i listan `x`. Återigen, använd `append`.
Listan `x` har nu tre element. Sista elementet är en tom lista. Lägg till värdet `12` i sista elementet.
Mata ut värdet i `x` på skärmen. Fick du resultatet `[4, 5.6, [12]]`?

21. Skapa en *mängd* med alla elementen från din sekvens i Uppgift 18. Fråga efter ett tal. Om det inmatade talet finns med i din mängd, så ska det tas bort ur mängden. Om det inte finns med, ska det läggas till. Använd operatorn `in` samt `set`-metoderna `add` och `remove`.

22. Bygg vidare på Uppgift 21, så du får en snurra som gång på gång frågar efter ett nytt värde. Om nya värdet inte finns med i mängden läggs det till, annars tas det bort. Efter tio inmatningar ska den aktuella mängden jämföras med den ursprungliga mängden. Alla nytillkomna tal läggs i en lista med namnet `tillkomna`. Alla tal som försvunnit läggs i en lista med namnet `borttagna`. Innehållet i de två listorna `tillkomna` och `borttagna` skrivs sedan ut på skärmen.
Ett råd: Använd metoden `copy` för att skapa kopia av den ursprungliga mängden. Använd en `for`-snurra för att upprepa inmatningen. Använd mängddifferensoperatorn för att avgöra i vilken av mängderna `borttagna` och `tillkomna` de inmatade talen ska läggas.

23. Gör om föregående program så inmatningen inte stannar automatiskt efter tio inmatningar utan i stället stannar när användaren matar in ordet `STOP`.
Ett råd: Använd `while` i stället för `for`.

11.7. Övningar till Avsnitt 7

24. Skriv en funktion (innehållande en dokumentsträng), med namnet `make_sorted_str_from_3_int` som tar tre olika `int`-värden. Värdena sorteras sedan i storleksordning och sätts ihop till en textsträng (typen `str`) och returneras av funktionen. Exempel på ett anrop:

```
make_sorted_str_from_3_int(23,-2,10)   # Ger strängen "-21023"
```

25. Gör om programmet från Uppgift 19, inläsningen av texten och extraheringen av de tre talen ligger i en funktion. Låt programmet fråga hur många gånger funktionen ska köras.
Ett råd: Använd en `for`-snurra i vilken funktionen anropas.

11.8. Övningar till Avsnitt 8

26. Skriv en funktion som kallas `Create_Random_Values`. Funktionen ska skapa en *lista* med 1000 strängar på formen `"Series: s, Measurement: m, Value: v"` (*nästan* som i Uppgift 19), där ***s***, ***m*** och ***v*** är *slumpmässiga* heltal mellan `0` och `100` (dubbletter är därmed möjliga i detta fall). Listan ska returneras av funktionen. Spara funktionen i en fil med namnet `createlist.py`.
Ett råd: Importera standardmodulen `random`. Använd `str`-metoden `split`.

27. Skriv ett program där funktionen `Create_Random_Values` används (importera modulen `createlist` från Uppgift 26). Programmet ska extrahera samtliga 1000 serienummer (***s***), mätningsnummer (***m***) och värden (***v***) och placera dem i 1000 tuplar som läggs i en ny lista med namnet `result`. Därefter ska eventuella dubbletter (alltså tuplar som har både samma ***s*** och samma ***m***) tas bort (behåll tupeln med det *högsta* mätvärdet (***v***)). Sedan ska programmet fråga användaren efter en mätserie (ett heltal mellan `0` och `100`) och sedan plotta alla mätningarna i denna serie (om det finns några) i en graf där x-axeln är mätningsnummer och y-axeln är mätvärden.
Ett råd: Använd `str`-metoden `split` vid extraheringen. Använd `pyplot` för att plotta grafen.

11.9. Övningar till Avsnitt 9

28. Skriv en funktion som bygger på funktionen `Create_Random_Values` från Uppgift 26 (kalla den nya utgåvan `Create_Simulated_Measurements`). Funktionen ska fråga användaren hur många mätserier (`x` stycken) som ska genereras (låt serienumren ***s*** löpa från `1` till `x`, så dubbletter *inte* är möjliga). För varje mätserie ska `1000` mätningar simuleras (låt ***m*** löpa från `1` till `1000`). Varje *värde* ska vara ett decimaltal mellan `-1000.0` och `+1000.0`. Funktionen ska sedan spara den genererade datan en *textfil* med valfritt namn (fråga användaren). En mätning per rad.

29. Skriv ett program som används för att öppna filerna som skapas av funktionen `Create_Simulated_Measurements` från Uppgift 28. Programmet frågar efter ett filnamn, kollar om filen existerar, öppnar filen, läser in datan och omvandlar alla värdena (***v***) till `float` och mätningsnumren ***m*** till `int`. Sedan skriver programmet ut antalet mätserier i filen, frågar efter mätserienummer och plottar slutligen denna mätserie (***m***, ***y***) med `pyplot` (jmf. Uppgift 27).

30. Öppna en valfri *bildfil* för läsning med filläget `"rb"` och skriv ut filens innehåll. Blev resultatet förvånande?