DA2004/DA2005: Programmeringsteknik

Kompendium

Lars Arvestad, Christian Helanow, Anders Mörtberg, Kristoffer Sahlin

Innehåll

Kapitel 1

Kom igång med Python

1.1 Programmering och Python

Detta kompendium handlar om *programmering*, d.v.s. att skriva program som en dator sedan kan köra för att lösa olika typer av problem. I kompendiet använder vi programmeringsspråket *Python*, men många av de underliggande idéerna går att överföra till andra språk (Java, JavaScript, C/C++, Ruby, Scala, Haskell, OCaml…). Python är ett väldigt populärt programmeringsspråk (testa att Googla *"most popular programming languages"*) som kan användas att skriva mjukvara till i princip alla sorters tillämpningar. Det används även friskt på många av de största IT-företagen, som Instagram, Google, YouTube, Spotify, Amazon…

Vi kommer att använda version 3.x av Python. Vi rekommenderar även en installation av *Anaconda*, ett verktyg för att hantera paket och omgivningar för Python. Installationsinstruktioner finns på kurshemsidan samt genom en videotutorial som ligger på YouTube.

En *programmeringseditor* stödjer "syntax highlighting" och operationer som gör det lättare att programmera. En *IDE*, "Integrated Development Environment", stödjer stora programmeringsprojekt, men är svårare att använda. Vi kommer att använda oss av *Spyder* som följer med installationen av *Anaconda* i denna kurs.

Exempel på andra programmeringseditorer:

- PyCharm — stark integration med Python
- Atom.io — generell
- MS Visual Studio Code — gratis och idag mycket spridd
- För entusiaster (kan vara svåra att komma igång med):
  - Emacs
  - Vim

1.1.1 Att programmera i Python

När man programmerar i Python skriver man kod i en `.py` fil med hjälp av sin favoriteditor (vi använder som sagt Spyder). Denna kod laddas sedan i Python tolken (eng. *interpreter*).

Startar man Spyder-applikationen inifrån Anaconda Navigator får man ett fönster med tre paneler: 'editor', 'help', och 'console'. I kompendiet är kod som är skriven i editorn inramad med grå bakgrund. Vi kan till exempel skriva följande tre rader i Spyders editor:

```
print(17)
print(1.1)
print('DA2004')
```

Om vi trycker på den gröna triangeln uppe till vänster eller tangentbordskommando `F5` tolkas denna kod i Python tolken och skrivs ut i konsolen. Om man inte sparat filen så dyker det först upp en ruta med "Save file". Spara filen på valfritt ställe på hårddisken, men kom ihåg att ha bra struktur så att du hittar filerna senare. Ett smidigt sätt är att skapa mappen "DA2004" och ha alla lämpliga undermappar (t.ex. "labb 1") där man sparar filerna för varje del av kursen.

Vi döper filen till `f1.py` och när vi kör den får vi följande i konsolen på Mac OSX (det ser lite annorlunda ut på Linux och Windows):

```
runfile('/Users/kxs624/kurser/DA2004/f1.py',
    wdir='/Users/kxs624/kurser/DA2004/')
17
1.1
DA2004
```

Obs: i kompendiet är kod och resultat som skrivs ut i konsolen inramad med turkos bakgrund.

1.2 Stil i kompendiet

1.2.1 Kodrepresentation

Vi kommer att beskriva programmering med svenska termer, men kod skrivs på engelska. Att skriva kod på engelska är vedertagen praxis då man ofta jobbar i internationella miljöer. Sedan är även själva språket Python på engelska.

När det är underförstått att vi skriver kod i texteditorn som vi vill köra och skriva ut i konsolen kommer vi visa den grå och turkosa miljön efter varandra på följande sätt:

```
print("welcome to DA2004")
print(2*2)
```

```
welcome to DA2004
4
```

1.2.2 Uppgifter

När det gäller uppgifter i kompendiet kommer vi att markera uppgifter som vi anser vara svårare med symbolen † först i uppgiftstexten.

1.3 Kommentarer

Det är väldigt viktigt att dokumentera sin kod med hjälp av informativa kommentarer. Dessa skrivs efter # och används för att förklara vad specifika kodsnuttar gör. Det är god praxis att dokumentera sin kod med bra kommentarer så att någon annan kan läsa koden. Vi kan även intyga att om man kommer tillbaka till kod man skrivit för längesedan så kan det vara väldigt svårt att förstå hur man tänkte om det inte finns några kommenterar.

Exempel:

```
print('Det här skrivs ut') # kommentarer gör det inte
```

```
'Det här skrivs ut'
```

Man kan på så sätt lägga in kommentarer i kod för att förklara vad man gör. Till exempel kan vi kommentera våra rader ovan

```
print(17) # Skriver ut talet 17
print(1.1) # Skriver ut talet 1.1
print('DA2004') # Skriver ut kurskoden för denna kurs
```

1.4 Variabler

Variabler används för att lagra olika typer av värden.

```
x = 17
y = 'DA2004'
z = 22/7
q = 22//7

print(x)
print(y)
print(z)
print(a)
print(x,y,z,a)
```

```
17
'DA2004'
3.14
3
17 DA2004 3.142857142857143 3
```

Notera att vi måste kalla på `print` för att variablerna ska skrivas ut till konsolen.

I exemplet ovan har vi använt följande typer av värden:

- heltal (eng. `int`)
- flyttal (eng. `float`)
- strängar (eng. `strings`)

Python har fler primitiva typer vilka vi kommer återkomma till.

Man kan även tilldela flera variabler på en gång:

```
x,y = 2,3
print(x)
print(y)
```

```
2
3
```

Notera att om `x` och `y` redan har definierats ovan i koden (t.ex. som `17` och `'DA2004'` tidigare) så kommer de att skrivas över med värdena `2` och `3`.

1.5 Stränghantering och `print`

Strängar kan skrivas inom antingen enkelfnuttar `'` eller dubbelfnuttar `"`. Man kan på så sätt enkelt blanda `'` och `"`:

```
print("hej")
print('hej')
print("h'e'j")
print('h"e"j')
```

```
hej
hej
h'e'j
h"e"j
```

Man kan även använda `\'` och `\"`:

```
print('h\'e\'j')
print("h\"e\"j")
```

```
h'e'j
h"e"j
```

Radbrytningar skrivs med `\n`:

```
print("h\ne\nj")
```

```
h
e
j
```

Vill man skriva ut flera saker med mellanslag mellan ger man flera argument till `print`:

```
print('hej','på','dig')
```

```
hej på dig
```

1.6 Aritmetiska operationer

Python stödjer de normala räknesätten:

- addition (+)
- subtraktion (-)
- multiplikation (*)
- division (/)
- heltalsdivision (//)
- exponentiering (**)
- modulus/rest (%)

Testa:

```
2+3   # ger 5
2-3   # ger -1
2*3   # ger 6
2/3   # ger 0.6666666666666666
2//3  # ger 0
2**3  # ger 8
2%3   # ger 2
```

Dessa fungerar även för flyttal. Man kan även addera strängar med + för att konkatenera dem:

```
'hej' + 'du'   # ger 'hejdu'
```

Det läggs dock inte in något mellanslag mellan orden, så det måste man lägga in själv:

```
'hej' + ' ' + 'du'   # ger 'hej du'
```

Egenskapen att + kan användas på olika sätt med olika typer av indata gör den till en *överladdad operation* (eng. *overloaded operator*).

De olika räknesätten binder på förväntat sätt (d.v.s. parenteser hamnar där de ska även om man inte skriver in dem). Exempelvis är `2 * 3 + 4` samma som `(2 * 3) + 4`:

```
2 * 3 + 4    # ger 10
(2 * 3) + 4  # ger 10
2 * (3 + 4)  # ger 14
```

I vissa fall kan man använda dem med olika typer av indata på samma gång:

```
2.0 ** 3 - 2.1312   # ger 5.8688
2 * 'hej'           # ger 'hejhej'
```

Men man kan t.ex. inte addera en siffra till en sträng eller multiplicera två strängar:

```
'hej' * 'hopp'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

en annan otillåten operation

```
2 + 'hej'
```

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

Här ser vi exempel på olika felmeddelanden som kan uppstå om man skriver något galet.

1.7 Input

Vi har sett att `print` kan användas för att skriva ut. För att läsa in använder man `input`. Om man bara skriver `input()` så skrivs inget ut först, om man ger `input` något argument så kommer det skrivas ut först:

```
namn = input("Vad heter du?\n")
print("Hej " + namn)
```

Detta lilla program ger i konsolen

```
Vad heter du? # Skriv in ditt namn här, t.ex. 'Johan'
Hej Johan
```

1.8 Speciella ord och namn i Python

1.8.1 Reserverade ord

Det finns ett antal reserverade ord i Python:

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

Om man försöker använda ett reserverat ord som variabelnamn får man ett felmeddelande:

```
and = 12
```

```
File "/Users/kxs624/.spyder-py3/DA2004/ht20/f1.py", line 1
  and = 12
      ^
SyntaxError: invalid syntax
```

Det finns annars ganska få krav på hur namnen kan väljas.

1.8.2 Identifierare

Identifierare är namn på resurser. Alla variabelnamn är exempel på identifierare (dvs de håller en "resurs" eller värde). Det finns dock identifierare som inte är variabler. Till exempel kommer vi gå igenom

funktioner, klasser och objekt senare i kursen. Dessa typer namnges med funktionsnamn, klassnamn och objektnamn som alla är identifierare.

Identifierare i Python:

- Kan bestå av bokstäver, siffror, och _.
- Får inte börja med siffra.
- Python skiljer på versaler och gemener; `Year` och `year` är inte samma identifierare.

Det är viktigt att välja bra namn på sina variabler för att koden ska vara lättare att läsa. Att döpa en variabel som innehåller någons ålder till `age` är mycket mer informativt än t.ex. bara `x`.

1.8.3 Litteraler

En litteral är ett konstant värde som man skriver i programkoden. Till exempel `17`, `2.5`, och `'SU'` är alla litteraler.

1.9 Uppgifter

1. Vad händer om vi kör nedanstående kod och varför blir det som det blir?

```
x,y = 2,3
x,y = y,x
print(x)
print(y)
```

2. Nedan har vi kod som definierar variablerna `x` och `y` med tilldelade värden. Skapa och skriv ut en variabel `z` med värdet 100 genom att använda aritmetiska uttryck av `x` och `y`.

```
x,y = 4, -3
```

3. Låt

```
s = 'hej'
```

   Skriv kod som skriver ut `'hejhejhejhejhejhejhejhejhejhejhejhejhejhej'` (d.v.s. `hej` sexton gånger på raken).

4. Låt

```
s = 'hej'
```

   Skriv kod som skriver ut `'hejhejhejhejhejhejhejhejhejhejhejhejhejhej'` genom att bara använda operatorn `+`, men så få gånger som möjligt.

5. Skriv kod som frågar användaren om dess förnamn och efternamn samt sparar resultatet i två variabler. Skriv sedan ut dessa. Ta inspiration från stycket 'Input' ovan (avsnitt 1.7). Till exempel ska det fungera så här:

```
Vad heter du i förnamn?    # Skriv in ditt förnamn här, tex 'Johan'
Vad heter du i efternamn?  # Skriv in ditt efternamn här, tex 'Jansson'
Hej Johan Jansson
```

6. Skriv kod som frågar användaren om dess ålder (spara i variabeln `age`) samt favorittal (spara i variabeln `num`). Vad händer om du adderar dessa variabler? Blir det vad du förväntar dig? Vad drar du för slutsatser om `age` och `num`?

Kapitel 2

Grunder i Python programmering

2.1 Syntax och semantik

Inom programmering pratar man ofta om *syntax* och *semantik*:

- *Syntax* handlar om regler för *hur* man skriver.
- *Semantik* handlar om *betydelsen* av vad man skriver.

Som ett exempel kan vi tänka på följande mening på Engelska:

> *Colorless green ideas sleep furiously.*

Mening är syntaktiskt korrekt, dvs den innehåller inget *grammatiskt* fel, men *innebörden* är inte något meningsfullt: semantiken är nonsens.

Syntax är speciellt viktigt, och kan vara svårt, när man börjar lära sig att programmera. Precis som ett vanligt språk, så handlar det om att lära sig reglerna för vad som är korrekt. Om man använder Python handlar det t.ex. om vad som är ett giltigt namn på en identifierare (vad man kan "döpa" en variabel till), var och hur man måste indentera kod-block, hur man skriver `if`-satser och `for`-loopar (vi kommer att gå igenom dessa i senare kapitel), etc. Syntax är specifikt för ett visst programmeringsspråk, även om det ofta finns många likheter mellan språk (Python har t.ex. många likheter med språket C). Att lära sig syntaxen går relativt snabbt genom övning, och fokus flyttas sedan till semantiken: vad händer i programmet (eller vad *borde* hända i programmet)?

Om du ser någonting i stil med `SyntaxError` i ett felmeddelande när du exekverar Python-kod så betyder det att du har skrivit något litet fel och att Python inte kan förstå din kod. I dessa fall får man ofta ett ganska instruktivt felmeddelande, eftersom Python kan säga till om vad som går fel. T.ex. kan följande felmeddelande fås:

```
    3a = 2
     ^
SyntaxError: invalid syntax
```

Felmeddelandet visar var den ogiltiga syntaxen förekommer: resultatet av ett ogiltigt namn för en identifierare.

Semantiska fel är svårare att hitta, eftersom Python inte säger till om något fel. Allt är giltig kod. Resultatet

av semantiska fel är att programmet inte gör vad man förväntar sig. Detta kan undvikas genom att skriva välorganiserad, väldokumenterad och välstrukturerad kod, dvs att skriva "bra" kod. Om det är tydligt vad man *vill* att koden ska göra när någon (eller man själv) läser koden, blir det mycket lättare att hitta semantiska fel.

2.2 Datatyper

Alla värden i Python har en typ. Vi har redan använt oss av olika typer när vi t.ex. tilldelat variabler värden som heltal, flyttal och strängar. De vanligaste typerna i Python som vi kommer att använda oss av är:

- `str` – strängar, dvs bokstäver (eller generellt tecken) inom `'` eller `"`
- `int` – heltal, av obegränsad storlek
  - (före Python v3.0: `int` 32 bitar, samt `long`)
  - De flesta språk: 32 bitar, ger [-2^32, 2^31-1]
- `float` – flyttal, en approximation av reella tal
  - Typiskt tal i [-10^308, 10^308], med 15 siffrors precision
  - 64 bitar, 11 för exponent e och 52 för "mantissa" m
  - Detaljer kan fås genom `import sys` och objektet `sys.float_info`
  - Varför är resultatet inte alltid exakt vad man förväntar sig? Svar: Python representerar flyttal genom bråk i bas 2. För detaljer se: https://docs.python.org/3/tutorial/floatingpoint.html Generellt gäller att det inte går att representera *alla* reella tal *exakt* på en dator, vilken talrepresentation som än väljs. *Vissa* reella tal kan representeras exakt medan andra får blir en approximation: fler bitar resulterar i en bättre approximation.
- `complex` – skrivs `1+2j`
  - Representeras med två flyttal
- `bool` – Booleska värden (dvs 'sant' eller 'falskt'), representeras av de reserverade orden `True` och `False`.
- `NoneType` – för värdet `None`

Man kan i Python se vilken typ en variabel eller litteral har genom att använda sig av `type`:

```
my_int = 1
my_float = 1.0
my_string = "1"
type(my_int)
type(my_float)
type(my_string)
type('hello')
```

```
int
float
str
str
```

2.2.1 Typomvandling

Man kan omvandla en typ till en annan, men det finns regler om hur specifika typer omvandlas och till vilka typer de kan omvandlas. Ofta behöver man inte ens tänka på att en typ omvandlas, utan det funkar ganska mycket som man rent intuitivt tänker sig. En sådan typomvandling kallas *implicit*. T.ex. är det sällan ett problem att blanda numeriska typer i ett uttryck:

```
my_int = 1
my_float = 1.0
c = my_int + my_float
type(c)
```

```
float
```

Här omvandlas alltså värdet i `my_int` till ett flyttal som sedan adderas till flyttalet som finns i `my_float`, och resulterar i ett flyttal `c`.

Som visats tidigare så finns det situationer där en sådan konvertering inte fungerar. Skulle man t.ex. utvärdera uttrycket `my_int + my_string` skulle man få ett `TypeError`. Man kan då använda sig av *explicit konvertering*, vilket man gör genom att ge typen man vill omvandla ett värde till värdet som argument. Om vi skulle vilja typomvandla `x`, så skulle vi kunna göra:

```
str(x)
int(x)
float(x)
complex(x)
bool(x)
```

Vi kan nu använda oss av explicit konvertering för att göra:

```
c = my_int + int(my_string)
type(c)
```

```
int
```

I detta fall kan värdet i `my_string` omvandlas till en `int` för att sedan adderas till `my_int`.

Varning: explicit typomvandling funkar inte alltid "magiskt":

```
int("hello!")
```

```
ValueError: invalid literal for int() with base 10: 'hello!'
```

Här vet inte Python hur `'hello!'` ska omvandlas till ett heltal.

Explicit konvertering är ofta viktigt när man använder `input()`, som vid lyckad inläsning ger typen `str`. I vissa fall vill man inte att ens variabel ska vara en `str`, som nedan:

```
age = int(input("Din ålder: "))
added_age = age + 10 # would not work if age was of type str
print(added_age)
```

Varning: Innan v3 så försökte Pythons `input()` tolka svaret åt dig. Det här gav buggar och säkerhetsproblem.

2.3 Booleska uttryck i Python

Vi har tidigare sett exempel på typiska aritmetiska operatorer (`+`, `-`,`*`, `/`, ...), som resulterar i ett värde som har typ beroende på vilka typer som uttrycket innehåller. Booleska uttryck använder sig av operatorer som resulterar i att en `bool` returneras, dvs `True` eller `False`. Sådana uttryck är väldigt användbara när man ska utvärdera om något är sant eller falsk givet ett test.

Det är värt att påpeka att det *mesta* i Python anses vara *sant*. Vi kan undersöka detta genom att använda typomvandling till `bool`:

```
bool(3)
bool(-427.2)
bool('hello')
```

```
True
True
True
```

Värden som anses vara *falska* är typiskt "tomma" värden, vad de än har för typ:

```
bool(0)
bool(0.0)
bool('')
bool(None)
```

```
False
False
False
False
```

2.3.1 Jämförelseoperatorer

Dessa operationer jämför indata genom följande relationer:

- Likhet: `x == y`
- Olikhet: `x != y`
- Strikt större än: `x > y`
- Strikt mindre än: `x < y`
- Större än eller lika: `x >= y`
- Mindre än eller lika: `x <= y`

Dessa fungerar som man förväntat för siffror (experimentera!), men som många operatorer i Python är det inte *endast* `int` eller `floats` man kan jämföra. T.ex.:

```
print('hello shark' < 'hello shark')
print('hello shark' < 'hello shork')
print('hello shark' < 'hello')
print('hello shark' < 'Hello shork')
```

```
False
True
```

```
False
False
```

Googla och se hur Python jämför strängar!

2.3.2 Logiska operatorer

Python använder sig av de logiska operatorerna `and`, `or` och `not`.

- Är `x` *och* `y` sant?: `x and y`
  `x and y` är `True` endast om både `x` och `y` är `True`.
- Är `x` *eller* `y` sant?: `x or y`
  `x or y` är `True` endast någon av `x` och `y` är `True`.
- Negera ett värde: `not x`
  `not x` är `True` om `x` är `False` och `False` annars.

Detta kan skrivas med sanningsvärdestabeller. Vi kan t.ex. använda oss av logiska operatorer för att definiera ett uttryck som är ett exklusiv-eller, `xor`, som utvärderas till sant om bara en av `x` eller `y`:

```
x = True
y = False
xor_result = (not ((x and y) or (not x and not y)))
print(xor_result)
```

```
True
```

Python behandlar `and` och `or` lite speciellt om (båda) operanderna inte är av typen `bool`: en `and`/`or`-sats utvärderas inte i dessa fall till `bool`. En `and`-sats ger tillbaka den *andra* operanden om båda operanderna är sanna, men ger tillbaka värdet av den *falska* operanden om uttrycket innehåller en sådan. En `or`-sats ger tillbaka det *första* värdet om första operanden är sann, annars värdet av andra operanden. Några exempel:

```
(42 and 'hello')
(42 and '')
(False and '')
(42.7 or 13)
('42' or None)
(False or 0)
```

```
'hello'
''
False
42.7
42
0
```

De här evalueringsreglerna kan förefalla konstiga, men har att göra med hur mycket av satsen som behöver utvärderas för att garanterat veta resultatet av hela uttrycket. Om första operanden till `and` är falsk, då behöver man inte beräkna värdet på det andra operanden; oavsett om den evaluerar till `True` eller `False` så ska resultatet motsvara `False`. Tekniken kallas ibland "kortslutning" (eng: *short circuiting*), eller mer generellt "lat evaluering" (eftersom det kan ses som "lathet" att inte beräkna allting). Syftet är förstås att få snabbare program genom att inte beräkna uttryck som inte behövs.

2.4 Tilldelningsoperatorer

Vi har redan sett att = används för tilldelning. En smidig funktionalitet i Python är att man kan kombinera detta med aritmetiska operatorer och få så kallade "tilldelningsoperatorer":

```
x += 3    # x = x + 3
x -= 3    # x = x - 3
x *= 3    # x = x * 3
x /= 3    # x = x / 3
x %= 3    # x = x % 3
x //= 3   # x = x // 3
x **= 3   # x = x ** 3
```

2.5 Funktioner

I och med att ens program blir större och kanske mer komplicerat, så märker man snabbt att det finns ett behov att kunna strukturera upp sin källkod. Det kan man börja göra med hjälp av *funktioner*. Vi har redan använt oss av flera av Pythons inbyggda funktioner, t.ex. `print`. Funktioner inom programmering har, som namnet antyder, likheter med matematiska funktioner: de tar ett eller flera argument och ger tillbaka ett retur-värde. Inom programmering *behöver* dock inte en funktion ta några argument eller lämna tillbaka ett värde.

Syftet med funktioner är att:

- Samla kod i lätthanterliga och återanvändbara delar
- Undvika upprepning av kod
- Förenkla avancerade program
- Öka *modulariteten*, dvs att programmet delas upp i mindre delar som är lättare att förstå än stora monolitiska program
- Göra programmet mer lättläst
  - Tydliggör beroenden i koden: "indata" som argument, resultat efter **`return`**. Det här påminner om en funktion i matematiken
  - Bra kodenhet för dokumentation
  - Bra kodenhet för att försäkra sig om kod gör det den ska (t.ex. genom testning)

Funktioner är därför ett mycket viktigt verktyg för att skriva bra kod.

Innan man kan använda en funktion, så måste funktionen *definieras*. Det gör man genom att använda sig av det reserverade ordet **`def`** för att ge funktionen ett namn (identifierare) och specificera argument. Nedan är ett exempel på en enkel funktion `f` som tar ett värde `x` som argument och adderar värdet till sig själv:

```
def f(x):
    return x + x
```

Denna funktion skulle matematiskt kunna skrivas som $f(x) = x + x$. När man väl definierat funktionen, så kan den kallas på följande sätt:

```
f(2.0)
```

```
4.0
```

Lägg märke till att Python inte bryr sig om vilken typ argumentet har, så länge alla operationer och uttryck i funktionen går att genomföra. Ovan används ett `float`, men vi kan också göra:

```
f(2)
f('hello')
```

```
4
'hellohello'
```

Precis som matematiska funktioner, så kan funktioner i Python ha flera argument:

```
def long_name(course_code, course_name):
    lname = course_code + course_name
    return lname
```

Vad är resultatet av `long_name('DA','2004')`?

Funktioner behöver inte ta några argument

```
def show_user_menu():
    print('Menu:')
    print('1. Add user')
    print('2. List users')
    print('3. Quit')
```

och kan också anropa andra funktioner:

```
def user_choice():
    show_user_menu()
    return input('Which assignment do you want to do? ')
```

2.5.1 Om funktioner

Ovan exempel introducerar flera nya syntaktiska koncept i Python. En funktion har en definition (även kallad "kropp") och ett huvud (även kallad "signatur").

- Huvudet har består av funktionsnamnet och en parameterlista ("funktionens argument").
  - Syntax: `def funktionsnamn(argument1, argument2, ...):`
- Ordet parameter har två betydelser.
  - Variabler i huvudet kallas *formella parametrar*. Formella parametrar är platshållare för de värden som kommer att ges vid ett funktionsanrop.
  - De värden man ger vid ett anrop till en funktion kallas *anropsparametrar*.
- Kroppen har en dokumentationssträng (frivillig) följt av kod, och är *indragen*.

Indragningen är viktig i Python! Python är ett "indenteringskänsligt" språk, dvs fel indrag ger syntaxfel. Vi använder här ordet *indentering* när vi menar indrag i källkoden. Indenteringen är det som specificerar det *kodblock* som hör till funktionskroppen. Även om man kan använda sig av vilken indentering man vill i Python (t.ex., två mellanslag eller en tab), så länge man är konsekvent, så är *standard*-indenteringen fyra mellanslag.

Att indenteringen ska vara fyra mellanslag specificeras i Pythons stilguide, känd som PEP-8, som också innehåller mycket annan bra information om hur man skriver bra Python-kod.

En modern programmeringseditor sätter in fyra mellanslag när använder tab-tangenten. En liten varning dock för att äldre eller mer primitiva editorer kanske inte hanterar tab-tangenten på ett bra sätt. Det kan ge problem beroende på hur editorn är konfigurerad vilket gör att er kod kan se olika ut på olika datorer.

Funktionskroppen avslutas ofta med det reserverade ordet **`return`**. Om **`return`** saknas så returneras värdet `None`. Det är bra att veta att **`return`** kan förekomma fler gånger i en funktion, men om en **`return`**-sats exekveras så avslutas alltid funktionen (och ger tillbaka de värden i den exekverade satsen).

Det är viktigt att förstå skillnaden på funktioner som returnerar något och de som bara t.ex. skriver ut något. Betrakta följande kodsnutt:

```
def f1(x):
    return x + x

def f2(x):
    print(x + x)
```

Skillnaden är att `f1` *returnerar* värdet på `x + x` medan `f2` bara *skriver ut* värdet. Om vi nu försöker skriva ut resultatet av `f1(2)` blir det som vi förväntat oss:

```
print(f1(2))
```

```
4
```

men om vi försöker skriva ut resultatet av `f2(2)` händer följande:

```
print(f2(2))
```

```
4
None
```

Anledningen är att `4` först skrivs ut av funktionen `f2`, men sen när vi skriver resultatet av `f2` så skrivs `None` ut då det är vad `f2` faktiskt returnerar! Det är väldigt viktigt att förstå den här skillnaden och när det står att ni ska skriva en funktion som *returnerar* något så ska **`return`** användas och inte `print`.

2.5.2 Multipla returvärden

Hur gör man om man vill returnera flera värden? Det finns flera dåliga sätt att lösa detta, men ett bra: multipla returvärden.

```
def integer_div(nominator, denominator):
    q = nominator // denominator
    r = nominator % denominator
    return q, r
```

Denna funktion kan man sedan anropa så här:

```
quotient, remainder = integer_div(17, 10)
```

2.5.3 Standardvärden

Det är vanligt att man skriver funktioner med parametrar som nästan alltid sätts till samma värde. Python låter dig ange explicit sådana vanliga värden: dessa kallas *standardvärden* (eng.: *default values*).

```
def f(x=1):
    return x + x

print(f())

print(f(3))
```

```
2
6
```

I första utskriften kan vi se att `f` har kallats med `x=1`, även om vi inte gett det som argument. När vi på nästa rad kallar `f(3)`, så sätts `x=3`.

2.5.4 Nyckelordsparametrar

Python låter dig namnge parametrar även i anropen. T.ex. så ger `integer_div(denominator=10, nominator=17)` ger samma effekt som anropet `integer_div(17, 10)`.

Vad händer om vi gör:

```
integer_div(denominator=10, nominator=17)
```

Python kopplar identifierarna på de formella parametrarna till identifierarna på argumenten, så ovan resulterar i precis samma output.

Ett exempel på en funktion som vi använt som har fler nyckelordsparametrar är `print`: * `sep=' '` — dvs ett mellanslag som separator * `end='\n'` — "backslash-n" betyder radbrytning. * `file=sys.stdout` — låt utdata gå till terminalen

Exempel:

```
print('hello','you',sep='\n')
```

```
hello
you
```

For att se mer information skriv `help(print)` i Spyder-konsolen (iPython).

2.5.5 Dokumentation av funktioner

Det finns två sätt att dokumentera i Python:

- Dokumentationssträngar
  - Sätts överst i funktionskroppen inom `""" """` eller `''' '''`
  - Primära dokumentationsmetoden
  - Knyts till funktionen. Prova `help(print)` i Python-tolken, eller Ctrl-i i Spyder när markören står vid funktionsanrop eller Python-instruktion.

- Kommentarer
  - Används för att förklara beräkningssteg, typiskt enskilda rader eller små block med kod.
  - Kommentarer är till för att förstå *hur* en funktion fungerar. Man ska kunna använda en funktion utan att läsa kommentarer.

Dokumentationssträngar är beskrivna i PEP-257:

> The docstring [...] should summarize its behavior and document its arguments, return value(s), side effects, exceptions raised, and restrictions on when it can be called (all if applicable). Optional arguments should be indicated. It should be documented whether keyword arguments are part of the interface.

Exempel på enradig dokumentationssträng:

```
def square(a):
    '''Returns argument a is squared.'''
    return a*a

print (square.__doc__)

help(square)
```

Exempel på flerradig dokumentationssträng:

```
def square(x):
    """Description of square function

    Parameters:
    x (int): input number

    Returns:
    int:Square of x
    """

    return x*x

print(square.__doc__)

help(square)
```

En allmän princip för dokumentation är att förklara det som inte framgår av identifierare. Om en funktion heter `compute_integral` så behöver inte dokumentationssträngen säga samma sak. Däremot kan det vara bra att skriva om vilken algoritm som används, vilka antaganden man har om parametrar, och vad som gäller för returvärdet.

2.6 Globala och lokala variabler

Variabler i Python kan vara antingen *globala* eller *lokala*. Namnet syftar till var en variabel är "synlig" för kod, och var variabeln kan modifieras/ändras. I följande exempel är `x` en global variabel då den har definierats på toppnivå (d.v.s. "längst ut").

```
x = "global"

def foo():
    print("x printed from inside foo:", x)

foo()
print("x printed from outside foo:", x)
```

Funktionen kan inte modifiera x, utan värdet som x innehåller är bara läsbart:

```
x = 42

def foo():
    x = x * 2
    print("x printed from inside foo:", x)
foo()
```

```
UnboundLocalError: local variable 'x' referenced before assignment
```

För att kunna ändra på värdet i x, så måste vi säga åt funktionen att den kan ändra på den globala variabeln. Det här gör man med det reserverade ordet **`global`**.

```
x = 42

def foo():
    global x
    x = x * 2
    print(x)
foo()
```

Varning: globala variabler som ändras av funktioner på detta sätt kan lätt leda till "spagettikod.". Det kan bli extremt svårt att hålla reda på var värdet på x ändrats! Så var väldigt försiktiga om ni använder globala variabler på detta sätt. Det finns dock scenarion när de är väldigt användbara, exempelvis globala räknare. Det är då väldigt bra att ha ett informativt namn på den globala variabeln så att man inte råkar ändra den av misstag!

Lokala variabler är variabler som definieras inuti i en funktion:

```
def foo():
    y = "local"
    print(y)

foo()
```

Variabeln y syns inte utanför funktionen (det är därför det kallas för *lokala* variabler):

```
def foo():
    y = "local"

foo()
print(y)
```

```
NameError: name 'y' is not defined
```

Var x är synlig är dess *synlighet* (eng. *scope*). Om x är definierat utanför funktionen och vi sedan skriver över den i funktionen så är dess värde oförändrat utanför funktionen:

```
x = 5

def foo():
    x = 10
    print("local x:", x)

foo()
print("global x:", x)
```

Samma sak händer med funktionsargument:

```
x = 5

def foo(x):
    print("local x:", x)

foo(12)
print("global x:", x)
```

2.7 Villkorssatser

Det händer ofta att man vill exekvera ett visst kodblock endast om någon sats eller något test är sant. Detta görs med så kallade *villkorssatser*, eller "`if`-satser". I Python görs detta med **`if`**, **`elif`** (som står för *else if*) och **`else`**.

```
answer = input('Write "done" if you are done: ')
if answer == 'done':
   quit()
print('Okay, I guess you were not finished yet.')
```

Obs: lägg märke till dubbla likhetstecken! Det är skillnad på *jämförelse* och *tilldelning* (dvs == och =).

Exemplet ovan kan även skrivas:

```
answer = input('Write "done" if you are done: ')
if answer == 'done':
   quit()
else:
    print('Okay, I guess you were not finished yet.')
```

Lägg märke till att `print`-satsen är indragen lika långt som `quit`. Precis som för funktioner så gäller det att det indenterade kodblocket exekveras.

Indenteringsfel är svåra att se ibland:

```
answer = input('Write "done" if you are done: ')
if answer == 'done':
   quit()
else:
    print('Okay, I guess you were not...')
   print('.. quite finished yet.') # indentation error
```

Ett till exempel med **else**:

```
namn = input('What is your name? ')
if namn == 'Anders':
   print('Hello Anders!')
else:
   print('Hello, whoever you are!')
```

Man kan använda sig av *nästlade* villkorssatser, dvs en villkorssats inuti en villkorssats:

```
temp = int(input('What is the temperature? '))
if temp < -30:
   print('That is very cold!')
else:
   if temp < 0:
      print('That is pretty cold...')
   else:
      if temp == 0:
         print('Zero')
      else:
         if temp < 30:
            print('That is a comfortable temperature')
         else:
            print('Wow! Super warm')
```

I ovan fall är koden inte särskilt lättläst Ett bättre alternativ vore att använda **elif**:

```
temp = int(input('What is the temperature? '))
if temp < -30:
   print('That is very cold!')
elif temp < 0:
    print('That is pretty cold...')
elif temp == 0:
    print('Zero')
elif temp < 30:
    print('That is a comfortable temperature')
else:
    print('Wow! Super warm')
```

Läsbarhet är viktigt! **elif** är strikt talat ej nödvändig då den alltid kan definieras med nästlade **if-else** satser, men den är väldigt användbar då den gör koden lättare att läsa. Denna typ av funktionalitet kallas ibland *syntaktiskt socker*.

Vad är skillnaden på beteendet för

```
x = int(input('Guess a number! '))
if x == 42:
    print('Correct!')
if x > 42:
    print('Too high...')
else:
    print('Too low...')
```

och

```
x = int(input('Guess a number! '))
if x == 42:
    print('Correct!')
elif x > 42:
    print('Too high...')
else:
    print('Too low...')
```

Vad händer om man skriver 42?

Slutligen så kan vi använda oss av villkorssatser för att skriva om `xor`-uttrycket från förra från Logiska operatorer och definiera en funktion:

```
def xor(x, y):
    if x and y:
        return False
    elif not x and not y:
        return False
    else:
        return True

print(xor(True,True))
print(xor(True,False))
print(xor(False,True))
print(xor(False,False))
```

```
False
True
True
False
```

Vilken version är lättast att läsa?

2.8 Uppgifter

1. Vilket eller vilka av alternativen nedan ska man tänka på när man väljer ett namn på en funktion eller variabel?

a) Namnet ska vara kort
b) Namnet ska vara beskrivande

c) Namnet ska vara unikt i funktionen och/eller skriptet
d) Namnet ska vara skrivet med stora bokstäver
e) Inget av ovanstående

2. Skriv kod som läser in ett heltal från användare och skriver ut `Odd` om talet är udda och `Even` om talet är jämnt.

3. Gör sanningstabeller för de booleska operatorerna `and` och `or` (d.v.s. fyll i `?` i tabellen nedan):

x	y	x and y	x or y
True	True	?	?
True	False	?	?
False	True	?	?
False	False	?	?

4. Försök, utan att köra koden, utvärdera dessa fyra uttryck för alla möjliga booleska värden på `x`, `y` och `z`:

```
(not x) or (not y)
x and (y or z)
(x != z) and not y
x and z
```

   Kör sedan koden för att bekräfta dina svar.

5. Skriv en Python-funktion som motsvarar den matematiska funktionen $f(x, y) = x/y$. Vi vill att funktionen ska säga till om man delar ett tal skiljt från 0 med 0. I detta fall så ska funktionen skriva ut `The result is infinite...`, men *inte* returnera något explicit värde. Om man delar 0/0, så ska funktionen skriva ut `Indeterminate form...` och inte heller returnera något.

6. Låt:

```
x = True
y = False
```

   Vilka uttryck nedan får värdet `True`? Varför?

```
(x and y) or (not x and not y)
(x or y) and (not x or not y)
x or y or not x or not y
not(not x and not (x and y))
```

7. Definiera `nand`, `nor` och `xnor` som Python-funktioner.

   Tips: https://en.wikipedia.org/wiki/Logic_gate#Symbols.

8. Prova funktionerna `nand`, `nor` och `xnor` med argument som *inte* är av typen `bool`. Är resultaten som du väntade dig? Om inte, varför?

Kapitel 3

Listor och iteration

Det här kapitlet handlar om:

- Listor
- Iteration med `for` och `while` loopar

Listor är viktiga i programmering då de låter oss samla flera element tillsammans. Iteration handlar om att automatisera upprepning och är ett fundamentalt koncept i så kallad *imperativ* programmering. Två viktiga konstruktioner som vi ska prata om i det här kapitlet är `for` och `while` loopar. Listor och loopar är väldigt användbara tillsammans, exempelvis kan man skriva ett program som läser in tal från användaren, sparar dem i en lista och beräknar medelvärdet av de inmatade talen.

3.1 Listor

Python har stöd för listor av element med olika typ. Dessa skrivs inom hakparenteser (“`[`” och “`]`”) och elementen i listan separeras med komma:

```
mylist = ["hej",2,"du"]
```

Pythons lista kan jämföras med *array* typen i andra språk, men den är mindre beräkningseffektiv. Dock är Pythons listor mer lättanvända. Detta är typiskt för scriptspråk: det är enklare att använda datastrukturer, på bekostnad av effektivitet.

3.1.1 Listindexering

Man kan hämta ut elementen ur en lista genom `mylist[i]` där `i` är ett heltal:

```
print(mylist[0])
print(mylist[1])
print(mylist[2])
```

```
hej
2
du
```

Viktigt: listor börjar indexeras från `0`!

Man kan även använda negativa värden för att hämta ut element från slutet av listan.

```
print(mylist[-1])
print(mylist[-2])
print(mylist[-3])
```

```
du
2
hej
```

Om man använder ett index som är för stort får man ett felmeddelande:

```
print(mylist[3])
```

```
IndexError: list index out of range
```

Fråga: vad händer om man indexerar med ett för stort negativt värde (t.ex. `mylist[-4]`)?

Man kan lägga till element i slutet av en lista med append.

```
mylist.append(True)
print(mylist)
```

```
['hej', 2, 'du', True]
```

Man kan enkelt tilldela ett nytt värde till ett element i en lista.

```
mylist[1] = 5
print(mylist)
```

```
['hej', 5, 'du', True]
```

Vi kan även göra detta genom att indexera från slutet av listan:

```
mylist[-2] *= 3
print(mylist)
```

```
['hej', 5, 'dududu', True]
```

Man kan hämta ut alla element med index `i < j` genom `mylist[i:j]`. Detta kallas för *slicing*.

```
print(mylist[0:1])
print(mylist[0:2])
print(mylist[1:3])
print(mylist[1:4])
```

```
['hej']
['hej', 5]
[5, 'dududu']
[5, 'dududu', True]
```

Man kan ändra steglängden i slicingen till `k` genom `mylist[i:j:k]`:

```
print(mylist[0:4:2])
print(mylist[1:4:2])
```

```
['hej', 'dududu']
[5, True]
```

3.1.2 Listfunktioner

Man kan addera listor och multiplicera dem med tal för att upprepa dem:

```
print(mylist + mylist)
print(2 * mylist)
```

```
['hej', 5, 'dududu', True, 'hej', 5, 'dududu', True]
['hej', 5, 'dududu', True, 'hej', 5, 'dududu', True]
```

Längden av en lista räknas ut med `len`:

```
print(len(mylist))
print(len(5 * mylist + mylist))
```

```
4
24
```

En väldigt användbar funktion är **`in`** som testar om ett element finns i en lista:

```
print(5 in mylist)
print(False in mylist)
```

```
True
False
```

Jämför med $x \in S$ från matematiken som säger att x finns i mängden S. Funktionen **`in`** fungerar även på strängar:

```
print("gg" in "eggs")
```

```
True
```

Det finns även **`not in`** vilket motsvarar $x \notin S$ från matematiken:

```
print(5 not in mylist)
print(False not in mylist)
```

```
False
True
```

Man hämtar ut minsta och största element ur en lista med `min` och `max`:

```
print(min([2,5,1,-2,100]))
print(max([2,5,1,-2,100]))
```

```
-2
100
```

Man kan ta reda på vilket index ett element finns på med hjälp av `index`:

```
print(mylist.index(True))
```

```
3
```

Försöker man hitta index för något som inte finns i listan får man ett felmeddelande:

```
print(mylist.index(42))
```

```
ValueError: 42 is not in list
```

Man kan räkna antalet gånger ett element finns med i en lista med `count`:

```
mylist.append(5)
print(mylist.count(5))
print(mylist.count(42))
```

```
2
0
```

Vi kan ta bort element på ett specifikt index eller en hel slice med **`del`**:

```
print(mylist)
del mylist[2]
print(mylist)
del mylist[0:2]
print(mylist)
```

```
['hej', 5, 'dududu', True, 5]
['hej', 5, True, 5]
[True, 5]
```

Detta är samma som `mylist[i:j] = []`. Man kan även använda **`del`** `mylist[i:j:k]` för att att få en annan steglängd.

Vi kan även ta bort elementet på index `i` och returnera det med `pop`:

```
mylist = ['hej', 5, 'dududu', True, 5]
print(mylist)
x = mylist.pop(2)
print(mylist)
print(x)
```

```
['hej', 5, 'dududu', True, 5]
['hej', 5, True, 5]
dududu
```

Detta fungerar även med negativa värden (dvs, genom att man indexerar från slutet). Ger man inget argument till `pop` så är det samma som `pop(-1)`, dvs sista elementet tas bort.

```
x = mylist.pop()
print(mylist)
print(x)
```

```
['hej', 5, True]
5
```

Man kan stoppa in ett element `x` på ett givet index `i` med `insert(i,x)`:

```
mylist.insert(1,'du')
print(mylist)
```

```
['hej', 'du', 5, True]
```

Detta är samma som att skriva `mylist[1:1] = ['du']`.

För att ta bort första elementet som är lika med något `x` använder man `remove(x)`:

```
mylist.append(5)
print(mylist)
mylist.remove(5)
print(mylist)
```

```
['hej', 'du', 5, True, 5]
['hej', 'du', True, 5]
```

Slutligen kan man använda `reverse` för att vända på en lista:

```
mylist.reverse()
print(mylist)
```

```
[5, True, 'du', 'hej']
```

3.1.2.1 Sammanfattning av användbara listfunktioner

- `mylist.append(x)`: lägg till `x` i slutet av `mylist`.
- `len(mylist)`: beräkna längden av `mylist`.
- `x in mylist`: testa om `x` finns i `mylist` (returnerar en `bool`).
- `x not in mylist`: testa om `x` inte finns i `mylist` (returnerar en `bool`).
- `min(mylist)` och `max(mylist)`: hämta ut minsta respektive minsta element i `mylist`.
- `mylist.index(x)`: räkna ut första index där `x` finns i `mylist`.
- `mylist.count(x)`: räkna hur många gånger `x` finns i `mylist`.
- `del mylist[i]`: ta bort elementet på index `i` ur `mylist` (utan att returnera det).
- `mylist.pop(i)`: ta bort elementet på index `i` ur `mylist` (och returnera det).
- `mylist.insert(i,x)`: sätt in `x` på index `i` i `mylist`.
- `mylist.remove(x)`: ta bort första förekomsten av `x` ur `mylist`.
- `mylist.reverse()`: vänd `mylist` baklänges.

För mer listfunktioner och detaljer om listor se: https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range

https://docs.python.org/3/library/stdtypes.html#lists

3.1.3 Praktisk funktion för att skapa listor: `range`

Ett sätt att skapa heltalslistor är att använda `range`:

```
print(list(range(1,4)))
print(list(range(3)))       # samma som range(0,3)
print(list(range(4,2)))
print(list(range(0,10,2)))
print(list(range(5,0,-1))) # iterera baklänges
```

```
[1, 2, 3]
[0, 1, 2]
[]
[0, 2, 4, 6, 8]
[5, 4, 3, 2, 1]
```

Obs: vi måste konvertera resultatet till en lista med hjälp av `list(...)` då `range` inte ger en lista direkt, utan ett "löfte" om en lista. Dvs, `range` returnerar ett objekt som skapar en lista vartefter man behöver en. Detta beteende kallas för *lat evaluering* och är smidigt då man inte behöver skapa hela listan på en gång. Detta har fördelen då hela listan inte behöver sparas i minnet på datorn på en gång vilket kan bli långsamt för stora listor. Mer om detta senare.

Om man försöker skriva ut `range(0,10)` utan att konvertera till en lista händer följande:

```
print(range(0,10))
```

```
range(0, 10)
```

3.1.4 Listors egenskaper

Listor har många styrkor, bland annat:

1. Enkla och effektiva att iterera över (mer om detta i nästa stycken).
2. Enkla att utöka.
3. Omedelbar åtkomst om man har index.
4. Listelementen kan ha olika typ.

Egenskap 3 är speciell i Python. Andra språk, som t.ex. C och Java, skiljer på *array* och *lista*. En array är som en vektor och har en fix storlek. I dessa språk är listor istället en speciell datastruktur där elementen är "länkade" så att man bara har omedelbar tillgång till det första (och kanske sista) elementet. Det gör att man inte automatiskt får omedelbar tillgång till alla element. Men i Python har man alltså direkt åtkomst till alla element genom listindexering vilket gör att dom påminner lite om arrayer.

En svaghet med listor är att de ineffektiva att *leta* i (dvs operatorn `in` kan vara långsam på stora listor). En annan svaghet är att oväntade saker kan hända om man har en lista som innehåller element av olika typ och försöker hämta ut största/minsta värde (se uppgift 2).

3.2 for loopar

`for` används för att iterera över uppräkningar av olika slag, framförallt listor. Dessa uppräkningar kallas *sekvenser* i Python och även strängar är sekvenser (av tecken).

```
for x in [1, 2, 3]:
    print(x)
```

```
1
2
3
```

Obs: notera att `print` satsen är indragen! Allt som kommer efter **`for`** och som är indenterat (med lika många mellanslag) kommer att köras i loopen. Som vi såg i förra kapitlet är Python är indenteringskänsligt, dvs det är viktigt att man indenterar kodblock lika mycket och detta gäller även för **`for`** loopar. Detta skiljer sig från språk som C och Java där man istället grupperar kodblock inom måsvingar (dvs "{" och "}") och rader avslutas med semikolon ";". Skulle man ha indenterat något fel i Python får man felmeddelandet:

```
IndentationError: expected an indented block
```

Ett kodblock kan innehålla flera instruktioner:

```
s = "A number: "

for x in [0,1,2]:
    print(x)
    s += str(x)

print(s)
```

```
0
1
2
A number: 012
```

Vi kan även iterera över en lista som ligger i en variabel:

```
mylist = [112, 857, "hej"]

for x in mylist:
    print(x)
```

```
112
857
hej
```

Eftersom strängar också är sekvenser kan man lätt iterera över bokstäverna i en sträng:

```
for c in 'DA2004':
    print(2*c)
```

```
DD
AA
22
00
00
44
```

3.2.1 Funktion som hämtar ut konsonanter ur en sträng

Låt oss nu implementera en lite mer komplicerad funktion som hämtar ut konsonanter ur en sträng. Innan man börjar skriva koden bör man fundera hur funktionen ska fungera. Givet en sträng `s` så ska den iterera över alla bokstäver och testa om de är konsonanter. Om bokstaven var det så ska den läggas till en sträng med alla konsonanter vi hittat i `s` och annars ska inget hände. Detta kan implementeras på följande sätt:

```
def consonants(s):
    cs = "BCDFGHJKLMNPQRSTVWXZbcdfghjklmnpqrstvwxz"

    out = ""

    for c in s:
        if c in cs:
            out += c

    return out

print(consonants("Hey Anders!"))
```

```
Hndrs
```

Notera hur vi definierar en variabel `out` vilken används för att spara de konsonanter vi hittat i `s`. Detta är ett väldigt vanligt sätt att programmera på och något vi kommer se mer av i kursen.

3.2.2 Dela upp en sträng i ord: `split`

Låt oss skriva en funktion som tar en sträng och bryter upp den i smådelar när man hittar ett blanksteg som separator. Det vill säga vi ska skriva en funktion `split(s)` som tar in en sträng `s` och returnerar alla ord i strängen. Detta kan göras genom att man har två variabler, en med de ord vi hittat och en med alla bokstäver i ordet vi håller på att bygga upp. För varje bokstav i `s` testar vi om den är ett mellanslag eller inte. Är det det så lägger vi ordet vi läst in i listan annars så lägger vi till bokstaven till ordet och tittar på nästa bokstav.

```
def split(s):
    out = []
    term = ""

    for c in s:
        if c == " ":
            out.append(term)
            term = ""
        else:
            term += c

    # add the final word unless it is empty
    if term != "":
        out.append(term)
```

```
    return out

print(split("Hey Anders! How are you?"))
```

```
['Hey', 'Anders!', 'How', 'are', 'you?']
```

Säg nu att vi vill generalisera detta till många olika möjliga separatorer, men att mellanslag ska vara standardvärdet, då skriver vi istället följande kod:

```
def split(s,sep = " "):
    res = []
    term = ""
    for c in s:
        if c in sep:
            res.append(term)
            term = ""
        else:
            term += c

    if term != "":
        res.append(term)

    return res

print(split("Hey Anders! How are you?"))
print(split("Hey Anders! How are you?","!"))
```

```
['Hey', 'Anders!', 'How', 'are', 'you?']
['Hey Anders', ' How are you?']
```

Vi kan nu alltså anropa funktionen med ett *eller* två argument.

Obs: det finns redan en variant på `split` inbyggd i Python!

3.2.3 Kontrollflöden: `pass`, `continue`, `break` och `return`

Man kan kontrollera och avsluta iterering med **`pass`**, **`continue`**, **`break`** och **`return`**.

Den enklaste av dessa är **`pass`** som inte gör någonting:

```
for i in range(4):
    if i == 1:
        pass
        print("passed 1")
    else:
        print(i)
```

```
0
passed 1
2
3
```

Obs: `pass` operationen kan verka oanvändbar då den inte gör någonting, men ett vanligt användningsområde är när man strukturerar upp sitt program men inte har skrivit klart allt. Man kan till exempel definiera en funktion där kroppen bara är `pass` eller ha en `if` sats där något fall inte är skrivet än. Detta gör att man kan testa all annan kod i programmet och sedan skriva klart de delar där man använt `pass` efter att man försäkrat sig om att allt annat fungerar.

En `continue` instruktion avslutar nuvarande iteration av loopen och går vidare till nästa:

```
for i in range(4):
    if i == 1:
        continue
        print("passed 1")
    else:
        print(i)
```

```
0
2
3
```

En `break` instruktion avslutar loopen helt:

```
for i in range(4):
    if i == 2:
        break
    else:
        print(i)
```

```
0
1
```

Slutligen, `return` används i en loop när man vill returnera något från en funktion. Till exempel:

```
def first_even_number(l):
    for number in l:
        if number % 2 == 0:
            return number

print(first_even_number([1,45,24,9]))
```

```
24
```

3.2.4 Pythons for är annorlunda

Det kan vara bra att vara medveten om att Pythons `for` loop är mer generell än i de flesta andra språk. Till exempel förväntas loopen gå över heltalsvärden i många äldre språk. Pythons konstruktion kan sägas vara modern: den bygger på att man konstruerar en datastruktur som stödjer iteration. Det är fullt möjligt, och inte så svårt, att skriva sina egna itererbara datatyper för speciella behov. Som vi sett går det att iterera över både listor och strängar, men vi kommer att se att även andra typer i Python stödjer det.

3.2.5 Ett vanligt problem med listor i Python: muterbarhet

Ett vanligt problem med listor i Python som vi kommer se flera gånger i kursen är att man kan råka ändra på listan som man itererar över utan att vilja det. Detta är en väldigt vanlig källa till buggar, både för nybörjare och mer erfarna programmerare.

Låt oss säga att vi vill köra följande program som är tänkt att lägga in de ord som är längre än 6 bokstäver längst fram i listan:

```
animals = ["cat", "dog", "tortoise", "rabbit"]

for x in animals:
    if len(x) > 6:
        animals.insert(0, x)

print(animals)
```

Men kör man detta kommer Python bara loopa och inte komma ur loopen! För att avsluta loopen trycker man `CTRL-c`. Problemet är att vi ändrar på `animals` listan samtidigt som vi loopar över den, så **`for`** loopen kommer aldrig avslutas.

För att undvika detta bör man itererar över en *kopia* av listan. Vi kan göra detta genom att använda `.copy()`:

```
animals = ["cat", "dog", "tortoise", "rabbit"]

for x in animals.copy():
    if len(x) > 6:
        animals.insert(0, x)

print(animals)
```

```
['tortoise', 'cat', 'dog', 'tortoise', 'rabbit']
```

Här används `animals.copy()`, istället för bara `animals`, så att vi får en kopia av listan och undviker den oändliga loopen.

Detta problem med listor kan komma upp när man programmerar och inte är försiktig med att kopierar saker. Detta kan leda till väldigt mystiska buggar som den oändliga loopen ovan. Det som gör att Pythons listor beter sig så här är att de är *muterbara*. Vi kommer prata mer om muterbarhet senare i kursen, men i dagsläget behöver ni bara komma ihåg att man kan behöva använda `.copy()` på en listvariabel för att undvika att råka ändra på den när man inte vill det.

3.3 `while` loopar

Ibland finns det inte någon självklar struktur att iterera över, men man vill ända repetera ett kodblock flera gånger. Då finns **`while`** loopen. Den fungerar så att man kör ett kodblock tills något booleskt test är uppfyllt. Om vi kör följande exempel:

```
i = 10
```

```
while i > 0:
   print(i)
   i -= 1

print("Done!")
```

```
10
9
8
7
6
5
4
3
2
1
Done!
```

Ett vanlig användningsområde för **while** loopar är att loopa tills man fått rätt indata från en användare. Programmet nedan kommer läsa in indata från användaren tills man skriver `exit`:

```
loop = True

while loop:
    print("Write 'exit' to quit")
    answer = input("Write something: ")
    if answer == "exit":
        print("Goodbye!")
        loop = False
    else:
        print("Let us keep going!")
```

```
Write 'exit' to quit
Write something: hey
Let us keep going!
Write 'exit' to quit
Write something: ho
Let us keep going!
Write 'exit' to quit
Write something: exit
Goodbye!
```

3.3.1 `while` loopar över listor

Listor är speciella och kan användas som booleska uttryck. Den tomma listan motsvarar `False` och alla andra listor motsvarar `True`.

```
list = []
```

```
if list:
   print("Not empty!")
else:
   print("Empty!")

list2 = ["Hey"]

if list2:
   print("Not empty!")
else:
   print("Empty!")
```

```
Empty!
Not empty!
```

Denna egenhet är till för följande *idiom* där man loopar över en lista och plockar bort dess element:

```
mylist = ["Ho","ho","ho!"]

while mylist:
    x = mylist.pop()
    print(x)

print(mylist)
```

```
ho!
ho
Ho
[]
```

När loopen är klar är listan tom och man behöver alltså inte jämföra listan med `[]`. Detta är alltså ett till exempel där Python har förändrat en lista i en loop.

3.3.2 Ta bort blanksteg i slutet av en sträng

Så här kan man ta bort blanksteg i slutet av en sträng med hjälp av en **`while`** loop och slicing:

```
def remove_trailing_space(s):
    i = len(s)
    while i > 0 and s[i-1] == " ":
        i -= 1
    return s[0:i]

print(remove_trailing_space("SU  "))
```

```
SU
```

Funktionen fungerar på så sätt att man börjar med en räknare `i` som sätts till längden på listan. Sen räknar vi hur många tecken från slutet av listan som är blanksteg och när vi är klara returnerar vi en slice av de ursprungliga listan där vi tagit bort alla blanksteg i slutet.

3.3.3 Största gemensamma delare: `GCD`

Den *största gemensamma delaren* (eng.: *greatest common divisor*, GCD) av två tal är det största heltal som delar dem. Exempelvis är 3 GCD av 15 och 12.

Euklides algoritm för att beräkna GCD för två tal `a` och `b` lyder så här:

- Antag `a >= b`,
- låt `r` vara resten av heltalsdivision `a // b`,
- om `r == 0`, då är `GCD(a, b) = b`,
- annars kan man använda att är `GCD(a,b) = GCD(b, r)`.

Lägg märke till att det är en villkorad iteration, men ingen tydlig struktur att iterera över. Så vi bör använda en **`while`** loop för att implementera den:

```
def GCD(a,b):
    if b > a:
        print("a must be bigger than b in GCD(a,b)")
        return None

    r = a % b
    while r != 0:
        a = b
        b = r
        r = a % b
    return b

print("GCD: " + str(GCD(15,12)) + "\n")
print("GCD: " + str(GCD(12,15)))
```

```
GCD: 3

b must be bigger than a in GCD(a,b)
GCD: None
```

3.4 Uppgifter

1. Vad är resultatet av `[1,2] + ['a', 'b']`?
2. Vad händer om man använder `min`/`max` på listor med element av olika typ?
3. Skriv klart funktionen

   ```
   def vowels(s):
       vs = "AEIOUYaeiouy"
       pass
   ```

   vilken tar in en sträng `s` och returnerar dess vokaler.
4. Modifiera funktionen i uppgift 3 så att den istället bara returnerar konsonanter utan att ändra på strängen `vs`. Skriv sen en funktion `vowels_or_consonants` med en extra parameter vilken

bestämmer om vokaler eller konsonanter ska returneras. Parametern ska ha ett standardvärde så att vokaler returneras om användaren inte kallar på funktionen med två argument.

5. Vad händer om man inte har `if term != ""` blocket i funktionen `split(s)` ovan? Hitta på ett exempel på en sträng `s` där utdata blir fel om man inte har det. Vidare, hur beter sig vår funktion `split` för strängar med flera mellanslag i rad? Skriv sen en korrekt version av funktionen.

6. Testa följande kodsnutt:

```
list = [1,2,3]
list2 = list
list2.reverse()
print(list)
```

   Fungerar den som ni trodde? Varför inte? Hur kan man fixa det?

7. Skriv om koden

```
mylist = ["Ho","ho","ho!"]

while mylist:
    x = mylist.pop()
    print(x)

print(mylist)
```

   så att listan `mylist` inte är tom när man är klar.

   Tips: använd `.copy()`. Hur kan man göra om man inte får använda copy?

8. Testa att använda **`continue`**, **`break`** och **`pass`** i **`while`** loopen nedan. Vad händer?

```
n = 10
while n > 0:
     n -= 1
     if n == 7:
         # test continue, break or pass here
     print(n)
```

9. Skriv en **`while`** loop som läser in tal från användaren och sparar dem i en lista. Detta ska göras tills användaren skriver in `0` och då ska loopen avslutas och listan skrivas ut.

10. Skriv funktionen

```
def naiveGCD(a,b):
    pass
```

    vilken beräknar GCD av `a` och `b` genom att testa alla tal mellan `1` och `min(a,b)` för att se vilket som är det största talet som delar båda.

11. Skriv funktionen `portkod()` som läser in en siffra (`1`–`9`) i taget från användaren. Efter varje siffra är inläst ska funktionen testa om de fyra senaste siffrorna matchar med korrekt kod. Om korrekt kod ges ska funktionen skriva ut `Door unlocked!` och returnera `True`. Välj vad som är korrekt portkod själv, t.ex. `1337`.

    Tanketips: Vilken loop ska vi använda?

Kapitel 4

Uppslagstabeller, filhantering och mer om loopar

En "datastruktur" är en typ för att spara data. Detta kan göras på många olika sätt och vi har sett ett antal i kursen än så länge, bland annat listor. I det här kapitlet ska vi titta på uppslagstabeller vilket är som listor som kan indexeras med andra typer än heltal. Ett annat sätt att spara data är att skriva det till en fil på datorn. Vi ska i kapitlet även titta på hur detta kan göras. Slutligen kommer vi även titta lite mer på vad man kan göra med loopar i Python.

4.1 Uppslagstabeller

Uppslagstabeller (eng. *dictionaries*) är en datastruktur som blivit en "arbetshäst" för scriptspråk. De är enkla att använda och bra till mycket. Uppslagstabellen är som en lista, fast istället för att indexera med heltal indexeras de med *nycklar* (eng. *keys*). Man kan tänka på en uppslagstabell som en mängd av nyckel-värde par (eng. *key-value pairs*). I Python heter typen för uppslagstabeller `dict`.

Det finns två typiska sätt i Python att skapa en uppslagstabell. T.ex. kan man skapa en uppslagstabell som har nycklarna `'DA2004'` och `'MM2001'` med respektive värden `7.5` och `30` genom:

```
hp1 = dict(DA2004=7.5, MM2001=30)
hp2 = {'DA2004': 7.5, 'MM2001': 30}
```

Båda sätten är identiska och `hp1 == hp2`. Lägg märke till att första sättet har samma syntax som att kalla en funktion med nyckelparametrarna `DA2004` och `MM2001`, medan det andra sättet använder sig av en *sträng* för nyckeln och värdet separerat med ett kolon.

För att komma åt ett värdet används `[]`-syntax med *nyckelvärdet*:

```
print('Number of ECTS points for DA2004: ' + str(hp1['DA2004']))
```

```
Number of ECTS points for DA2004: 7.5
```

Lägg märke till att själva nyckeln också kan ha ett värde. I både `hp1` och `hp2` är typen av nyckelvärdet `str`, dvs strängar. Ett exempel som illustrerar denna poäng:

```
DA2004 = 33
MM2001 = 'hello'
hp3 = {DA2004: 7.5, MM2001: 30}
print('Number of ECTS points for DA2004: ' + str(hp3[DA2004]))
print('Number of ECTS points for DA2004: ' + str(hp3[33]))
print('Number of ECTS points for MM2001: ' + str(hp3[MM2001]))
print('Number of ECTS points for MM2001: ' + str(hp3['hello']))
```

```
Number of ECTS points for DA2004: 7.5
Number of ECTS points for DA2004: 7.5
Number of ECTS points for MM2001: 30
Number of ECTS points for MM2001: 30
```

Här har antingen värdet av t.ex. `MM2001` använts som nyckel direkt, eller så identifieraren använts. Om man skulle försöka använda sig av en *sträng* av *värdet* "MM2001' här, så får man ett fel:

```
print('Number of ECTS points for MM2001: ' + str(hp3['MM2001']))
```

```
KeyError: 'MM2001'
```

Det här har att göra med att alla objekt i Python som inte kan ändras kan användas som nycklar. Uppslagstabeller implementeras med en teknik som kallas hashtabeller, som innebär man konverterar nyckeln till ett heltal (det "hashas", vilket avser att man "hackar upp" och blandar bytes till ett heltal) som används för uppslagning. Om nyckel kan ändras, "muteras", då finns det risk att man inte hittar tillbaka till det man stoppat in i tabellen. Generellt gäller att *icke-muterbara* värden, t.ex. strängar, tal, tupler etc., kan användas som nycklar. Det här utesluter muterbara typer som t.ex. listor.

Använder man sig av `dict()` för att skapa en uppslagstabell, så tolkas nyckelparametrarna alltid som strängar.

För mer detaljerad dokumentation av uppslagstabeller i Python se:

https://docs.python.org/3/tutorial/datastructures.html#dictionaries

https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

4.1.1 Viktiga egenskaper

- En uppslagning är billig (dvs snabb).
- Det är enkelt att ändra innehåll.
- Uppslagstabeller är enkla att iterera över (men du får inte ändra i en uppslagstabell under iteration).

4.1.2 Andra namn

- Hashtabell
- Hash map

I "traditionella" språk (C, C++, Java, m.fl.) har de inte samma särställning som i scriptspråken. Scriptspråken har gjort uppslagstabeller lättanvända.

4.1.3 Fler exempel

Tomma uppslagstabellen kan skrivas `{}`, detta är syntaktiskt socker för `dict()`.

```
empty = {}
print(empty)
print(dict())
```

```
{}
{}
```

Som vi noterat ovan kan man skapa uppslagstabeller på följande sätt:

```
test1 = {'hopp': 4127, 'foo': 4098, 'hej': 4139, 12: 'hej'}
test2 = dict([('hej', 4139), ('hopp', 4127), ('foo', 4098)])

print(test1)
print(test2)
```

```
{'hopp': 4127, 'foo': 4098, 'hej': 4139, 12: 'hej'}
{'hej': 4139, 'hopp': 4127, 'foo': 4098}
```

Observera att nycklar och värden inte behöver vara av samma typ!

Man kan uppdatera uppslagstabeller på liknande sätt som listor, och lägga till nya nyckel-värde par genom `[-]` och ta bort med **`del`**:

```
hp = {}
hp['DA2004'] = 7
print(hp['DA2004'])

hp['DA2004'] += 0.5
print(hp['DA2004'])

hp['MM2001'] = 30
hp['UG1001'] = 1
print(hp)

del hp['UG1001']
print(hp)
```

```
7
7.5
{'DA2004': 7.5, 'MM2001': 30, 'UG1001': 1}
{'DA2004': 7.5, 'MM2001': 30}
```

4.1.4 Iterera över uppslagstabeller

Vi kan använda **`for`** för att iterera över nycklarna:

```
for key in hp:
    print(key)
```

```
DA2004
MM2001
```

Vi kan även använda `.keys()`:

```
for key in hp.keys():
    print(key)
```

```
DA2004
MM2001
```

och `.values()`:

```
for val in hp.values():
    print(val)
```

```
7.5
30
```

Vi kan även extrahera alla nyckel-värde par med `.items()`:

```
for key, val in hp.items():
    print('Number of ECTS points for ' + key + ': ' + str(val))
```

```
Number of ECTS points for DA2004: 7.5
Number of ECTS points for MM2001: 30
```

Alla dessa tre funktioner returnerar itererbara objekt med element som består av antingen nycklar, värden eller tupler med nyckel-värde par.

4.1.5 Funktioner för uppslagstabeller

Vi kan även manipulera uppslagstabeller med liknande funktioner som för listor:

```
print(len(hp))
hp['foo'] = 20.0
print(len(hp))

print('DA2004' in hp)
print('20.0' in hp)
```

```
2
3
True
False
```

En till användbar funktion är `.get()`, som tar två argument: nyckelvärdet och ett värde som returneras om nyckeln *inte* finns i uppslagstabellen. Om nyckeln finns, returneras värdet som hör till nyckeln.

```
print(hp.get('DA2004', 42)) # key exists
print(hp.get('bar', 42)) # key does not exist
```

```
7.5
42
```

4.2 Filhantering

Filer innehåller information som ligger på hårddisken, t.ex. textfiler, bilder eller filmer, etc. Du kommer åt filer genom att interagera med operativsystemet. Detta kan göras direkt från Python.

4.2.1 Läsa från filer

För att t.ex. läsa från en fil `data.txt` skaffar vi först ett "handtag" (eng. *handle*). Det gör man med `open()`, som tar två obligatoriska argument: filnamnet som en sträng och en sträng som specificerar om filen ska vara t.ex. läsbar, skrivbar etc. Se hjälpen/dokumentationssträngen för `open`.

```
h = open('data.txt', 'r')   # 'r' för "read"
```

För att läsa in hela filens innehåll till en sträng skriver vi sedan:

```
h.read() # whole file in string
```

eller en lista med strängar där varje list-element är en rad från filen:

```
h.readlines() # whole file as list of strings
```

Detta funkar inte bra i många fall med stora filer. Det blir problem om man läser in en fil som är större än datorns RAM-minne.

För att bara läsa in en rad från `h` använder vi:

```
h.readline()
```

Notera att innehållet i en fil läses *en gång*, så har vi redan använt oss av `read` eller `readlines` så kommer `readline` att inte göra något, eftersom handtaget/filpekaren redan har kommit till slutet av filen.

4.2.2 Skriva till filer

Om vi vill skriva till en fil `out_data.txt` måste vi först skapa ett handtag (med `w` för *write*).

```
output = open('out_data.txt', "w")
```

Detta *skapar* eller *tömmer* filen `out_data.txt`. **Varning:** Var försiktig då allt i filen `out_data.txt` kommer att försvinna om den redan finns! Om vi vill skriva till `output` (dvs till filen `out_data.txt`) så använder vi `write`:

```
output.write("This is my first line")
output.write("\n") # start a new line
output.write("This is my second line")
```

4.2.3 Stänga filer

När vi är klara med en fil bör vi stänga den:

```
h.close()
output.close()
```

Det är viktigt att stänga filer:

- Antalet öppnade filer är begränsat i ett operativsystem – slösa inte med dem!
- Ett annat program kan vilja skriva till öppnad fil – släpp den så annat program inte behöver vänta!
- Du kan förlora data om du inte stängt filen.
- När ett program avslutas så stängs filen automatiskt.

4.2.4 Ett exempel

Tänk dig en fil `people.txt` med komma-separerad information: `efternamn,förnamn,födelseår`.

```
Svensson,Lisa,1815
Olsson,Erik,1901
Mörtberg,Anders,1986
```

Följande program skapar en fil `modern.txt` med alla från 1900-talet:

```
h_in = open('people.txt', 'r')
h_out = open('modern.txt', 'w')
for n in h_in:    # loop through the file, row by row
    lname, fname, year = n.split(',')
    if int(year) > 1900:
        h_out.write(fname + ' ' + lname + '\n')
h_out.close()
h_in.close()
```

Öppnar vi nu filen `modern.txt` så kommer den innehålla

```
Erik Olsson
Anders Mörtberg
```

4.2.5 Viktigt idiom: `with`

Alternativ notation till open/close:

```
with open('ut.txt', 'w') as out:
     out.write("hej hopp")
```

Detta lilla exempel motsvarar

```
out = open('ut.txt', 'w')
out.write("hej hopp")
out.close()
```

God praxis att använda ‘with’ — det är idiomatisk Python.

Fördelar:

- Lättläst
- Uppmuntrar god kodstruktur
- Anses underlätta felhantering
- Filer stängs när man lämnar `with`-blocket

I Python kallas `with` för en *kontexthanterare* (eng: *context handler*) och kan tillämpas även för annat än filer.

Exemplet ovan kan skrivas om med `with` på följande sätt:

```
with open('people.txt', 'r') as h_in, open('modern.txt', 'w') as h_out:
    for n in h_in:
        lname,fname,year = n.split(',')
        if int(year) > 1900:
            h_out.write(fname + ' ' + lname + '\n')
```

4.2.6 Att tänka på kring filhantering

- Läsa och skriva filer är långsamt.
  - Undvik att läsa en fil två gånger.
  - Använd som mellanlagring bara om du måste.
- En del filer är stora. Undvik att läsa in allt om det ej behövs.
- Undvik att hårdkoda sökvägar.
- Bra princip: läs från `stdin` och skriv till `stdout`. Unix-tradition.
- Tillfälliga filer bör ha unika och tillfälliga namn.
  - Använd modulen `tempfile` (https://docs.python.org/3/library/tempfile.html för säkerhet och bekvämlighet!

4.3 Mer om loopar: nästling

Syntaxen för en `for`-loop bygger på att den indragna "kroppen" på loopen har en eller flera satser. Man kan förstås använda *fler* loopar:

```
for i in range(3):
    for j in range(3):
        print(i * j)
```

```
0
0
0
0
1
2
0
2
4
```

Märk att man gör ökar indraget för varje loop man gör. Indraget är otroligt viktigt för att avgöra strukturen på programmet och hur det exekveras! Ett exempel:

```
print("---------")
for i in range(1,4):
    for j in range(1,4):
```

```
        print(i, j, i * j, sep="   ")
    print("---------")
```

```
---------
1   1   1
1   2   2
1   3   3
---------
2   1   2
2   2   4
2   3   6
---------
3   1   3
3   2   6
3   3   9
---------
```

Notera att `print("---------")` endast körs en gång i den yttre loopen. Vad hade hänt vi istället haft den lika långt indragen som `print(i, j, i * j, sep="   ")`?

Vi kan även kontrollera att endast beräkna jämna produkter med hjälp av villkorssatser i looparna. Notera indenteringsnivån på de två olika **`if`**-satserna.

```
for i in range(1,8):
    if i % 2 == 1:
        continue
    for j in range(1,8):
        if j % 2 == 1:
            continue
        print(i * j)
```

```
4
8
12
8
16
24
12
24
36
```

4.4 Uppgifter

1. Givet de två listorna nedan

   ```
   a = ['a','b','c']
   b = [1, 2, 3]
   ```

   skriv kod som skapar en uppslagstabell `{'a': 1 ,'b': 2,'c': 3}`.

2. Skriv kod som tar alla par `(x,y)` av tal mellan `2` och `20` med `x != y` och skriver ut deras gemensamma delare (GCD).

   Tips 1: Använd nästlade loopar och range.

   Tips 2: Använd funktionen `GCD` från förra kapitlet.

   Koden ska till exempel skriva ut följande:

```
x y GCD
2 3 1
2 4 2
2 5 1
2 6 2
2 7 1
2 8 2
# osv...
```

3. I uppgift 2 är det onödigt att skriva ut GCD för t.ex. `x,y = 2,3` och `x,y = 3,2`. Gör om koden så att `x` alltid är mindre än `y` i raderna som skrivs ut.

   Tips: I den inre `for` loopen, vad händer om man itererar från nuvarande värde på `x`?

4. I kapitel 3 jobbade vi med funktionen `vowels`. Använd en uppslagstabell som har vokalerna i `vs` som nycklar och ett tal som värde (du kan hitta på talen själv). Modifiera funktionen så att den istället byter ut vokalerna mot talen.

5. I kapitel 3 jobbade vi med funktionen `split` (given nedan). Ha strukturen i denna kod som hjälp för att skriva en funktion `annotate_word` som tar en sträng och associerar varje ord med ett tal där talet representerar när ordet sist förekom i texten (räknat från 1). Funktionen ska returnera en uppslagstabell med ord som nycklar och tal som värden.

```
def split(s):
    out = []
    term = ""
    for c in s:
        if c == " ":
            out.append(term)
            term = ""
        else:
            term += c

    # add the final word unless it is empty
    if term != "":
        out.append(term)

    return out
```

   Funktionen ska funka så här:

```
print(annotate_word("hej du hej hej du"))
```

```
{'hej': 4, 'du': 5}
```

6. Hur kan vi enkelt få `annotate_word` i uppgift 5 att istället returnera en uppslagstabell med tal som nycklar och ord som värden?
7. Modifiera `annotate_word` i uppgift 5 så att inte bara sista förekomsten sparas i texten, utan alla förekomster. Vilken datastruktur bör vi ha som *värde* i vår uppslagstabell?

Kapitel 5

Felhantering/särfall och listomfattning

Det blir fel ibland, även när man kör program, och felen måste då hanteras. När man programmerar är det typiskt status på en funktion man är intresserad av: gick beräkningen bra eller har det uppstått ett fel? Hur fortsätter vi efter felet?

Hittills i kompendiet har vi definierat funktioner som returnerar resultat (värden), samt att något instruktivt meddelande kanske har skrivits ut med hjälp av att kalla `print`. Alternativt har programmet bara kraschat eller fastnat i en oändlig loop.

I detta kapitel kommer vi att undersöka hur osäker kod vid fel kan påkalla, eller "lyfta", ett **särfall**. Detta kan hanteras på ett lämpligt sätt med hjälp av **felhantering** vilket i Python görs med `try` och `except`.

Vi ska även i slutet av kapitlet titta på ett helt annat koncept: listomfattningar. Detta är ett smidigt sätt att skapa och modifiera listor på vilket ofta är mycket mer kompakt än att skriva en loop.

5.1 Felhantering och särfall

Ett naivt sätt att hantera fel i program är att returnera någon form av felkod i form av ett tal eller en sträng. Vi ska nu börja med att titta på hur detta kan se ut, men sen snabbt gå över till att titta på hur fel kan hanteras bättre med hjälp av särfall.

5.1.1 Felhantering med hjälp av returnerade felkoder

Att förlita sig på returvärden för felhantering ger komplexa program som tenderar att bli svårlästa. Många funktionsanrop måste följas av `if`-satser för att se om ett fel uppstod som man måste anpassa sig efter. Om ett fel uppstod kan det bli knepigt att återgå till en bra plats i programmet. Erfarenheten är att programmerare undviker den ökade komplexiteten i sitt program genom att ignorera de returnerade felkoderna vilket ger instabila program som kan krascha. Denna typ av felhantering förekommer ofta i t.ex. C, Fortran och andra äldre programmeringsspråk.

Ett exempel på denna omoderna teknik följer nedan där vi kallar på en funktion som ska modifiera en lista där alla returkoder utom 0 signalerar att något fel har skett.

```
def divide_list(l,x):
    if type(x) != int or type(x) != float:
        return 11
    elif x == 0:
        return -1
    else:
        # check att elements in list are integers
        if not all([ type(item) == int for item in l]):
            return 12
        for i in range(len(l)):
            l[i] = l[i]/x # we are modifying the list items here
        return 0

return_code = divide_list(l,x)
if return_code == -1:
    # do something
elif return_code == 11:
    # do something else...
elif return_code == 12:
    # do something else...
elif return_code == 0:
    # good!
```

Denna kod är ganska rörig och onödigt komplicerad. Många moderna språk stödjer därför specifika sätt att hantera fel på genom så kallade "särfall" (eng: *exceptions*).

5.1.2 Särfall

Python använder sig av de reserverade orden **try** och **except** för att runt osäkra passager dela in koden i block som exekveras som åtgärd vid ett särfall.

```
try:
   # unsafe code
   ...
except:
   # action
   ...

# Rest of the code
...
```

Konstruktionen **try** lägger man alltså runt ett kodblock som kan betraktas som en osäker passage (`# unsafe code` ovan) och efter **except** lägger man de åtgärder som man hanterar eventuella problem med. Precis som vid funktionsdefinitioner, villkorssatser och kontrollflöden så är här indenteringen viktig. Blocket som hör till **except**-satsen (`# action`) exekveras endast om ett fel har gett ett särfall i den osäkra passagen under **try**-satsen: om inget fel har uppstått fortsätter koden utan att **except**-blocket körs. Om ett fel uppstår, så kan åtgärderna under **except** antingen avsluta exekveringen av programmet

eller så kan felet behandlas på något annat sätt (t.ex. ignoreras, även om man ska vara försiktig med att just strunta i fel som ger ett särfall).

5.1.2.1 Exempel: läsa in ett heltal från användaren

Betrakta följande kodsnutt:

```
while True:
    answer = input("Write a number (write 0 to quit): ")
    x = int(answer)
    if x == 0:
        print("Bye bye!")
        break
    else:
        print("The square of the number is: " + str(x ** 2))
```

Vad händer om användaren skriver in något annat än ett tal?

```
Write a number (write 0 to quit): hej
```

```
ValueError: invalid literal for int() with base 10: 'hej'
```

Detta är, som vi sett tidigare, en konsekvens av att resultatet av `input` alltid har typen `str`, och att det inte alltid går att explicit typomvandla från `str` till `int`. Här är `ValueError` ett särfall: värdet av inparametern (`'hej'`) till `int` är felaktigt. Om man istället använder sig av **`try`**/**`except`**-sats så kan felet hanteras på rätt sätt:

```
while True:
    try:
        answer = input("Write a number (write 0 to quit): ")
        x = int(answer)
        if x == 0:
            print("Bye bye!")
            break
        else:
            print("The square of the number is: " + str(x ** 2))
    except ValueError:
        print("You must write a number!")
```

I koden ovan fångas alla särfall av typ `ValueError` som lyfts i **`try`**-blocket. Om ett annat typ av särfall skulle påträffas skulle programmet avslutas. Man kan även fånga *alla* fel med endast **`except`**:

```
while True:
    try:
        answer = input("Write a number (write 0 to quit): ")
        x = int(answer)
        if x == 0:
            print("Bye bye!")
            break
        else:
            print("The square of the number is: " + str(x ** 2))
```

```
    except:
        print("You must write a number!")
```

För att koden ska bli så överskådlig och tydlig som möjligt, vill man dock helst skriva så lite kod som möjligt innanför **try**-blocket. Genom att vara specifik i sin avsikt, kan ovan exempel där **if**-satsen inbakat i **try** och bara **except** används, förbättras till följande:

```
while True:
    answer = input("Write a number (write 0 to quit): ")
    try:
        x = int(answer)
    except ValueError:
        print("You must write a number!")
        continue

    if x == 0:
        print("Bye bye!")
        break
    else:
        print("The square of the number is: " + str(x ** 2))
```

5.1.2.2 Exempel: beräkna längden av rader i en fil

Låt oss säga att vi vill beräkna längden på alla rader i en fil:

```
def len_file(filename):
    with open(filename, 'r') as h:
        for s in h:
            print(len(s))

len_file("foo.txt")
```

Om filen `foo.txt` inte finns får vi följande

```
FileNotFoundError: [Errno 2] No such file or directory: 'foo.txt'
```

Det är bättre att skriva:

```
def len_file(filename):
    try:
        with open(filename, 'r') as h:
            for s in h:
                print(len(s))
    except FileNotFoundError:
        print("Could not open file " + filename + " for reading.")

len_file("foo.txt")
```

5.1.2.3 Olika åtgärder för specifika särfall

Det är mycket som kan gå fel. Beroende på vilket särfall som uppstår, så kan det vara lämpligt att utföra olika åtgärder. I nedan exempel så görs separata åtgärder för om något går fel vid filinläsningen, om ett tal delas på noll eller om den inlästa strängen från filen inte kan typomvandlas till ett heltal:

```
def divide_by_elems(filename,x):
    divs = []
    try:
        with open(filename, 'r') as h:
            for n in h:
                frac = x / int(n)
                divs.append(frac)

    except IOError:
        print("divide_by_elems: A file-related problem occured.")
    except ZeroDivisionError:
        print("divide_by_elems: Division by zero.")
    except ValueError:
        print("divide_by_elems: Character could not be converted to int.")
    return divs

data = divide_by_elems('numbers.txt', 2)
print(data)
```

T.ex. om ovan kod skulle köras med `numbers.txt` som innehåller följande

```
1
2
0
4
3
0
```

så skulle två typer av särfall fångas. Kan du se vilka och på vilka rader i `numbers.txt`?

5.1.2.4 Särfall för kontrollflöde

Man kan även använda särfall för kontrollflöde i sitt program. Låt oss säga att vi vill beräkna kvoten mellan varje element i två listor (och ignorera alla element i slutet av den längre listan), men inte krascha om det blir division med noll. Då kan vi skriva följande:

```
def compute_ratios(xs,ys):
    ratios = []
    for i in range(min(len(xs),len(ys))):
        try:
            ratios.append(xs[i]/ys[i])
        except ZeroDivisionError:
            ratios.append(float('NaN')) # NaN = Not a Number
    return ratios
```

```
print(compute_ratios([2,3,4],[2,0,5]))
print(compute_ratios([2,3,4,0],[2,0,5]))
```

I detta fall hade vi dock lika gärna kunnat använda en **if**-sats och testat om `ys[i] == 0`. Mer om detta senare.

5.1.3 Allmänt om särfall

Det anses vara idiomatisk Python att använda sig av **try-except**. Det är inte ovanligt att när man läser om Python-kod ser *EAFP*, som står för *It's Easier to Ask Forgiveness than Permission*. Det här kontrasteras mot *Look Before You Leap (LBYL)*. Det är inte menat att man ska *överanvända* **try-except**: särfall som uppstår kan ibland ge den tydligaste informationen. Varje möjliga särfall i kod är inte menat att behandlas. Men osäkra passager där man har en tydlig åtgärd i åtanke är ett bra användningsområde.

Så vad är en osäker passage i ett program? Typiskt är:

- All filhantering
- Kommunikation med användare
- Kod som annonserar osäkerhet i dokumentationen ("throws exception if…")
- Kod som man upptäcker är osäker (man t.ex. implementerar algoritm som är instabil)

Det finns flera vanliga åtgärder när man fångat ett särfall:

- **Skriv ut felmeddelande och avsluta:** det gör man till exempel om data saknas eller om felet är så allvarligt att det inte finns rimliga sätt att fortsätta.
- **Skriv ut varning, ignorera felet, och fortsätt:** om man har konstiga indata som man kan bortse ifrån, och beräkningarna kan fortsätta ändå.
- **Hantera felet och fortsätt:** fungerar bra för en del förutsägbara fel, om man t.ex. får en dålig inmatning från en användare kan man ju be om ett nytt försök.
- **Skapa ett nytt särfall:** det är lämpligt om du har kod där flera fel kan uppstå, men där felet kan sammanfattas bättre i ett visst sammanhang. Om en algoritm är instabil så kanske `ArithmeticError` är ett bättre fel än `ZeroDivisionError`, eftersom användaren av algoritmen vet algoritmens svaghet men kanske inte kan associera det med division med noll.

Det finns också åtgärder man ska undvika:

- **Ignorera inte felet!** Det värsta man kan ha efter **except** är instruktionen **pass**. Som användare blir det väldigt svårt att veta vad och vart någonting går fel. Egentligen skulle Python lyft ett särfall, men man har som programmerare bestämt att göra ingenting åt detta. Det potentiella felet kan fångas på en helt annan plats i koden.
- För **allvarliga** fel *räcker inte* en felutskrift. Det är inte snyggt att låta ett program fortsätta om felet är så allvarligt att man vet att det kommer uppstå ett annat fel senare. Vissa fel ska helt enkelt avbryta ett program.

5.1.4 Hantera egna stabilitetsproblem

Det är inte nödvändigtvis en svaghet att problem uppstår. Det är svårt, om inte helt omöjligt, att t.ex.:

- Förutse formatet på *all* möjlig indata.
- Se till att *alla* användare (inklusive du själv) kallar funktioner helt rätt.

I Python finns många specifika särfall definierade (se dokumentationen för en detaljerad beskrivning):

- `Exception`
- `ArithmeticError`
- `IOError`
- `IndexError`
- `MemoryError`
- `ZeroDivisionError`
- m.m.fl.

I Python är särfall inte bara en textsträng eller felkod som skrivs, särfallet är en klass (klasser kommer att behandlas i mer detalj senare i kompendiet). Det gör det möjligt att lätt skapa egna särfallsvarianter, som är baserade på de existerande särfallen.

Man kan även lyfta ett generellt särfall med det reserverade ordet **`raise`** (det är härifrån termen "lyfta" kommer). I t.ex. Java och C++, åstadkoms detta med hjälp av `throw`. Om vi t.ex. vill ta ut första elementet ur en lista och kasta ett fel om listan är tom kan vi skriva:

```
def head(xs):
   if not xs:
      raise Exception("head: input list is empty")
   else:
      return xs[0]

try:
    print(head([]))
except Exception as e:
    print(str(e))

print(head([]))
```

```
head: input list is empty
# specific information about where the exception was caught
Exception: head: input list is empty
```

Notera att koden fortsätter efter *första* gången `head` kallas i **`try`**-**`except`**-blocket: endast *strängen* med felmeddelandet skrivs ut. Man kan "spara" särfallet i en variabel som man vidare kan använda (för t.ex. information, eller om man vill lyfta särfallet *efter* man utfört en viss operation som att spara något till en fil) genom **`except`** `Exception as e`, där `e` är identifieraren för det fångade särfallet. Strängen med felmeddelandet fås genom `str(e)`. *Andra* gången `head` kallas så lyfts särfallet som stöts på i funktionen: samma information skrivs ut igen men man ser att det *men* det är `Exception` som sköter särfallet snarare än **`except`**-blocket och programmet *avslutas*.

Nyckelordet **`raise`** är användbart för att skriva mer detaljerade felmeddelanden, eller när ett "fel" uppstår som inte täcks av Pythons felhantering. Ett exempel på ett sådant fel är om en algoritm som vi skrivit måste bibehålla ett värde på en parameter `h` större än `0.01` om algoritmen ska fungera (tänk t.ex. steglängd på x-axeln för att hitta nollställe till polynom). Vi kan då skriva något i stil med

```
def my_algorithm(x, h):
    while True:
        if h < 0.01:
            raise ArithmeticError('Stepsize h should not be smaller than 0.01.\
```

```
                                    The value of x was:', h)

        # check polynomial root
        ...

        # update h
        ...
```

5.1.5 Vad du skriver påverkar hur du jobbar med särfall

Man kan säga att det finns olika scenarier för användning av särfall.

- **Applikationsprogrammering:** Om du skriver ett program som använder kod från olika moduler, såväl egna och som andras, så kommer det finnas funktioner och metoder som genererar särfall. Det blir då viktigt att använda `try-except` för att ta om hand om de särfall som kan uppstå. Programmet ska inte avslutas på grund av problem som kunnat förutses och som det kanske finns enkla lösningar till. Man kan också vilja undvika att användaren av ditt program (du själv eller andra) ska behöva se och tolka felmeddelanden.

  Några exempel:

  - Om ett särfall uppstår på grund av dålig indata, då ska man tala om det på ett tydligt sätt. Att släppa igenom ett felmeddelande från ett särfall är inte användarvänligt, eftersom den egentliga orsaken ofta inte alls förklarar varför särfallet uppstått.
  - Om ett särfall uppstår på grund av, exempelvis, numerisk instabilitet eller osäkerheter i omgivningen (som att en fil med ett visst namn redan finns) så vill man antagligen hantera det på ett bra sätt istället för att slänga ett särfall i ansiktet på användaren.

- **Stödprogrammering:** Det är mycket vanligt att man skriver kod som är stöd till det program man verkligen vill skriva. Om ditt mål är att skriva en egen variant av Matlab så behöver du många funktioner som gör beräkningar; om du vill analysera data lagrade i ett komplicerat format kanske det är bra att ha en modul som just hanterar data så att du senare kan fokusera på beräkningarna. Dessa stödrutiner kan stöta på problem som måste signaleras till användaren och det görs lämpligen genom att använda `raise` till att skapa ett särfall.

Python talar om *att* ett fel uppstått. Du som programmerare kan tala om *varför* ett fel har uppstått.

5.1.6 Vanliga misstag

5.1.6.1 `if`-sats istället för särfall för flödeskontroll

Det kan vara lockande att använda `except` för flödeskontroll, men ofta är det bättre att istället använda `if`-satser. Till exempel i denna kod:

```
def elem_access(xs,i):
    try:
        if i >= len(xs):
            raise IndexError
    except IndexError:
```

```
        print("Index is out of range.")
    return xs[i]
```

Det är ju ingen poäng med att först skapa ett särfall och sedan genast fånga det. Ett annat misstag i just detta exempel är att om det uppstår ett fel, så *fortsätter* programmet efter en felutskrift, här med att försöka komma åt element `i` trots att `i` är för stort. Om man fångar ett särfall så ska man hantera det på något sätt. Det är ofta rimligt att avsluta programmet med ett felmeddelande och `quit()`.

5.1.6.2 Överanvändning

Använd inte särfall för att tackla att kod inte fungerar som du hade tänkt dig.

I exemplet nedan skapas en ny uppslagstabell `d` för nycklar `e`, tagna från en lista, genom att anropa en funktion som använder informationen i uppslagstabellen `data`. Tydligen har programmeraren märkt att alla värden från `selected` inte finns med i `data` och därför skyddat sig genom att fånga de `KeyError`'s som kan uppstå:

```
data = {1 : "Make bed",
        2 : "Shower",
        4 : "Eat breakfast"}
selected = [1,3,2]

d = {}
for e in selected:
    try:
        d[e] = perform_task(data[e])
    except KeyError:
        print()      # Bad line of code!
```

5.2 Listomfattningar

Listomfattning (eng: *list comprehension*) är en teknik för att skapa listor som också är populär bland funktionella programmeringsspråk och därför ofta tas upp i det sammanhanget. Listomfattning är praktiskt för att initiera listor med värden att jobba vidare med. En fördel, som kanske ligger bakom listomfattningars populäritet, är att de kan vara mycket tydliga tack vare sin nära koppling till matematisk notation. Betrakta till exempel mängden av kvadrater av tal upp till `10`:

```
[x**2 for x in range(10)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Man kan också filtrera bort elementen — inte alla behöver presenteras. Till exempel alla udda heltal upp till `20` kan skrivas så här:

```
[x for x in range(20) if x % 2 == 1]
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

(Man kan korta ner det uttrycket lite till, kan du se hur?)

Vi kan även kalla på funktioner som konverterar element enligt något system

```
def transform(x):
    if x % 2 == 0:
        return "Even"
    if x % 3 == 0:
        return True
    else:
        return x
print([transform(x) for x in range(20)])
```

```
['Even', 1, 'Even', True, 'Even', 5, 'Even', 7, 'Even', True]
```

Sammanfattning: en listomfattning har tre delar:

- resultatdelen, med ett uttryck som ger elementen i resultatet,
- generatordelen, där `for` används för att plocka fram värden till resultatdelen, och
- predikatdelen, som bestämmer vilka element som ska presenteras.

Mer komplicerat exempel med nästlad loop:

```
[(a,b) for a in range(2,8) for b in range(2,8) if a % b == 0]
```

```
[(2, 2), (3, 3), (4, 2), (4, 4), (5, 5), (6, 2), (6, 3), (6, 6), (7, 7)]
```

Smidigt för att få alla kombinationer av element i två listor. Notera att utdata är en lista av **tupler**, dvs par av element.

En användbar funktion som tar två listor med samma längd och gör en lista av tupler där man parat ihop elementen på samma index med varandra är `zip` (finns dock redan implementerat i python). Man skriver den lätt med en listomfattning.

```
def zip(xs,ys):
    if len(xs) != len(ys):
        print("zip unequal lengths!")
        return []
    return [ (xs[i],ys[i]) for i in range(len(xs)) ]
```

5.2.1 `dict`-omfattningar

Listomfattning är den traditionella tekniken, men i Python har man generaliserat dessa till uppslagstabeller (och mängder som kommer i senare kapitel). Man kan alltså skapa uppslagstabeller med `dict`-omfattning (eng. `dict`-*comprehensions*). Man måste i dessa fall se till att man skapar både nycklar och värden:

```
my_dict1 = {x: x ** 2 for x in (2, 4, 6)}
my_dict2 = {str(x): x ** 2 for x in range(3)}
my_list = ['1', '42', '13']
my_dict3 = {'key_' + x: int(x) for x in my_list}

print(my_dict1)
print(my_dict2)
print(my_dict3)
```

```
{2: 4, 4: 16, 6: 36}
{'0': 0, '1': 1, '2': 4}
{'key_1': 1, 'key_42': 42, 'key_13': 13}
```

5.3 Uppgifter

1. Vad händer om vi kallar på funktionen `divide_by_elems` (funktionen given i avsnitt 5.1.2.3) med en sträng som argument för `x`? Om strängen kan göras om till ett heltal ska funktionen fortsätta utan att krascha, annars ska den returnera `None`. Lös detta i funktionen med hjälp av **`try`** och **`except`**.

2. Skriv om funktionen `divide_by_elems` så att den inte bryter efter första fel, utan fortsätter att dela talet x med rader som kommer efter. Till exempel för filen `numbers.txt` som ovan ska funktionen returnera listan

   ```
   [2.0, 1.0, 0.5, 0.6666666666666666]
   ```

   Tips: Byt ut raden

   ```
   with open(filename, 'r') as h:
   ```

   med

   ```
   h = open(filename, 'r')
   ```

   D.v.s. skippa **`with`** funktionaliteten. Då blir det lättare med omstruktureringen. Du kan stänga filen med `h.close()`.

3. Givet följande lista

   ```
   xs = [2,-167,12,5676,-14,-7, 12,12,-1]
   ```

   Skapa med hjälp av en listomfattning listan `ys` som innehåller endast de positiva talen i `xs`.

4. Skapa med listomfattning en lista `ys` där alla talen i listan `xs` i uppgift 3 har konverterats till positiva om de är negativa (t.ex. `-167` blir `167`).

5. Givet funktionen nedan

   ```
   def is_prime(x):
       if x >= 2:
           for y in range(2,x):
               if not ( x % y ):
                   return False
       else:
           return False
       return True
   ```

   Skapa en lista med alla primtal under `100` genom att använda en listomfattning.

6. Givet följande lista

```
xs = [2,2,7,9]
```

Skapa med listomfattning en lista `ys` som innehåller produkter från multiplikation av alla parvisa element i `xs`.

7. Skapa med listomfattning en lista `ys` som innehåller produkter från multiplikation av alla parvisa element i `xs` från uppgift 6 utom elementen själva.

   Obs: Talet `2` på index `0` i `xs` ska inte tolkas samma som `2` på index `1`. Du kan alltså inte testa om talen är lika utan måste testa om index är samma.

   Tips: använd `range`.

8. Skriv om labb 1 med exceptions så att programmet inte kraschar om man skriver in något fel.

Kapitel 6

Sekvenser och generatorer

6.1 Sekvenser

Python har stöd för ett antal olika sekvenstyper. Sekvenstyper i Python har gemensamt några vanliga operationer, t.ex. gäller:

- `+`: konkatenering/sammanslagning
- `*`: repetition
- `in` och `not in`: test om ett värde finns/inte finns som element i sekvensen
- `len()`: längden av sekvensen/hur många element
- `min()` och `max()`: minimum och maximum av element (om element-typerna är kompatibla med < och >)
- `[:]`: slicing för åtkomst av element
- m.fl.

Många operationerna ovan känns förmodligen igen från `list`. Mycket riktigt är listor en av sekvenstyperna i Python. De andra typerna vi kommer att titta på i detta kapitel är tupler, `range` och strängar, vilka vi alla stött på tidigare i ett fall eller annat.

Sekvenstyper kan delas in i två underkategorier: *muterbara* och *icke-muterbara*. Med muterbarhet så menas att ett skapat objekt kan *ändras* på efter att det skapats. På samma sätt så är icke-muterbara objekt konstanta. Hur ett objekt används, och vilka potentiella sidoeffekter kod kan ha, kan bero på om ett objekt är muterbart eller inte.

För mer detaljerad information, se Python-dokumentationen om sekvenstyper.

6.1.1 Listor

Listor är en sekvenstyp som är *muterbar*. Det är en viktig egenskap hos listor som betyder att vi kan uppdatera deras värden utan att kopiera listan. Vi har redan sett många exempel på vad som kan göras med listor, men för att poängtera vilken effekt det kan ha att listor är muterbara, betrakta på följande funktionsdefinition och kod:

```
def sum_of_increment(l, x):
    for i in range(len(l)):
        l[i] += x
    ret_sum = sum(l)
    return ret_sum

mylist = [3, 3, 4, 5]
print("mylist is:", mylist)
s = sum_of_increment(mylist, 2)
print("mylist after first call:", mylist)
print("First call returns:", s)
s = sum_of_increment(mylist, 2)
print("mylist after second call:", mylist)
print("Second call returns:", s)
```

```
mylist is: [3, 3, 4, 5]
mylist after first call: [5, 5, 6, 7]
First call returns: 23
mylist after second call: [7, 7, 8, 9]
Second call returns: 31
```

Resultatet blir olika varje gång funktionen kallas, även om funktionsanropet görs med samma variabler! Det här beror på att `mylist` ändras inuti funktionen, vilket går att göra eftersom listor är muterbara. Det här är också möjligt för att Python inte kopierar hela listan till funktionsargumentet (i det här fallet `l`). Detta kan ibland vara användbart, men man bör vara medveten om och försiktig med detta, så att det inte får oanade konsekvenser.

För mer om listor och vilka funktioner som `list`-typen har, se Python-dokumentationen och mer info om listor.

6.1.2 Tupler

Tupler liknar listor, men de är *icke-muterbara*. Så vi kan inte bara uppdatera en tupel som vi vill. Vi har sett tupler förr, även om vi kanske inte explicit skapat variabler av typen tupler: multipla returvärden från *funktioner* returneras nämligen som en tupel med flera element. Syntaxen för tupler är väldigt likt den som listor, med den skillnaden är att man använder `()` för att skapa en tupel istället för `[]`.

```
mytuple = (3, 3, 4, 5)
myothertuple = 3, 3, 4, 5 # equivalent syntax
print(mytuple)
print(mytuple == myothertuple)
print(mytuple[0])
print(mytuple[3:1:-1])
print(len(mytuple))
print(3 in mytuple)

# function returning multiple values
def integer_div(nominator, denominator):
    q = nominator // denominator
```

```
    r = nominator % denominator
    return q, r

divs = integer_div(17, 10)
print(type(divs))

# tuples are immutable
mytuple[0] = 42
```

```
(3, 3, 4, 5)
True
3
(5, 4)
4
True
<class 'tuple'>
TypeError: 'tuple' object does not support item assignment
```

Lägg märke till att slicing av en tupel ger tillbaka en tupel: slicing av en sekvenstyp ger tillbaka samma typ.

Om vi t.ex. skulle köra om kodexemplet från sektionen Listor, med funktionen `sum_of_increment` definierad på samma sätt, fast med en tupel:

```
mytuple = (3, 3, 4, 5)
s = sum_of_increment(mytuple, 2)
```

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

Precis som listor, kan tupler innehålla olika datatyper och vi kan nästla dem:

```
t = (12345, 54321, 'hello!')
mytuple = (t, (1, 2, 3, 4, 5))
print(mytuple)
```

Tupler kan innehålla *muterbara* objekt:

```
t2 = ([1, 2, 3], [3, 2, 1])
print(t2)
```

Det sista exemplet `t2`, kan *inte* användas som nyckel i en uppslagstabell. Jämför t.ex.:

```
d = {t: 'hello'} # works
d2 = {t2: 'goodbye'} # TypeError: unhashable type: 'list'
```

Det här är som diskuterats tidigare för att nyckelvärden måste vara konstanta. I detta fall räcker det alltså inte med att själva tupeln är icke-muterbar eftersom den innehåller muterbara objekt (listor), vilket gör att `t2` *inte* är kan användas som nyckel.

```
# t2[0] = [1, 2, 4] # throws TypeError, immutable
print(t2)
t2[0][0] = 42
print(t2)
```

```
([1, 2, 3], [3, 2, 1])
([42, 2, 3], [3, 2, 1])
```

6.1.3 range

Pythons range är också en *icke-muterbar* sekvenstyp. Den beter sig som en tupel, men den har inte stöd för vissa operationer, t.ex. + och *.

```
myrange = range(10)
print(myrange[3])
print(myrange[2:9:3])
print(4 in myrange)
print(42 in myrange)
myrange + myrange
```

```
3
range(2, 9, 3)
True
False
TypeError: unsupported operand type(s) for +: 'range' and 'range'
```

Återigen, slicing ger här tillbaka en range-typ.

6.1.4 Strängar

Strängar är även de ett exempel på en sekvenstyp, också *icke-muterbara*. Vi har precis som listor använt strängar flitigt. Det finns väldigt många användbara strängfunktioner; de flesta funktionerna har namn som tydligt säger vad de gör. Här nedan ges bara några exempel för vanligt förekommande strängfunktioner:

```
my_string = "Hello world! This is fun"
print(my_string.index("o"))
print(my_string.count("l"))
print(my_string.upper())
print(my_string.lower())
print(my_string.capitalize())
print(my_string[3:7])
print(my_string[3:7:2])
print(my_string[::-1])
print(my_string.startswith("Hello"))
print(my_string.endswith("asdfasdfasdf"))
print(my_string.split())
print(my_string.split("i"))
```

```
4
3
HELLO WORLD! THIS IS FUN
hello world! this is fun
Hello world! this is fun
```

```
lo w
l
nuf si sihT !dlrow olleH
True
False
['Hello', 'world!', 'This', 'is', 'fun']
['Hello world! Th', 's ', 's fun']
H
```

Glöm inte att man in Spyder lätt i t.ex. konsolen kan se vilka funktioner som är associerade med en sträng (eller generellt ett givet objekt) genom att använda sig av `dir()`. T.ex. kan man:

```
dir(my_string)
```

Detta kan ge lite "för mycket" information. Vill man ha en mer interaktivt sätt, kan man i konsolen skriva `my_string.` (lägg märke till punkten) och sen trycka `TAB`.

För mer om strängar och fler strängfunktioner, se Python-dokumentationen.

6.2 Generatorer

Generatorer i Python är itererbara objekt som i varje iteration lämnar tillbaka ett värde till användaren. Vad värdet är beror på hur en generator har definierats. Vi har stött på flera itererbara objekt tidigare, t.ex. är alla sekvenstyper som precis beskrivits i Sekvenser itererbara.

Den största skillnaden mellan generatorer och sekvenstyper (förutom `range`) är när själva beräkningen av elementen sker: detta kan antingen ske *direkt* när man skapar objektet eller först när värdet på elementet *behövs*, d.v.s. när man väl utför en loop. Generatorer är ett exempel på en typ där värdet av ett element beräknas först när det behövs, detta är så kallad *lat evaluering*.

6.2.1 Lat och strikt evaluering

Om alla värden i ett objekt beräknas när objektet skapas kallar man det *strikt* eller *ivrig evaluering/beräkning* (eng. *eager evaluation*). Motsatsen är *lat* eller *selektiv evaluering* (eng. *lazy evaluation*) där värden beräknas när de behövs. Ett exempel med sekvenstyper som visar skillnaden:

```
mylist = [0, 1, 2, 3, 4, 5]
myrange = range(6)
print("mylist: ", mylist)
print("myrange: ", myrange)
```

```
[0, 1, 2, 3, 4, 5]
range(0, 6)
```

Här har vi skapat en lista `mylist`, som är ett exempel på strikt evaluering: alla värden som listan innehåller är här beräknade och sparade i minnet som utskriften visar. Utskriften av `range` å andra sidan visar bara just det, att det är ett objekt av typen `range` skapat med inparametrar `0, 6`. `range` är ett exempel på lat evaluering: alla värden som `myrange` innehåller *beräknas* varje gång man t.ex. gör `myrange[0]`. Lat evaluering kommer väl till hands när man använder sig av iteration, då det är mycket mer *minneseffektivt* än att iterera över ett objekt som är helt sparat i minnet.

`range` är lite av ett specialfall jämfört med de andra sekvenstyperna, just eftersom det är lat evaluering. Den finns för att det är en så vanlig struktur att ha en sekvens av ökande tal. Om man vill använda sig av lat evaluering, fast för mer komplicerade strukturer, så kommer generatorer väl till hands.

6.2.2 Generatorfunktioner

I Python kan man skapa en generatorfunktion på ett väldigt likt sätt som man skapar en vanlig funktion. När en vanlig funktion anropas, så exekveras all kod i funktionen fram tills att en `return`-sats stöts på, då ett värde (potentiellt) lämnas tillbaka till användaren och avslutar funktionskörningen. En generatorfunktion returnerar ett värde med **`yield`**, men funktionens *tillstånd sparas*. Nästa gång funktionen anropas så fortsätter funktionen från där den var sist tills *nästa* gång en **`yield`**-sats stöts på, vid vilken den lämnar tillbaka värdet och "pausar".

Vi använder här funktionsanrop för generatorfunktioner lite löst. Egentligen kallas själva generatorfunktionen endas *en gång* och ger då tillbaka en generator. Det är denna generator som när den itereras över varje gång exekverar koden i generatorfunktionen.

Ett exempel visar nog detta bättre:

```
def inf_gen(start=0, step_size=1):
    """Infinitely generate ascending numbers"""
    while True:
        yield start
        start += step_size
```

Detta är en oändlig uppräkning av tal: en "oändlig `range`". Hade vi försökt att använda oss av en lista, så inser man snabbt att det är svårt att skapa ett objekt med *oändligt* många element: vi hade fått slut på minne på datorn. Men representeras det som en generator är detta inget problem och vi kan fortfarande iterera över den. För att komma åt underliggande generator som man kan iterera över anropar man generatorfunktionen:

```
my_inf_gen = inf_gen(10, 1)
print(my_inf_gen)
print(next(my_inf_gen))
print(next(my_inf_gen))
print(next(my_inf_gen))
```

```
<generator object inf_gen at 0x7f1d7855be40>
10
11
12
```

Här har vi använt oss av `next`-funktionen med generatorn `my_inf_gen` som argument för att stega fram nästa värde. Det är faktiskt implicit `next` som anropas om man skulle iterera över `my_inf_gen` genom att t.ex. göra **`for`** `i` **`in`** `my_inf_gen: print(i)`. Vi kan inte göra det här, eftersom det skulle fortsätta i oändlighet, men testa gärna själv för att se vad som händer. (För att avsluta en oändlig loop kan man trycka `CTRL+c`)

Generatorer behöver självklart inte vara oändligt itererbara (som ovan). Som du ser kan du bestämma själv hur komplicerad en generator ska bli med nästan samma syntax som för en vanlig funktion. Iterationen över en generator avslutas funktionskroppen körs utan att stöta på ett **`yield`**. Precis som man kan ha flera **`return`**-satser i en funktion, kan man ha flera **`yield`** i en generatorfunktion. Skillnaden är att

funktionsblocket fortsätter från där det slutade vid senaste **yield**. En viktig detalj, och skillnad mot range, är att en generator bara kan itereras över *en gång*.

Vi illustrerar ovan koncept med en enkel generatorfunktion:

```
def my_generator(stop=5):
    i = 0
    j = 0
    while i + j < stop:
        print("i: ", i, "j:", j)
        i += 1
        yield i + j
        print("i: ", i, "j:", j)
        i += 1
        j += 1
        yield i + j
        print("i: ", i, "j:", j)

it = 1
gen = my_generator()
for i in gen:
    print("Iteration: ", it, "gen returns: ", i)
    it += 1

# loop a second time
for i in gen:
    print("Will this be printed???")
```

```
i:  0 j: 0
Iteration:  1 gen returns:  1
i:  1 j: 0
Iteration:  2 gen returns:  3
i:  2 j: 1
i:  2 j: 1
Iteration:  3 gen returns:  4
i:  3 j: 1
Iteration:  4 gen returns:  6
i:  4 j: 2
```

6.2.3 Generatoruttryck

Generatorer kan även skrivas på ett liknande sätt som listomfattningar. Det skapas då ingen lista utan istället får man en generator som man kan arbeta vidare med, framförallt iterera över med olika verktyg. Här skapas en lista och en generator.

```
big_list = [x**2 for x in range(10**8)]
big_generator = (x**2 for x in range(10**8))
print("Size of big_list in memory:", big_list.__sizeof__())
print("Size of big_generator in memory:", big_generator.__sizeof__())
```

```
Size of big_list in memory: 859724448
Size of big_generator in memory: 96
```

Som vi ser har listomfattningen genererat hela listan i minnet på datorn. Detta illustrerar vikten av att använda sig utav generatorer när vi ska iterera över många värden.

6.2.4 Exempel: Generator för alla primtal

Generatorer är smidiga att använda till vissa saker, bland annat oändliga sekvenser.

Låt oss säga att vi vill beräkna alla primtalsdelare till ett tal `x`. Vi kan då skapa en generator med alla primtal och se hur många gånger varje primtal delar `x`:

```
def is_prime(x):
    for t in range(2, x):
        if x % t == 0:
            return False
    return True


def prime_divs(x):
    primes = (i for i in inf_gen(2) if is_prime(i))
    ps = []
    for p in primes:
        if p > x:
            return ps
        else:
            while x % p == 0:
                ps.append(p)
                x //= p

print(prime_divs(2))
print(prime_divs(3))
print(prime_divs(24))
print(prime_divs(25))
print(prime_divs(232))
```

```
[2]
[3]
[2, 2, 2, 3]
[5, 5]
[2, 2, 2, 29]
```

6.3 Uppgifter

1. Vi såg att funktionen `sum_of_increment(l, x)` ändrade i vår lista när vi kallade på den flera gånger. Vi såg även att vi inte kunde kalla på `sum_of_increment(l, x)` med en tupel. Kan du av detta dra slutsats när det kan vara bra att använda sig av tupler?

2. Skriv en funktion `is_palindrome` som tar en sträng som input, och returnerar `True` om strängen är ett palindrom (d.v.s. om det är samma sak framlänges som baklänges) och returnerar den omvända strängen om det inte är ett palindrom.

   Tips: använd sträng-slicing.

3. Fungerar `is_palindrome` som du skrev i uppgift 2 för listor och tupler? Varför?

4. Skriv en funktion `occurrences` som beräknar hur många gånger bokstäver förekommer i en sträng. Funktionen ska returnera en uppslagstabell med bokstäverna som nycklar och antal gånger bokstäverna förekommer i strängen. Funktionen ska inte göra skillnad på versaler och gemener, utan räkna dessa gemensamt (för en given bokstav). Alla tecken som *inte* är bokstäver ska räknas under nyckeln `'non_alphas'`, och tecken som *inte* finns med i strängen ska inte finnas med i uppslagstabellen.

   Tips: använd dig av de inbyggda strängfunktionerna. Se till exempel Python-dokumentation för strängfunktioner

   Exempel:

```
s = 'Hello my very happy friend! Is the sun shining?!'
uo = occurrences(s)
print(s)
for k, v in uo.items():
    print(k, ":", v)

s = 'JjJJJjj hHhh !/:;(+[//=--*])'
uo = occurrences(s)
print("\n" + s)
for k, v in uo.items():
    print(k, ":", v)
```

```
Hello my very happy friend! Is the sun shining?!
h : 4
e : 4
l : 2
o : 1
non_alphas : 11
m : 1
y : 3
v : 1
r : 2
a : 1
p : 2
f : 1
i : 4
n : 4
d : 1
s : 3
t : 1
u : 1
g : 1
```

```
JjJJJjj hHhh !/:;(+[//=--*])
j : 7
non_alphas : 17
h : 4
```

5. Använd `occurrences(s)` ovan för att skriva ett program som läser in en fil och skriver ut bokstäverna som nycklar och antalet gånger bokstäverna förekommer för hela filen. För att testa kan ni använda filen `palindrome.txt` med innehåll:

   ```
   not palindrome line
   palindrome line enil emordnilap
   no way
   hello ! olleh
   Tomorrow will be a glorious day!!!
   oooooooooooooooooooooooooo
   rust is also a programming language
   ooooooooooooioooooooooooo
   ```

   Tips: läs in hela filinnehållet som en sträng.

6. Skriv en generatorfunktion `fibonacci(stop=x)` som genererar Fibonacci tal (se https://sv.wikipedia.org/wiki/Fibonaccital) mindre eller lika med `x`.

   Tips: Använd dig utav strukturen i funktionen `my_generator` ovan.

7. I första raden i funktionen `prime_divs` i Exempel: generator för alla primtal så använde vi oss av ett generatoruttryck för att "filtrera" alla tal ur en generator som var primtal i en ny generator. Man kan för sin filter-generator alltså skriva:

   ```
   filter_gen = (i for i in iterable if function(i))
   ```

   Här är `function` en (användardefinierad) funktion som ger tillbaka `True` eller `False` när `i` ges som argument, och `iterable` är ett itererbart objekt (t.ex. en generator som vi hade i `prime_divs`). Alla `i` som ger `function(i) == True` kommer vara en del av `filter_gen`. Python har redan implementerad detta i funktionen `filter`, som du kan läsa dokumentationen om. En liknande funktion är `map` (se dokumentationen) där generatorn som skapas bör innehålla *värdena* som en funktion `f` *returnerar* när `f` appliceras på alla element i ett itererbart objekt.

   Skriv en egen funktion `map_gen` som åstadkommer detta genom att ta en *godtycklig* **funktion** som argument och ett itererbart objekt, och som *returnerar* en generator.

   Exempel:

   ```
   gen = map_gen(int, '123456')
   sum(gen)
   ```

   ```
   21
   ```

8. Skriv en funktion `palindrome_rows` som tar ett filnamn som argument, och kollar varje rad i filen om det är ett palindrom eller inte. `palindrome_rows` ska resultera i en generator som man kan iterera över när funktionen anropas.

   Tips: tänk på att rader läses in *inklusive* ny-rad-tecken. Modifiera `is_palindrome` från uppgift 2 så att den bara lämnar tillbaka `True` eller `False` baserat på om strängen är ett palindrom (alltså inte lämnar tillbaka själva strängen baklänges om det *inte* är ett palindrom).

Exempel med filen `palindrome.txt` ovan:

```
p_rows = palindrome_rows('palindrome.txt')
i = 1
for p in p_rows:
    print("Row", i, "palindrome?", p)
    i += 1
```

```
Row 1 palindrome? False
Row 2 palindrome? True
Row 3 palindrome? False
Row 4 palindrome? True
Row 5 palindrome? False
Row 6 palindrome? False
Row 7 palindrome? False
Row 8 palindrome? True
```

9. Du "vet" att det i filen `palindrome.txt` gömmer sig *ett enda* versalt ord, men du vet inte vilket det är. Det här tycker du är extremt jobbigt! Använd dig av `open` tillsammans med strängfunktioner för att *utan en loop* ta reda på vilket ord det är. Du kan anta att ett "ord" avgränsas av mellanrum.

   Tips: någon av `strip`-funktionerna tillsammans med indexering kan komma till användning här.

Kapitel 7

Moduler, bibliotek och programstruktur

7.1 Moduler

Många programmeringsspråk har ett "modulsystem", så även Python. Dessa är smidiga när man skriver större program och egna bibliotek. De kan användas för:

- Kodåtervinning – importera kod från andra (och dig själv).
- Struktur – svårt att arbeta med kod i en enda fil, så sprid koden över flera filer.
- Samarbete – svårt samarbeta med kod i en enda fil, lättare att arbeta på separata filer om man är ett team istället för att alla ska ändra på samma fil.

Med hjälp av moduler kan du börja förstå ett stort projekt som bestående av självständiga enheter. Du ska inte behöva förstå alla detaljer i en modul, det ska räcka att förstå dess *gränssnitt*. Gränssnittet är de funktioner och variabler som är tänkta att vara synliga för andra programmera som använder sig av modulen.

Stora program blir snabbt oöverskådliga. Väl valda moduler kan då bli viktiga för att bättre förstå ditt och andras program.

En teknisk fördel med moduler är att de hjälper dig att hantera *namnrymden* (eng. *name space*). Många kan vilja använda samma identifierare för t.ex. funktioner i ett stort program (exempelvis `sort`). Moduler döljer implementationen och variabel-/funktionsnamn.

Principer:

- Varje Python-fil är en modul.
- Allt som ska användas (från en annan fil) måste *importeras*.

7.1.1 Import av moduler

Man kan importera moduler på ett antal olika sätt:

- `import modulename`: Tillåter att funktioner/variabler används med punktnotation (dvs allt från modulen är importerat "qualified").

```
import math
math.ceil(3.5)  # Avrunda uppåt
ceil(4.4)  # Fel!
```

- `from modulename import f1, f2, ...`: Tillåter användning av `f1` och `f2` i ditt program utan punktnotation.

```
from math import ceil
ceil(3.5)
```

- `from modulename import *`: Gör allt från `modulename` tillgängligt utan punktnotation.

  Varning: detta anses orsaka dålig läsbarhet och det blir även svårt att veta vilka delar av en modul en annan modul beror på.

  Obs: identifierare som börjar med _ importeras inte. De anses "privata" till modulen.

  **Notera:*x* vissa moduler bygger på att man `*`-importerar, då de *behöver* (eller är skrivna så att) skriva över inbyggda funktioner i Python för att fungera.

- `import modulename as X`: Lättare hantera långa modulnamn när man använder punktnotation. Exempel:

```
import math as M
M.ceil(3.4)     # Praktiskt för långa modulnamn
```

- `from modulename import X as Y`: För att undvika namnkrockar. Exempel:

```
from math import gcd as libgcd

def gcd(a,b):
    r = a % b
    if r == 0:
        return b
    else:
        return gcd(b, r)

for a in range(2,10):
    for b in range(2,a):
        print(gcd(a,b) == libgcd(a,b))
```

- Moduler kan grupperas i *paket* (eng. *packages*), vilka ska förstås som hierarkiskt ordnade grupper av moduler.

```
import os.path as OS
if OS.isfile('data.txt'):
    ...
```

7.1.2 Skriva egna moduler

Låt oss säga att vi har följande Python kod i en fil `circle.py`:

```
pi = 3.14

def area(radius):
    return pi * (radius ** 2)

def circumference(radius):
    return 2 * pi * radius
```

Vi kan då importera den i en annan fil:

```
import circle

print(circle.pi)
print(circle.area(2))

import circle as C

print(C.pi == circle.pi)

tau = C.pi * 2

def circumference(radius):
    return tau * radius

print(C.circumference(2) == circumference(2))
```

Notera: filen `circle.py` måste hittas av Python när programmet letar efter potentiella moduler att importera. Den första platsen Python letar på är i samma mapp som det program/`.py`-fil man kör ligger i; det enklaste sättet är därför att ha sin modulfil (här `circle.py`) i *samma* mapp som huvudprogrammmet/filen som importerar modulen.

7.1.3 `main()`-funktioner

Ett vanligt idiom:

```
print("Hello")

def main():
    print("python main function")

if __name__ == '__main__':
    main()

print("__name__ value: ", __name__)
```

Om och endast om man startar programmet med denna fil så anropas `main`. Om man använder filen som en modul så körs inte `main`. Du kan alltså skriva moduler som innehåller exempelprogram och/eller testkod som inte påverkar vad som händer vid `import`. Anledningen att `main` inte körs om man importerar filen som en modul är att variabeln `__name__` sätts till modulnamnet, så `if`-satsen kommer då inte att köras.

7.2 Populära bibliotek och moduler

Python kommer laddat med batterier: det finns massor av bra och användbara moduler som är en del av Python-implementationen. Dessa moduler kallas tillsammans Pythons standardbibliotek. För en lång lista samt dokumentation se: https://docs.python.org/3/library/.

- `math` – Matematik: trigonometri, logaritmer, osv.
- `sys` – Avancerad filhantering, med mera…
- `os` – Undvik att vara beroende av Win/Max/Linux (typ undvik problem med "/" i filnamn i Linux/Mac vs. "\" i filnamn för Windows).
- `argparse` – Används för parametrar när man startar program på kommandoraden.
- `itertools` – Bra för avancerade och snabba iterationer.
- `functools` – Fler högre ordningens funktioner.
- `datetime` – Datum och tidstyper.
- `pickle` – Serialisering av Python objekt.
- `csv` – Det är svårare än du tror att läsa en komma-separerad fil!
- `json` – Spara data i JSON format för att lätt kunna läsa in igen.
- `matplotlib` – Visualisera data.
- `numpy` och `scipy` – för beräkningar
- …: Utforska själv!

7.2.1 JSON och serialisering

Problem: Jobbigt att skriva och läsa filer! Särskilt med stora datastrukturer (tänk: nästlade listor och/eller uppslagstabeller).

Lösning: *serialisering*

Idé: Dumpa datastruktur på fil i standardformat som lätt kan läsas in igen.

Finns standard pythonmoduler för detta (`marshal` och `pickle`).

Bra portabel lösning: JSON – JavaScript Object Notation

```
import json
legends = [ ['Lovelace', 'Ada', 1815]
          , ['Babbage', 'Charles', 1791]
          , ['Beurling', 'Arne', 1905]]

# Skapa datafil:
with open('legends.json', 'w') as outfile:
    outfile.write(json.dumps(legends))

# Senare... Skriv ut födelsesår:
with open('legends.json', 'r') as data_h:
```

```
    external_data = json.load(data_h)
    for x in external_data:
        print(x[2])            # Print birth year
```

7.2.2 NumPy: för vektorer och matriser

Med numpy (https://www.numpy.org/) får du ett paket för vetenskapliga beräkningar i Python. Det kommer med mycket effektiv lagring av vektorer och matriser, m.h.a. "arrays". Pythons `list` har gränssnitt som (bl.a.) en array, men är mycket mer flexibelt. Det kostar dock i effektivitet. Arrayer i numpy är närmare datorns egen representation, vilket gör beräkningar mer effektiva (bättre användning av minne och beräkningstid). Effektiviteten kommer på bekostnad av flexibilitet, men i många tillämpningar är det mer än väl värt.

Skapa vektorer och matriser med numpy-arrayer:

```
import numpy as np        # idiomatisk Python att importera numpy som np

a = np.array([1,2,3,4])   # Från vanlig lista
b = np.ones((2,2))        # 2x2 enhetsmatris, notera tupeln!
c = np.zeros((10,10))     # 10x10 noll-matris
d = np.linspace(0, 1, 16) # 16 element från 0 till 1, flyttal!
e = np.reshape(d, (4,4))  # Ger 4x4 matris med tal från 0 till 1

print(a)
print(b)
print(c)
print(d)
print(e)
```

```
[1 2 3 4]
[[1. 1.]
 [1. 1.]]
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
[0.         0.06666667 0.13333333 0.2        0.26666667 0.33333333
 0.4        0.46666667 0.53333333 0.6        0.66666667 0.73333333
 0.8        0.86666667 0.93333333 1.        ]
[[0.         0.06666667 0.13333333 0.2       ]
 [0.26666667 0.33333333 0.4        0.46666667]
 [0.53333333 0.6        0.66666667 0.73333333]
 [0.8        0.86666667 0.93333333 1.        ]]
```

Notera att `np` är ett etablerat alias, och `import` görs oftast som i exemplet, inklusive i `numpy`-dokumentationen.

Tips: prova exempelvis `help(np.linspace)` för att se hur den fungerar.

Man har naturligtvis tillgång till operationer på matriser, som t.ex. matrismultiplikation. Prova gärna:

```
m = np.array([[1,2], [3,4]])
v = np.array([1, 0.5])
product = np.matmul(m, v)
```

```
[[2. 5.]]
```

`numpy` betraktar generöst en array som en vektor på ett sätt som passar in. Här ovan tolkas alltså `v` som en kolonnvektor, och om man istället skriver `np.matmul(v, m)` tolkas den som radvektor.

Det finns en särskild datastruktur för matriser definierad i numpy, men den bör man idag undvika eftersom den planeras att plockas bort: rekommendationen är att alltid använda `np.array`.

`numpy` är en del av `scipy` (https://www.scipy.org/) som innehåller en massa olika paket för att göra vetenskapliga beräkningar och hantera "stor" data med Python.

7.3 Tumregler för programstruktur

Vi ska nu diskutera lite olika koncept som gör kod "bättre" och mer läsbar.

7.3.1 Välj informativa identifierare

- Det ska framgå av identifieraren vad en variabel lagrar eller vad en funktion är till för.
  - Krävs en kommentar för att förklara identifieraren? Dåligt. Kan du använda ord från kommentaren som identifierare?
    Som exempel, om du har raden

    ```
    x = 17  # critical value, abort if exceeding
    ```

    så är det nog bättre att byta `x` mot något annat, kanske så här:

    ```
    critical_value = 17  # abort if exceeding
    ```

    Ett till exempel. Istället för:

    ```
    nc = 1.7       # normalization constant
    ```

    bör man skriva:

    ```
    normalization_constant = 1.7
    ```

    Här behövs det inte en kommentar.
- Undvik generiska identifierare. Använd identifierare som är beskrivande.
  - *Dåliga*: `start`, `compute`, `f`
  - *Bra*: `remove_outliers`, `newton_raphson`, `compute_integral`
- Kod ändras. Om funkionens syfte ändras så bör du byta identifierare!
  - Om din funktionen heter `print_integral` men inte gör några
  - utskrifter så är det förvirrande.
- Det finns tillfällen när funktionsnamnen får vara oinformativa:

- `main`, pga konventioner

7.3.2 En funktion ska göra en sak och göra det bra.

Exempel: läsa in flera dataset och göra beräkningar på vart och ett och skriva ut sammanfattning?

- En funktion för inläsning av en fil.
- En funktion som gör beräkningen på ett dataset.
- En funktion som loopar över indata-filerna (kanske huvudprogrammet?) och sammanfattar.

7.3.3 Skilj på funktioner som beräknar och som interagerar!

Interaktion är såväl enkla utskrifter som frågor till användaren

Du vill ofta återanvända beräknande funktioner, men inte dess diagnostiska utskrifter

7.3.4 En funktion bör bara bero av sina parametrar

Man ska kunna förstå en funktion utan att titta på andra delar av ett program.

Att använda globala variabler är olämpligt (men försvarbart i vissa fall)

Det är direkt olämpligt att manipulera globala variabler (dvs använda nyckelordet `global`)

7.3.5 En funktion bör returnera något

Det finns många tillfällen som en funktion inte behöver returnera något (utskrifter, listmanipulation, osv), men förväntningen bör ändå vara att funktioner returnerar något. Ett vanligt nybörjarfel är att man använder globala variabler för att hantera dataflödet i sitt program och då "behövs inte" returvärden. För nybörjaren är alltså brist på returvärden då en indikation på dålig programstruktur.

7.3.6 Tänk på vad du returnerar

Returnera inte numeriska värden som strängar även om du vet att de ska hamna i en sträng senare. Det är mer framtidssäkert att returnera numeriska värden som numeriska värden. Returnera inte "magiska" värden för att markera fel eller problem. Använd hellre särfall. Exempelvis: "return 1000000" är dålig markör för "konvergerade inte".

- Undantag: `None` är ett acceptabelt magiskt värde

Multipla returvärden är praktiska! Bättre än strängar som innehåller flera värden.

7.3.7 En funktion ska få plats på en skärmsida

Mycket att göra? Bryt upp i smådelar.

Det är inte bara för undvika upprepning som man definierar funktioner. De ger också struktur.

7.3.8 Indikationer på problem

7.3.8.1 Djup indentering indikerar onödigt komplicerad implementation

En vanlig orsak till djup indentering är att man har flera olika logiska fall—skapa då funktioner för de olika fallen. På så vis blir fallanalysen tydligare och de olika funktionsanropen indikerar vad som sker i fallen.

Här är ett exempel på "dålig" kod:

```
# prints collatz sequences for 100 consecutive integers
for i in range(1000,1100):
    n = i
    collatz_seq = []
    while n != 1:
        collatz_seq.append(n)
        if n % 2 == 0:
            n = n/2
        else:
            n = 3*n + 1
    print(i, collatz_seq)
```

Detta kodblock är indenterat i tre nivåer. Det är inte så farligt, men man kan strukturera det lite bättre:

```
def collatz(n):
    """Computes the collatz sequence for an integer
    Parameters:
    n (int): input number
    Returns:
    list: list of numbers in collatz sequence for n
    """
    collatz_seq = []
    while n != 1:
        collatz_seq.append(n)
        if n % 2 == 0:
            n = n/2
        else:
            n = 3*n + 1
    return collatz_seq

for i in range(1000,1100):
    collatz_seq = collatz(i)
    print(i, collatz_seq)
```

Här har vi brutit ut en särskild funktion, `collatz(n)`, som används i en `for`-loop, och den funktionen har bara två indenteringsnivåer (`while`-satsen och `if`-satsen).

Vi kan ta det ett steg till och lägga uppdateringen av variabeln `n` till en egen funktion:

```
def collatz(n):
    """Computes the collatz sequence for an integer
    Parameters:
```

```
    n (int): input number
    Returns:
    list: list of numbers in collatz sequence for n
    """
    collatz_seq = []
    while n != 1:
        collatz_seq.append(n)
        n = collatz_update(n)
    return collatz_seq

def collatz_update(n):
    if n % 2 == 0:
        return n/2
    else:
        return 3*n + 1

for i in range(1000,1100):
    collatz_seq = collatz(i)
    print(i, collatz_seq)
```

Det är inte alls självklart att den här sista versionen är lämpligare. Vilken version tycker du är lättast att läsa och förstå?

7.3.8.2 Funktioner saknar parametrar?

Funktioners syfte är vanligen att ta några data och beräkna något. Beräkningen behöver vara inte vara matematisk utan kan vara enkel behandling av data. Det är mindre vanligt att funktioner helt saknar parametrar, men det finns förstås legitima exempel, till exempel för utskrift av en meny eller initialisering av datastrukturer. Hos nybörjare är det dock inte ovanligt att man finner det bekvämt att lägga data i en eller flera globala variabler som funktioner sedan arbetar mot. En sån konstruktion gör det dock svårt att se hur funktioner beror av varandra; man måste läsa funktionerna noggrant för att identifiera programflödet. **Sedan kan man heller inte använda funktionen i en annan modul om den inte tar någon parameter!**

Nedan skriven kod fungerar om vi lägger `is_prime` i annan modul `m` (dvs fil) och kallar på den med `m.is_prime(x)`.

```
def is_prime(x):
    if x >= 2:
        for y in range(2,x):
            if not ( x % y ):
                return False
    else:
        return False
    return True

def main(x):
    print("The number is prime:", is_prime(x))
```

```
x = int(input("Provide a number to analyze: \n"))
main(x)
```

Nedan (dåligt!) skriven kod fungerar endast om vi har globala variabeln x i samma fil, vilket gör att vi inte kan importera funktionen.

```
def is_prime():
    if x >= 2:
        for y in range(2,x):
            if not ( x % y ):
                return False
    else:
        return False
    return True

def main(x):
    print("The number is prime:", is_prime())

x = int(input("Provide a number to analyze: \n"))
main(x)
```

7.3.9 Programmeringsriktlinjer

- Varje funktion ska få plats på en halv laptopskärm med fontstorlek 12.
- Varje funktion ska ha en dokumentationssträng.
- En funktion som gör en beräkning får inte ha utskrifter. Separera alltid interaktion och beräkning.

7.4 Uppgifter

1. Importera biblioteket `random` (se https://docs.python.org/3/library/random.html) för att generera 10 slumpnässiga *heltal* mellan 1 och 100 och 10 slumpnässiga *flyttal* mellan 1 och 100.

   Tips: Läs dokumentationen. Vilka funktioner är lämpligast att använda?

2. Betrakta koden nedan som läser in ett tal och avgör om det är ett perfekt tal eller ett primtal. Skriv om koden så att den får bättre struktur.

   Tips: Dela upp koden i funktioner. Vilka bitar i koden tenderar att bli naturliga funktioner? Vad bör funktionerna ta för parametrar och returnera för att vara återanvändbara utanför modulen? Ha gärna en huvudfunktion `main(x)` som tar talet `x` som parameter.

   ```
   # checks if a number is (1) perfect (2) prime
   x = int(input("Provide a number to analyze: \n"))
   sum = 0
   for i in range(1, x):
       if(x % i == 0):
           sum = sum + i
   ```

```
if x >= 2:
    prime = True
    for y in range(2,x):
        if not ( x % y ):
            prime = False
else:
    prime = False

if sum == x:
    print("The number is perfect")
else:
    print("The number is not perfect")
if prime:
    print("The number is prime")
else:
    print("The number is not prime")
```

3. Lägg till i ditt omstrukturerade program från uppgift 2 valet att låta användaren bestämma om hen ska ange ett tal eller om programmet ska slumpa fram ett tal. Var i programmet bör vi placera denna nya kod?

4. Nedan kod läser in en DNA-sträng (t.ex. `'AGCTAGCGGTAGC'`) och letar först upp den vanligast förekommande strängen av längd `k`. Sedan skriver programmet ut alla parvisa avstånd mellan delsträngen på vår DNA-sträng. Till exempel för `k = 3` returnerar koden nedan `[4, 6]` då det är 4 respektive 6 nukleotider mellan startpositionerna för den vanligast förekommande delsekvensen `'AGC'`.

   Uppgift: Skriv om programmet så att det blir mer läsbart. Vilka bitar i koden utgör lämpliga funktioner? Se till så att de både tar lämpliga parametrar och returnerar lämpliga datastukturer. Funktionerna ska kunna kallas på individuellt från annan modul.

```
# Given a sequence as input find distances between the copies of
# the most frequent substring

seq = input("Enter a dna string: ")
k = 3
substrings = {}
for i in range(len(seq) - k + 1):
    substring = seq[i: i+k]
    if substring in substrings:
        substrings[substring] += 1
    else:
        substrings[substring] = 1

max_count = 0
most_freq_substring = ""
for substring, count in substrings.items():
    if count > max_count:
        most_freq_substring = substring
        max_count = count
```

```
positions = []
for i in range(len(seq) - k + 1):
    substring = seq[i: i+k]
    if substring == most_freq_substring:
        positions.append(i)

distances = []
for p1,p2 in zip(positions[:-1], positions[1:]):
    distances.append(p2 - p1)

print(distances)
```

5. † Du har fem nummer i en lista `[13, 24, 42, 66, 78]` och vill beräkna alla möjliga sätt ett nummer kan skrivas som en "direkt" sammanslagning av 2 element ur listan; med en "direkt sammanslagning" av två element (heltal) menas här t.ex. `13, 78 -> 1378` och `66, 42 -> 6642`. Dvs, du vill ta reda på alla permutationer av två element ur listan så och slå samman varje permutation till ett tal.

   Tips: Titta på standardmodulen `itertools` för permutationerna. För att slå samman heltal, försöka att först explicit typomvandla talen i listan till `str`, sen konkatenera permutationerna m.h.a strängfunktionen `.join()` och till sista typomvandla tillbaka till `int`.

   Utdata bör bli (om alla tal sparats i en lista som man printar):

```
[1324, 1342, 1366, 1378, 2413, 2442, 2466, 2478, 4213, 4224, 4266, 4278, 6613,
 6624, 6642, 6678, 7813, 7824, 7842, 7866]
```

6. Skriv om laboration 1 (temperaturkonvertering) så att funktionerna hamnar i separat modul med namnet `temp_functions.py` och dokumentera funktionerna med en dokumentationssträng (se kapitel 2). Huvudprogrammet ska importera funktionerna i `temp_functions.py` samt hantera fel med **`try`**/**`except`**.

Kapitel 8

Funktionell programmering

I detta kapitel ska vi titta på *funktionell* programmering i Python. Detta sätt att programmera på bygger på att man anropar, sätter ihop och modifiera funktioner. På detta sätt kan vi göra mycket av vad vi redan sett i de andra kapitlen, men endast genom att hantera funktioner. T.ex. kan vi byta ut loopar mot *rekursiva* funktioner, dvs funktioner som anropar sig själva.

8.1 Rekursion

Funktioner kan anropa sig själva och gör de det kallas de för *rekursiva*. Det finns många exempel på detta i matematiken, men även inom programmering kan detta vara väldigt användbart då det ofta leder till väldigt eleganta och korta program. Det kan dock vara svårt att börja "tänka rekursivt" och se hur man kan lösa olika problem genom rekursion. Ett bra sätt att göra det är att ta program man skrivit med hjälp av loopar och göra om dem med rekursion istället.

8.1.1 Fakultet

Ett klassiskt exempel på en rekursiv funktion från matematiken är fakultet. Detta skrivs $n!$ och beräknas genom att man tar produkten av alla tal från 1 till n. Exempel:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Detta kan beskrivas rekursivt, dvs i termer av sig själv, genom:

$$\begin{aligned} 5! &= 5 * 4! \\ &= 5 * 4 * 3! \\ &= 5 * 4 * 3 * 2! \\ &= 5 * 4 * 3 * 2 * 1! \\ &= 5 * 4 * 3 * 2 * 1 * 0! \\ &= 5 * 4 * 3 * 2 * 1 * 1 \\ &= 120 \end{aligned}$$

Detta är en rekursiv beräkning med *basfall* $0! = 1$. En matematiker skriver detta som:

$$n! = \begin{cases} 1 & \text{om n är 0} \\ n * (n-1)! & \text{annars} \end{cases}$$

I Python skriver vi detta genom:

```
def fac(n):
   if n == 0:
      return 1
   else:
      return n * fac(n - 1)
```

Fallet `n == 0` kallas *basfall*. Det är här som rekursionen stannar och börjar nystas upp. Det andra fallet kallas *rekursivt steg* och går ut på att göra ett "enklare" anrop, där problemet i någon mening har gjorts mindre.

Denna funktion anropas precis på samma sätt som vanligt:

```
print(fac(5))
```

```
120
```

8.1.2 Fibonacci talen

Varje kurs som tar upp rekursion måste ha med Fibonacci talen som exempel. Detta är en talserie som ser ut på följande sätt:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Det n:te Fibonacci talet, $F(n)$, är summan av de två tidigare Fibonacci talen. Denna sekvens har två basfall, $F(0) = 0$ och $F(1) = 1$. Därför kan vi skriva

$$F(n) = \begin{cases} 0 & \text{om n är 0} \\ 1 & \text{om n är 1} \\ F(n-1) + F(n-2) & \text{annars} \end{cases}$$

Detta är tydligt en rekursiv definition och vi kan implementera den genom:

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

for i in range(10):
    print(fib(i))
```

```
0
1
1
2
3
5
8
13
21
34
```

Du kan jämföra denna implementation av en funktion som beräknar Fibonacci-tal med den i uppgift 6 i Kapitel 6, som använder sig av en generator.

8.1.3 Största gemensamma delare, rekursivt

Minns Euklides algoritm för största gemensamma delare för två tal `a` och `b` från avsnitt 3.3.3:

- Antag `a >= b`,
- låt `r` vara resten av heltalsdivision `a // b`,
- om `r == 0`, då är `GCD(a, b) = b`,
- annars kan man använda att är `GCD(a,b) = GCD(b, r)`.

En enkel iterativ implementation:

```
def GCD(a,b):
    if b > a:
        print("a must be bigger than b in GCD(a,b)")
        return None

    r = a % b
    while r != 0:
        a = b
        b = r
        r = a % b
    return b

print("GCD: " + str(GCD(15,12)) + "\n")
print("GCD: " + str(GCD(12,15)))
```

```
GCD: 3

a must be bigger than b in GCD(a,b)
GCD: None
```

Det är inte helt lätt att se hur denna implementation följer av algoritmen. En rekursiv implementation är mycket närmare:

```
def GCD(a,b):
    if b > a:
        print("a must be bigger than b in GCD(a,b)")
```

```
        return None

    r = a % b
    if r == 0:
        return b
    else:
        return GCD(b,r)

print("GCD: " + str(GCD(15,12)) + "\n")
print("GCD: " + str(GCD(12,15)))
```

```
GCD: 3

a must be bigger than b in GCD(a,b)
GCD: None
```

8.1.4 Loopar genom rekursion

Ovan såg vi ett exempel på ett program vi redan skrivit med loopar som kan ersättas med rekursion. Detta kan alltid göras, både för **for** och **while** loopar.

I GCD exemplet var det en **while** loop som byttes ut mot en rekursiv definition. Nedan är en enkel funktion som använder en **for** loop för att beräkna summan av alla tal upp till och med x:

```
def f(x):
    s = 0

    for i in range(x+1):
        s += i;

    return s
```

Detta kan skrivas rekursivt på följande sätt:

```
def f_rec(x):
    if x == 0:
        return 0
    else:
        return x + f_rec(x-1)
```

Tanken är här att vi anropar `f_rec` rekursivt med mindre och mindre tal tills vi kommer till basfallet när `x` är `0`. Detta representerar loopen i `f`. Vi kan testa så att funktionerna returnerar samma resultat genom:

```
for i in range(5):
    print("f(" + str(i) + ") = " + str(f(i)))
    print("f_rec(" + str(i) + ") = " + str(f_rec(i)))
```

```
f(0) = 0
f_rec(0) = 0
f(1) = 1
f_rec(1) = 1
```

```
f(2) = 3
f_rec(2) = 3
f(3) = 6
f_rec(3) = 6
f(4) = 10
f_rec(4) = 10
```

8.1.5 Rekursion över listor

Vi har sett hur man kan skriva rekursiva funktioner över tal. Detta går även att göra över andra datatyper. Det generella tankesättet är att man måste ha ett eller fler basfall och att de rekursiva fallen reducerar indata på något sätt så att man till slut kommer till ett basfall. För tal använde vi ofta `0` som basfall och minskade indata när vi gjorde rekursiva anrop. Vill man skriva rekursiva funktioner över listor använder man istället ofta `[]` som basfall och tar bort ett element från början eller slutet i det rekursiva anropet.

Vi kan på detta sätt skriva vår egen `len` funktion genom:

```
def length(xs):
    # obs: lists behave as boolean values ([] is False)
    if xs:
        xs.pop()
        return (1 + length(xs))
    else:
        return 0

print(length([1,341,23,1,0,2]))
```

```
6
```

Varning: den här funktionen ändrar `xs`! Testa följande:

```
mylist = [1,2,3,6,7]

print(mylist)

print(length(mylist))

print(mylist)
```

```
[1, 2, 3, 6, 7]
5
[]
```

Då `length` har anropat `pop` upprepade gånger har `mylist` blivit tom! För att det inte ska bli så här måste vi anropa `length` med en *kopia* av `mylist`:

```
mylist = [1,2,3,6,7]

print(mylist)

print(length(mylist.copy()))
```

```
print(mylist)
```

```
[1, 2, 3, 6, 7]
5
[1, 2, 3, 6, 7]
```

Summan av alla element i en lista kan beräknas på följande sätt

```
def sum(xs):
    if xs:
        x = xs.pop()  # pop returns the element
        return (x + sum(xs))
    else:
        return 0

print(sum([1,3,5,-1,0,2]))
```

```
10
```

och produkten på detta sätt:

```
def prod(xs):
    if xs:
        x = xs.pop()  # pop returns the element
        return (x * prod(xs))
    else:
        return 1

print(prod([1,3,5,-1,2]))
```

```
-30
```

8.1.5.1 Sortering av listor

En snabb och bra sorteringsalgoritm är *quicksort*. Den fungerar på följande sätt:

1. Välj ett element i listan ("pivotelementet").
2. Dela upp listan i två dellistor: en med alla element som är mindre än pivotelementet och en med alla element som är större än pivotelementet.
3. Sortera dellistorna genom rekursion. Basfallet är tomma listan vilken redan är sorterad.
4. Sätt ihop listorna med pivotelementet i mitten.

För att implementera detta i Python skriver vi först två hjälpfunktioner för att hämta ut alla element som är mindre och större än ett givet element:

```
# All elements less than or equal to v in xs
def smallereq(xs,v):
    res = []
    while xs:
        x = xs.pop()
```

```
        if x <= v:
            res.append(x)
    return res

# All element great than v in xs
def greater(xs,v):
    res = []
    while xs:
        x = xs.pop()
        if x > v:
            res.append(x)
    return res

print(smallereq([2,4,5,6,7,8],5))
print(greater([2,4,5,6,7,8],5))
```

```
[5, 4, 2]
[8, 7, 6]
```

Vi kan sedan implementera `quicksort` på följande sätt:

```
def quicksort(xs):
    if xs == []:
        return xs

    p = xs.pop()
    s = quicksort(smallereq(xs.copy(),p))
    g = quicksort(greater(xs.copy(),p))
    return (s + [p] + g)

print(quicksort([8,3,3,2,4,5,8,5,6,7,8]))
```

```
[2, 3, 3, 4, 5, 5, 6, 7, 8, 8, 8]
```

Vi kan även skriva `smallereq` och `greater` direkt med listomfattningar och få en ännu kortare implementation:

```
def quicksort(xs):
    if xs == []:
        return xs
    else:
        p = xs.pop()
        s = quicksort([ x for x in xs if x <= p ])
        g = quicksort([ x for x in xs if x > p ])
        return (s + [p] + g)

print(quicksort([8,3,3,2,4,5,8,5,6,7,8]))
```

```
[2, 3, 3, 4, 5, 5, 6, 7, 8, 8, 8]
```

8.1.6 Svansrekursion

En fara med att skriva rekursiva funktioner är att de kan bli väldigt långsamma jämfört med iterativ kod skriven med loopar. Exempelvis om vi utvidgar definitionen av `fib(5)` för hand så ser vi följande:

```
fib(5) = fib(4) + fib(3)
       = (fib(3) + fib(2)) + (fib(2) + fib(1))
       = ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
       = (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0)))
         + ((fib(1) + fib(0)) + fib(1))
       = (((1 + 0) + 1) + (1 + 0)) + ((1 + 0) + 1)
       = 5
```

Problemet är att vi beräknar samma tal många gånger, så vi får väldigt långsam kod. Testar man följande så ser man att det tar längre och längre tid att skriva ut utdata:

```
for x in range(35):
    print(fib(x))
```

```
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657
46368
75025
121393
196418
```

```
317811
514229
832040
1346269
2178309
3524578
5702887
```

Det sista talet här tar flera sekunder att skriva ut.

Detta kan lösas genom att skriva en smartare version av `fib` genom *svansrekursion* (eng. *tail recursion*):

```
def fib_rec(n,a=0,b=1):
    if n == 0:
        return a
    elif n == 1:
        return b
    else:
        return fib_rec(n-1,b,a+b)
```

Tanken här är att `n` är en räknare för indexet på det Fibonacci tal vi håller på att beräkna, `a` är värdet 2 steg tidigare i sekvensen och `b` är värdet 1 steg tidigare. På detta sätt behöver vi bara ett rekursivt anrop och koden blir mycket snabbare. Exempelvis skrivs följande ut på nolltid:

```
print(fib_rec(100))
```

```
354224848179261915075
```

Denna funktion kommer fungera på följande sätt för att räkna ut `fib_rec(5)` (dvs. `fib_rec(5,0,1)`):

```
fib_rec(5) = fib_rec(5,0,1)
           = fib_rec(4,1,1)
           = fib_rec(3,1,2)
           = fib_rec(2,2,3)
           = fib_rec(1,3,5)
           = 5
```

På detta sätt har vi alltså undvikit den kombinatoriska explosionen och `fib_rec` är väldigt mycket snabbare än `fib`.

Vill man prova med större tal går det också:

```
print(fib_rec(1000))
```

```
43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875
```

Vill man prova med väldigt stora tal måste man göra följande då Python har en inbyggd gräns för hur många rekursiva anrop som tillåts:

```
import sys

sys.setrecursionlimit(100000)
```

```
print(fib_rec(10000))
```

Beräkningen av detta väldigt stora tal går på nolltid, även fast det består av över `2000` siffror. Så rekursiva funktioner kan vara väldigt snabba om man skriver dem på rätt sätt.

Python är dock inte alltid jättesnabbt när det kommer till att exekvera funktioner skrivna med rekursion. Så det är oftast bättre att skriva en funktion med hjälp av `for` eller `while` om man behöver snabb kod. Andra språk optimerar även rekursiva funktioner så att de blir svansrekursiva, men i Python måste man göra detta för hand.

8.1.7 Diskussion om rekursion

- Princip: bryt upp i mindre instanser, lös dessa.
- Rekursion är att "ta hjälp av en (osynlig) kompis"
- Rekursion är en effektiv algoritmisk teknik
  - Rekursion är dock inte Pythons starkaste sida. Många programmeringsspråk konverterar rekursiv kod till while loopar (möjligt när rekursiva anropet är sist i funktionen, "svansrekursion"), vilket ger snabbare och mer minnessnål exekvering.
- Ofta användbar teknik i datastrukturer (t.ex. sökträd).

Principen för ett funktionsanrop är som följer.

- När ett funktionsanrop görs läggs anropsparametrarna på den minnesarea som kallas *stacken*.
- En funktion börjar exekvera genom att plocka upp anropsparametrarna från stacken till lokala variabler.
- När `return` exekveras så läggs returvärdet på stacken.
- Den kod som gjort ett funktionsanrop kan alltså hämta resultatet på stacken när funktionen är klar.

Det är denna princip som gör att rekursion i praktiken är enkel i datorer.

8.2 Anonyma och högre ordningens funktioner

När vi har pratat variabler och värden har vi hittills framförallt pratat om heltal, flyttal, strängar, listor, med mera. Det finns goda skäl att lägga *funktioner* till uppräkningen av värden att arbeta med, vilket handlar om möjligheten att skapa nya funktioner när man behöver dem, skicka funktioner som argument, returnera funktioner, och kunna tillämpa operatorer på funktioner. När ett programmeringsspråk har de möjligheterna säger man att funktioner behandlas som *first class citizens* (eller *first class objects* eller *first class functions*).

Gamla språk i samma tradition som Python, så kallade *imperativa språk*, var dåliga på att hantera funktioner, men i Lisp och andra språk för *funktionell programmering* så har funktioner varit *first class citizens* sedan 50-talet. Idag kan man säga att det är en vanlig finess som många språk stödjer, däribland Python.

8.2.1 Anonyma funktioner

Begreppet *anonyma funktioner* med hjälp av *lambda-uttryck* finns i många språk idag. Man säger att funktionerna är *anonyma* eftersom man inte ger dem ett namn, utan bara skapar dem. Tag kodsnutten

```
double = lambda x: 2 * x

print(double(3))
```

```
6
```

Definitionen `double` är en funktion som tar ett argument, `x`, och beräknar `2 * x`. Vi kan även få en funktion av två argument genom:

```
f = lambda x, y: 2 * x + y

print(f(3,4))
```

```
10
```

Detta är ekvivalent med följande definition:

```
def f(x,y):
   return 2 * x + y
```

Man ska dock inte se lambda-uttryck som bara ett alternativ till vanliga funktionsdefinitioner; de är snarare komplement till `def` och dess främsta användningsområde är för att skapa tillfälliga små funktioner att använda tillsammans med *högre ordningens funktioner*.

Anledningen att dessa introduceras med `lambda` är från den så kallade λ-kalkylen. Detta är en beräkningsmodell uppfunnen av logikern Alonzo Church på 1930 talet (https://en.wikipedia.org/wiki/Lambda_calculus). I denna modell är allting uppbyggt från funktioner och variabler (så t.ex. heltal representeras även de som funktioner), och man kan på detta sätt representera alla beräkningar som kan göras med en Turingmaskin genom att bara använda lambda-uttryck.

Anmärkning: En Turingmaskin är en matematisk beräkningsmodell/koncept, som avänds för att förstå algoritmer och deras begränsningar. Datorer som används idag är inte uppbyggda enligt Turingmaskinkonceptet, men termen __Turingkomplett_ om en formell definition av regler, som t.ex. utgör ett programmeringsspråk, är sådan att vilken Turingmaskin som helst kan representeras med instruktioner enligt dessa regler.

8.2.2 Högre ordningens funktioner

En *högre ordningens funktion* tar en funktion som argument och är ett djupare begrepp än det kanske först låter som. Poängen med högre ordningens funktioner är att de kan vara ett sätt att generalisera funktionalitet. Genom att identifiera vanliga mönster för beräkningar och låta dessa implementeras med högre ordningens funktioner kan man förenkla, korta ner, och kanske även snabba upp sin kod. Några exempel på beräkningar man kan vilja göra är:

1. Ta en lista med tal och returnera en lista med dess kvadrater.
2. Ta en lista med icketomma listor och returnera en lista med det första elementet från varje lista.
3. Ta en lista med strängar och räkna ut deras längder.

Dessa tre beräkningar har gemensamt att man ska utföra något på varje element i en lista. Det är inte svårt att skriva en `for`-loop för detta, men funktionen `map` är mer kompakt, känns igen av andra programmerare, och är redan implementerad (se också uppgift 7 i kapitel 6). Det första argumentet till `map` är en funktion som sedan appliceras på alla element i listan. Denna funktion kan antingen ges genom `lambda` eller genom namn på vilken Pythonfunktion som helst.

Ovan uppgifter kan därför lätt lösas med några små enkla uttryck:

```
# 1
print(list(map(lambda x: x**2, [1,2,3,4])))

# 2
print(list(map(lambda t: t[0], [[1,1], [0,2,1], [3,1,53,2]])))

# 3
print(list(map(len, ["hej","hopp"])))
```

```
[1, 4, 9, 16]
[1, 0, 3]
[3, 4]
```

Obs: `map` returnerar en generator, så vi måste lägga in `list` för att få ut en lista.

8.2.2.1 Inte bara för listor

Observera också att `map` och liknande funktioner inte är låsta till listor, utan kan tillämpas på allt som i Python är definierat som *itererbart*. Det inkluderar till exempel strängar, tupler och uppslagstabeller, men kan också definieras av programmeraren själv.

För att beräkna längden på varje rad i filen `people.txt` med innehåll:

```
Anders
Christian
Kristoffer
Lars
```

kan vi köra:

```
print(list(map(len, open('people.txt'))))
```

```
[7, 10, 11, 5]
```

Notera att newline tecknet `\n` räknas med och är av längd 1.

8.2.2.2 Högre ordningens funktion: `filter`

En annan typ av funktionalitet som vi ofta vill använda är att välja ut element som uppfyller något kriterie. Till det finns funktionen `filter`. Den tar en funktion `pred` och en itererbar datastruktur och returnerar en generator (som du kan iterera över eller plocka fram som en lista med `list`) med de element som `pred` returnerar `True` eller motsvarande (t.ex. icke-noll) för. Exempel:

```
print(list(filter(lambda x: x % 2, range(10))))

print(list(filter(lambda x: x, [[1], [2,3], [], [], [4,5,6]])))
```

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

8.2.3 Egna högre ordningens funktioner

Funktionerna `map` och `filter` har ingen särställning i Python, de är helt enkelt bra verktyg som många finner praktiska att använda. Det är inte så svårt att skriva dessa funktioner själv. Däremot kan det ju vara så att du hittar egna beräkningsbehov som kan generaliseras.

Till exempel kan man beräkna summan och produkten av elementen i två listor med hjälp av slicing på följande sätt:

```
def sum_rec(xs):
    if xs == []:
        return 0
    else:
        return (xs[0] + sum_rec(xs[1:]))

def prod_rec(xs):
    if xs == []:
        return 1
    else:
        return (xs[0] * prod_rec(xs[1:]))

print(sum_rec([1,2,3,4]))
print(prod_rec([1,2,3,4]))
```

```
10
24
```

Den enda skillnaden mellan dessa är att en använder `+` och `0`, och den andra `*` och `1`. Vi kan alltså generalisera dessa till en funktion som tar in en funktion `f` och ett element `x` :

```
def foldr(xs,f,x):
    if xs == []:
        return x
    else:
        return f(xs[0],foldr(xs[1:],f,x))

sum = lambda xs: foldr(xs,lambda x,y: x + y,0)
prod = lambda xs: foldr(xs,lambda x,y: x * y,1)

print(sum([1,2,3,4]))
print(prod([1,2,3,4]))
```

```
10
24
```

Vi kan nu även lätt skriva en funktion som använder exponentiering som operation:

```
exps = lambda xs: foldr(xs,lambda x,y: x ** y,1)

print(exps([2,3,4]))
```

```
241785163922925834941235 2
```

Vi kan även skriva en funktion för att slå ihop listor av listor:

```
join = lambda xs: foldr(xs,lambda x,y: x + y,[])

print(join([[1], [2,3], [], [], [4,5,6]]))
```

```
[1, 2, 3, 4, 5, 6]
```

Högre ordningens funktioner är alltså mycket smidiga för att skriva väldigt generella funktioner och minimera duplikation av kod!

Obs: det finns en funktion som heter `reduce` som fungerar exakt som `foldr`, men med argumenten i en annan ordning. Den ligger i biblioteket `functools` och importeras på följande sätt:

```
import functools

print(functools.reduce(lambda s, x: s+x, [1,2,3], 0))
```

```
6
```

8.3 Uppgifter

1. Skriv en rekursiv funktion `rec_prod_m(n,m)` som tar två tal `n` och `m` och beräknar `n * (n-m) * (n-2m) * ...` tills `n-km` har blivit negativt.

   Tips: Funktionen kan skrivas genom att modifiera funktionen `fac` något.

2. Vad händer om vi ändrar till `return 0` i basfallet för `prod`? Varför?

3. Deklarera fem funktioner `f1,...,f5` med hjälp av `lambda` som utför addition, subtraktion, multiplikation, division, och exponentiering. Till exempel ska `f1(4,7)` returnera 11. Använd dessa funktioner för att skriva uttrycken

   ```
   (6*(2+3)/5)**2
   (10 - 2**3)*5
   ```

4. Använd `map` och `lambda` för att skriva funktion `get_evens(xs)` som tar en lista med tal `xs` och konverterar jämna tal till `True` och udda tal till `False`.

5. Studera koden nedan. Hur många anrop till `f` görs totalt om vi anropar `f` med 3? Hur många blir det för 4? Varför?

```
def f(n):
    if n <= 1:
        return 1
    else:
        return f(n-1) + f(n-2) + f(n-3)
```

6. Skriv om funktionen `collatz(n)` (se kapitel 7) som en rekursiv funktion.
7. Använd funktionen `foldr` för att skriva en funktion `joinstrings` som konkatenerar en lista med strängar.
8. Skriv `fac` och `fib` med hjälp av loopar istället.
9. Vad blir fel om vi inte kopierar `xs` i `quicksort`?
10. † Pascals triangel ser ut så här:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

osv… Skriv en funktion `pascal(n)` som returnerar den `n`:e raden. Den ska fungera på följande sätt:

```
for i in range(1,8):
   print(pascal(i))
```

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
```

Kapitel 9

Objektorientering 1: Klasser

9.1 Objektorientering

Objektorientering (förkortas OO) är ett programmeringsparadigm eller programmeringsstil där man samlar data och datastrukturer med de funktioner som kan tillämpas på dem. I Python är i princip allt implementerat som objekt. Detta gäller såväl heltal, flyttal, strängar, listor, uppslagstabeller som funktioner. Vad menas då med att knyta ihop datastrukturer med "funktioner som kan tillämpas på dem"? Ta till exempel listor:

```
l = [2,7,2]
print(l.append(1))
```

```
[2, 7, 2, 1]
```

I objektorienterade termer har vi på första raden skapat ett objekt `l` av typ `list`. Vi har på andra raden använt funktionen `append` som *är specifik* för datatypen `list`. I objektorientering benämner man funktioner som är knutna till objekt för *metoder*. De anropas med hjälp av punktnotationen `object.method(<parameters>)`. Vi ska snart se på anatomin för en metod. Ett annat exempel är strängar:

```
s = "Hej"
print(s.lower())
```

```
hej
```

Här har vi skapat ett objekt `s` av typ `str` och använt metoden `lower` (specifik för datatypen `str`).

9.1.1 Varför objektorientering?

Objektorientering lämpar sig bra att använda när man vill:

- Knyta samman mycket data som relaterar till varandra (hör till samma objekt).
- Kunna definiera egna datatyper.

Säg till exempel att vi vill hålla data associerat till en bil (kanske för ett datorspel) i minnet i ett program. Det kan vara däckbredd, färg, antal dörrar, nuvarande bensinnivå, hastighet, vinkel på framhjulen och en mängd annan data. Det skulle då vara smidigt att ha all data för denna bil "på ett ställe", om vi hade ett bil-objekt `b` skulle vi komma åt och kunna ändra alla *attribut* i detta objekt med

```
b.tire_width()
b.color()
b.increase_speed(2)
```

Det är naturligt att tänka på objekt som "fysiska ting" (t.ex. en bil), men det finns även naturliga objekt som vi inte kan ta på. Vi kommer strax visa hur vi implementerar en kurs (t.ex. DA2004) där vi kan samla användarinformation, resultat, med mera. Slutligen ska vi även titta på hur vi kan implementera vår egna datastruktur för att representera heltalsmängder.

Två fördelar med att skriva objektorienterad kod är att det:

1. Uppmuntrar god struktur genom att samla data och relaterad funktionalitet på samma ställe.
2. Är bra för abstraktion, eftersom det blir lättare att gömma detaljer.

9.1.2 Hur skapas objekt?

Hur skriver vi då objektorienterad kod? För att skapa egna objekt (tänk till exempel på en bil eller en kurs som ett objekt) så måste vi specificera en beskrivning för objektets data, metoder och allmänna struktur. En sådan mall heter *klass* i objektorienteringens terminologi. Vi kan skapa en klass som paketerar data och funktionalitet (du kan tolka "klass" som "klassificering", som minnestrick). En klass är att jämföra med en typ och man kan skapa nya *instanser* av klasser. Till exempel är `[1,2,3]` en instans av klassen `list`. Varje instans kan ha olika *attribut* (variabler) kopplade till den för att representera dess nuvarande tillstånd. Instanser kan även ha *metoder* (funktioner definierade i klassen) för att modifiera dess tillstånd (ex. `xs.pop()`).

Vi har i kursen använt många exempel på klasser: heltal, listor, uppslagstabeller, m.m. Vi ska nu lära oss att skriva våra egna.

Ett minimalt exempel, en tom klass:

```
class Course:
    pass
```

Konventionen är att klassnamnet börjar med stor bokstav. Detta tillåter *instansiering* av klassen vilket skapar ett *objekt*. Man kan sedan knyta olika *attribut* till objektet.

```
k = Course()       # An object is created, an instance of the class Course
k.code = "DA2004"  # An attribute is assigned to the object
k2 = Course()      # A new object is created
print(k.code)
```

```
DA2004
```

Om vi istället försöker nå `code` attributen för objektet `k2` så får vi ett felmeddelande då vi inte har tillskrivit `k2` något sådant attribut:

```
print(k2.code) # This object does not have an attribute "code"
```

```
AttributeError: 'Course' object has no attribute 'code'
```

Här är code alltså en variabel som tillhör objektet/instansen k. Vi kommer ofta säga "instansattribut" för denna typ av variabler.

9.1.3 Metoder

En *metod* är en funktion definierad i en klass. I `Course` (nedan) är alla **`def`**-initiationer metoder. Metoder följer samma konventioner som funktioner, det vill säga små bokstäver och, om flera ord, separerade med _.

```
class Course:

    def __init__(self, code, name, year=2020):       # Constructor
        self.participants = []
        self.code = code
        self.name = name
        self.year = year

    def number_participants(self):
        return len(self.participants)

    def new_participant(self, name, email):
        self.participants.append((name, email))
```

Den första metoden, `__init__`, har en särställning och kallas *konstruktor*. En konstruktor initierar ett nytt objekt. Den anropas bara vid instansiering (d.v.s. då vi skapar objektet). Metoder på formen `__fkn__` används av Python-systemet, så man bör vara försiktig med hur dessa används. Konstruktorn heter alltid `__init__`, så när vi kör

```
da2004 = Course("DA2004", "Programmeringsteknik")
```

anropas konstruktormetoden `Course.__init__` och resultatet sparas i variabeln `da2004`. Objektet i `da2004` är då en *instans* av `Course`-klassen. Det är vidare endast `__init__` som anropas vid instansiering. De övriga metoderna finns tillgängliga att använda när vi anropar dem.

Vi kan modifiera ett objekt genom att anropa dess metoder. Detta gör man genom punkt-notation (precis som vi använt `xs.pop()` och andra listmetoder för att modifiera listor).

```
da2004.new_participant('Johan Jansson', 'j.jansson@exempel.se')
print(da2004.number_participants())
```

```
1
```

9.1.4 Self

Vad är då uttrycket `self` som finns med framför alla attribut och som parametrar till metoder? Variabeln `self` i `Course`-klassen representerar objektet som metoden anropas från (man måste inte kalla det argumentet för `self`, men det är en konvention bland Python programmerare att göra så). I exemplet

ovan är `self` i `new_participant`, själva objektet `da2004`. Detta kan vara svårt att greppa när man först lär sig om klasser och objektorientering. För illustration kan du tänka dig att raden

```
da2004.new_participant('Johan Jansson', j.jansson@exempel.se')
```

egentligen motsvarar:

```
__course_object__.new_participant(da2004, 'Johan Jansson', 'j.jansson@exempel.se')
```

Obs: Detta är inte korrekt syntax i Python, utan endast en illustration av konceptet att metoden `new_participant` tar objektet `da2004` som parameter för att kunna modifiera det.

9.1.5 Klass- och instansattribut

Attribut är ett annat sätt att benämna variabler som är specifika för objekt eller klasser. Det finns två olika typer av attribut; *instansattribut* och *klassattribut*. Generellt sett används *instansattribut* för att hålla data som är unik för varje instans (dvs, specifika för objektet), medans *klassattribut* används för attribut och metoder som delas av alla instanser av klassen:

```
class Course:

    university = 'Stockholm University'  # class attribute shared by all instances
                                         # (1) Occurs outside of constructor
                                         # (2) Has no 'self', i.e., specific object,
                                         # associated to it

    def __init__(self, code, name, year=2020):
        # instance attributes below, unique to each instance
        self.participants = []
        self.code = code
        self.name = name
        self.year = year
```

Exempelanvändning:

```
da2004 = Course('DA2004', 'Programmeringsteknik')
da2005 = Course('DA2005', 'Programmeringsteknik (online version)', year = 2021)

print(da2004.university)                          # shared by all courses
print(da2005.university)                          # shared by all courses
print(da2004.code, da2004.year, da2004.name)      # unique to da2004
print(da2005.code, da2005.year, da2005.name)      # unique to da2005
print(Course.university)                          # you can also access a class
                                                  # attribute from the class itself
```

```
Stockholm University
Stockholm University
DA2004 2020 Programmeringsteknik
DA2005 2021 Programmeringsteknik (online version)
Stockholm University
```

Vi kan även skriva över nuvarande värden på attribut genom

```
da2004.year = 2021
print(da2004.year)
```

```
2021
```

På samma sätt som vi ovan skrivit över instansattributen `year` i `da2004`, men inte i `da2005`, kan vi även skriva över metoder i ett objekt (kom ihåg att funktioner ej har någon särställning utan är som andra objekt i Python).

Varning: Muterbara data kan ha överraskande effekter om de används som klassattribut!

Om vi till exempel hade lagt listan `participants` som en klassvariabel kommer den delas av *alla* kurser:

```
class Course:

    university = 'Stockholm University'   # class attribute shared by all instances
    participants = []                     # class attribute shared by all instances

    def __init__(self, code, name, year=2020):
        # instance attributes below, unique to each instance
        self.code = code
        self.name = name
        self.year = year

    def new_participant(self, name, email):
        self.participants.append((name, email))

d = Course('da2004', 'Programmeringsteknik')
e = Course('da2005', 'Programmeringsteknik (online version)')

d.new_participant('Johan Jansson', 'j.jansson@exempel.se')
print(d.participants)                  # shared by all courses
print(e.participants)                  # shared by all courses
```

```
[('Johan Jansson', 'j.jansson@exempel.se')]
[('Johan Jansson', 'j.jansson@exempel.se')]
```

9.1.6 Privata och publika metoder och attribut

I objektorienterad programmering skiljer man ofta på *privata* och *publika* metoder och attribut. De publika metoderna utgör *gränssnittet* (eng. *interface*) för en klass och är de metoder som användare av klassen ska använda. De privata metoderna och attributen är för internt bruk. I många programmeringsspråk finns det stöd för att kontrollera åtkomst av metoder och attribut så att man inte av misstag börjar förlita sig på annat än klassens gränssnitt. På detta sätt uppnår man *abstraktion* med hjälp av objektorientering: https://en.wikipedia.org/wiki/Abstraction_(computer_science)#Abstraction_in_object_oriented_programming.

Python fokuserar på enkelhet och har istället en konvention: identifierare för attribut/metoder som börjar med understreck `_` *ses som privata*. De bör alltså bara användas internt.

Vi har nedan lagt till en privat metod `_check_duplicate` i vårt kursexempel. Det är vanligt att studenter

registrerar sig med olika email-adresser. Metoden kontrollerar om identiska namn förekommer i deltagarlistan när vi lägger till en deltagare och skriver i så fall ut en varning. Vi kan då vidta lämpliga åtgärder (t.ex. implementera en metod `remove_participant` för att ta bort dubbla deltagare).

```
import warnings  # included in Pythons standard library

class Course:

    def __init__(self, code, name, year=2020):       # Constructor
        self.participants = []
        self.code = code
        self.name = name
        self.year = year

    def _check_duplicate(self, name, email):
        for p in self.participants:
            if name == p[0]: # check if identical name
                # raise warning and print message if identical name:
                # uses string formatting, see e.g.:
                # https://www.w3schools.com/python/ref_string_format.asp
                warnings.warn(
                    "Warning: Name already exists under entry: {0}".format(p))

    def number_participants(self):
        return len(self.participants)

    def new_participant(self, name, email):
        self._check_duplicate(self, name, email)
        self.participants.append((name, email))
```

Vi kan nu se vad som händer om vi registrerar studenter med samma namn:

```
da2004 = Course('DA2004', 'Programmeringsteknik')
da2004.new_participant('Johan Jansson', 'j.jansson@exempel.se')
da2004.new_participant('Johan Jansson', 'j.jansson@gmail.com')
```

Den tredje raden kommer att generera en "varning" som skrivs ut

```
UserWarning: Warning: Name already exists under entry:
    ('Johan Jansson', 'j.jansson@exempel.se')
```

I detta exempel används metoden `_check_duplicate` för internt bruk i klassen `Course`. Det är alltså inte tänkt att användaren av klassen ska anropa denna metod. Metoden är inte med i *gränssnittet* och ökar således inte storleken (komplexiteten i användning) av vår klass. Samtidigt utför den gömt en användbar funktionalitet när vi lägger till deltagare. Notera att även om det är markerat genom _ att metoden är privat, så kan vi fortfarande anropa den med följande explicita metodanrop:

```
da2004._check_duplicate('Johan Jansson', 'j.jansson@exempel.se')
```

9.1.7 Ordlista

- *Objekt*: ett element av någon typ med olika attribut och metoder.
- *Klass*: en typ innehållande attribut och metoder (t.ex. `list`, `dict`, `Course` från exemplet ovan, etc.).
- *Instansiering*: då ett objekt skapas från en viss klass.
- *Metod*: en funktion **`def`**inierad i en klass (ex. `pop`, `insert`).
- *Konstruktor*: metod för att instansiera ett objekt av klassen (ex. `dict()`). Varje klass har en konstruktor `__init__` som antingen är implicit eller explicit definierad.
- *Klassattribut*: variabel definierad utanför konstruktorn i en klass (delas av alla instanser av klassen).
- *Instansattribut*: variabel definierad i konstruktorn för klassen (eller satt med `self` i klassen). Tillhör en specifik instans av klassen.

9.2 Modularitet genom objektorientering

I `Course` exemplet ovan har vi använt en lista med tupler för att representera deltagarlistan `participants`. En mer objektorienterad lösning hade varit att ha en separat klass för deltagare:

```
class Participant:

    def __init__(self, name, email):
        self._name = name
        self._email = email

    def name(self):
        return self._name

    def email(self):
        return self._email
```

Vi måste nu modifiera metoderna i `Course` så att de hanterar deltagare representerade som `Participant` istället för tupler av strängar. Exempelvis:

```
class Course:

    # old code unchanged, except for _check_duplicate which
    # has to be modified to take a Participant

    def new_participant(self, p):
        self._check_duplicate(self, p)
        self.participants.append(p)
```

Vi måste nu skapa deltagare genom:

```
d = Participant("Johan Jansson", 'j.jansson@exempel.se')
da2004.new_participant(d)
da2004.new_participant(Participant("Foo Bar", "foo@bar.com"))
```

Det kan tyckas konstigt att skriva om sin kod så här, men fördelen är att eventuella framtida förändringar kommer bli mindre komplexa. Exempelvis, låt oss säg att vi vill separera förnman och efternamn för deltagare. Då räcker det att skriva:

```
class Participant:

    def __init__(self, fname, lname, email):       # Changed!
        self._fname = fname                        # Changed!
        self._lname = lname                        # Changed!
        self._email = email

    def name(self):
        return self._fname + " " + self._lname     # Changed!

    def email(self):
        return self._email
```

Koden för `Course` behöver nu inte alls ändras något mer då den bara använder `Participant` gränssnittet. På detta sätt leder objektorientering till mer modulär kod, dvs varje klass kan ses som en liten kodmodul i sig och interna modifikationer påverkar ingen annan kod i programmet så länge gränssnittet är oförändrat.

9.3 Ett till exempel på OO: talmängder

Python har en typ för mängder, dessa är som uppslagstabeller, fast nycklarna inte är associerade med några värden. För dokumentation se:

https://docs.python.org/3/tutorial/datastructures.html#sets

https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset

Syntaxen för att skapa en mängd (`set`) är liknande t.ex. listor, men med `{}`. Några exempel:

```
basket = set(['apple', 'banana', 'orange', 'pear'])
# the above is equivalent to
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)
print('orange' in basket)
print('crabgrass' in basket)

a = set([1,2,5])
b = set([3,5])
print(a)
print(a - b)  # difference
print(a | b)  # union
print(a & b)  # intersection
print(a ^ b)  # numbers in a or b but not both
```

```
{'pear', 'apple', 'orange', 'banana'}
True
False
{1, 2, 5}
{1, 2}
{1, 2, 3, 5}
```

```
{5}
{1, 2, 3}
```

9.3.1 Egen implementation

Här kommer ett till större exempel taget från kursboken (Introduction to Computation and Programming using Python av John V Guttag, sida 111), där vi implementerar våran egen mängd-datatyp `IntSet` med hjälp utav listor. Vi kan representera mängder av heltal som listor med invarianten att det inte finns några dubletter.

```
class IntSet:
    """An IntSet is a set of integers"""

    def __init__(self):
        """Create an empty set of integers"""
        self.vals = []

    def insert(self,e):
        """Assumes e is an integer and inserts e into self"""
        if e not in self.vals:
            self.vals.append(e)

    def member(self,e):
        """Assumes e is an integer.
           Returns True if e is in self, and False otherwise"""
        return e in self.vals

    def delete(self,e):
        """Assumes e is an integer and removes e from self
           Raises ValueError if e is not in self"""
        try:
            self.vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def get_members(self):
        """Returns a list containing the elements of self.
           Nothing can be assumed about the order of the elements"""
        return self.vals
```

Vi bibehåller *invarianten* att ett `IntSet` ej innehåller några dubletter i alla funktioner.

Lite tester:

```
x = IntSet()
print(x)
x.insert(2)
x.insert(3)
x.insert(2)
```

```
print(x)
print(x.member(5))
print(x.member(3))
x.delete(2)
print(x.get_members())
```

```
<__main__.IntSet object at 0x7febb7002040>
<__main__.IntSet object at 0x7febb7002040>
False
True
[3]
```

De två första `print` anropen skriver ut information om objektet; var det är definierat och på vilken minnesadress det sparats på. Om man kör denna kodsnutt igen kommer troligtvis objektet ligga på ett annat ställe i minnet.

Vi kan även lägga till metoder `__str__(self)` och `__len__(self)` som gör att vi kan använda oss utav Python-systemets syntax `str()`, och `len()`. Se https://docs.python.org/3/reference/datamodel.html#object.__str__ och https://docs.python.org/3/reference/datamodel.html#object.__len__. På detta sätt kan vi få mängder att skrivas ut snyggare än ovan och ett exempel på hur `__str__` metoden kan se ut är

```
    def __str__(self):
        result = ''
        for e in self.vals:
            result += str(e) + ','
        return '{' + result[:-1] + '}'
```

Detta gör att följande skrivs ut istället:

```
x = IntSet()
print(x)
x.insert(2)
x.insert(3)
x.insert(2)
print(x)
```

```
{}
{2,3}
```

Låt oss säga att vi vill kunna jämföra `IntSet` med <= och att `x <= y` ska tolkas som att "x är en delmängd till y". Då kan vi lägga till följande metod till `IntSet` klassen:

```
    def __le__(self,other):
        for x in self.vals:
            if not other.member(x):
                return False
        return True
```

Med tester:

```
print(x <= x)
empty = IntSet()
print(empty <= x)
```

```
print(x <= empty)
```

```
True
True
False
```

Vi kan på detta sätt definiera de olika standardoperationerna i Python (+, <, =, ...) för våra egna klasser. Vi kan även *operatoröverlagra* andra operatorer som `in`, `and` med flera. För dokumentation kring namn på standardoperationer för metoder se https://docs.python.org/3/library/operator.html. Till exempel kan vi döpa om metoden `member` i vår `IntSet`-klass till `__contains__`. Då kan vi använda oss av operatorn `in` och skriva `5 in x` istället för att skriva `x.member(5)`.

9.4 Uppgifter

1. Implementera metoden `__str__` så att vi kan använda oss av `str()` för att skriva ut lämplig sträng i `Participant` och `Course`. Det är valfritt hur strängen ser ut men namn och email ska förekomma i `Participant` och kurskod och kursnamn i `Course`.
2. Lägg till attributen `teacher` av typ `bool` till klassen `Participant`. Skriv ut lärarens namn på något speciellt sätt så att man från `Participant`-listan ser vem det är.
3. Lägg till metoder så att operatorerna == och != finns för klassen `IntSet`. Operatorn == ska returnera `True` om mängderna innehåller samma element, annars `False`. Operatorn != är negationen (logiska motsatsen) av ==.

   Tips: läs dokumentationen https://docs.python.org/3/library/operator.html.
4. Implementera union, differens och snitt för `IntSet`. Snitt kan du implementera på valfritt sätt, men implementera union genom att operatoröverlagra + och differens genom att operatoröverlagra -. Metoderna skall returnera ett nytt `IntSet` objekt. Till exempel:

   ```
   x = IntSet()
   x.insert(2)
   x.insert(3)
   x.insert(1)

   y = IntSet()
   y.insert(2)
   y.insert(3)
   y.insert(4)
   print(x == y, y != x)
   print(x.intersection(y))
   print(x + y, y - x)
   ```

   ```
   False True
   {2,3}
   {4,2,3,1} {4}
   ```
5. Implementera en klass `Dog` med funktionalitet som möjliggör gränssnittet nedan.

```
d = Dog('Fido')
e = Dog('Buddy')
d.add_trick('roll over')
e.add_trick('play dead')
d.add_trick('shake hands')
print(d.name, d.tricks)
print(e.name, e.tricks)
```

```
Fido ['roll over', 'shake hands']
Buddy ['play dead']
```

6. Implementera metoderna `__str__` och `__gt__` i klassen `Dog` så vi kan skriva `str(d)` och jämföra `d > e` (`>` baseras på hur många trick de kan). Exempel nedan (givet koden vi körde i uppgift 5)

```
print(str(d))
print(d > e)
```

```
Fido knows: roll over, shake hands
True
```

Kapitel 10

Objektorientering 2: Arv

Arv är en central princip i objektorientering (OO) som handlar om att på ett strukturerat sätt kunna återanvända kod och funktionalitet i redan definierade klasser med möjligheten att *lägga till* eller *ändra* på den existerande klassen. Ofta så finns redan en klass som nästan gör det man vill, och det är då en god idé att låta *ärva* från den klassen.

Ett enkelt exempel är vi har en klass `A` som vi vill använda funktionaliteten från, men ändra på vissa delar:

```
class A:
    """This is a base class
    """
    class_integer = 10

    def __init__(self):
        self.vals = [5]

    def power_sum(self):
        return sum([i**2 for i in self.vals]) + A.class_integer

class B(A):
    """This is a sub class
    """
    def __init__(self, x, y):
        self.vals = [x, y]

    def print_result(self):
        print(self.power_sum())
```

Här är `A` en *basklass* (eng. *base class*) för `B` och `B` är *subklass* (eng. *sub class*) till `A`. Man kan också säga att `A` är *superklass* (eng. *super class*) till `B`. I flera programmeringsspråk används ofta de engelska termerna *parent class* och *child class* för bas- och subklass.

Lägg märke till att **`class`**-definitionen av `B` tar `A` som ett argument, vilket indikerar att `B` ärver från `A`. Med ett arv i detta fall så menas att `A`:s funktionalitet *automatiskt* kommer att vara implementerad i `B`,

så länge man inte aktivt definierar om något. I det här fallet har klass-konstruktorn för B, `__init__`, definierats om för att ta *två* istället för *ett* argument (x och y). Utöver detta så används funktionaliteten från A: t.ex. finns i ovan exempel inget skäl att definiera om `power_sum` i B.

Användningen är ganska rättfram:

```
a = A()
print(a.power_sum())   # method defined in A

b = B(10, 20)
print(b.power_sum())   # calling method in B inherited from A
b.print_result()       # uses method specific to B
```

```
35
510
510
```

Obs: i Python ärver man inte instansattribut automatiskt! Det görs i många andra programmeringsspråk.

```
class C(A):
    def __init__(self, y):
        self.vals.append(y)

c = C(10)
```

```
AttributeError: 'C' object has no attribute 'vals'
```

För att se till att C ärver även instansattribut från A, får vi ändra konstruktorn lite genom att anropa `super`, som ser till att metoden som följer anropas på i basklassen:

```
class C(A):
    def __init__(self, y):
        super().__init__()  # call constuctor of super class
        self.vals.append(y)

class D(B):
    def __init__(self, y):
        super().__init__(5, 10) # call constuctor of super class
        self.vals.append(y)

c = C(10)
d = D(10)
print(c.power_sum())
d.print_result()
c.print_result()       # raises an Error
```

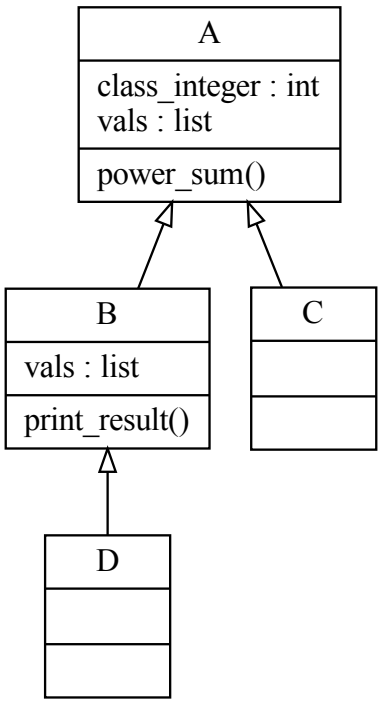
```
135
235
AttributeError: 'C' object has no attribute 'print_result'
```

Felet uppstår på grund av att `print_result()` inte är definerad för C, som ärvt från A. Klass D däremot har ärvt metoden från B.

10.1 Klassdiagram

Ett klassdiagram visualiserar hur klasser hänger ihop. Det innehåller ingen information om objekt. Man kan ange att en klass ärver från en annan (relationen "is-a", dvs "är en") och att en klass innehåller objekt från en annan klass ("aggregation" och "komposition").

Det finns många sätt att rita klassdiagram och informell notation används ofta. Det finns professionella verktyg, som ofta bygger på industristandarden UML, som genererar kod från klassdiagram. T.ex. är fig. 10.1 genererad m.h.a. Python-biblioteket `pyreverse` (som är en del av kodanalysverktyget `pylint`).



Figur 10.1: Klassdiagram för `A`, `B`, `C` och `D`

Ett *objektdiagram* liknar till stor del ett klassdiagram, men förklarar *tillståndet* i ett program vid en *viss* tidpunkt: t.ex. skulle ett objektdiagram av ovan visa att `a.vals` innehåller `[5]`, `b.vals` innehåller `[10, 20]`, `c.vals` innehåller `[5, 10]` och `d.vals` innehåller `[5, 10, 10]`. Klassdiagrammet i fig. 10.1 förklarar relationer mellan klasser *oavsett* tidpunkt.

10.2 Substitutionsprincipen

Det är erkänt svårt bestämma bra klasser. Många böcker har skrivits på ämnet, med många olika tillvägagångssätt för att bestämma vilken klass-struktur man ska ha. Det här kan bli väldigt komplicerat fort, men en väldigt bra tumregel, som är känd som Liskov's substitutionsprincip (efter MIT professorn och Turingprisvinnaren Barbara Liskov), är:

- Allt som gäller för en klass ska gälla för dess subklasser.

Principen ger *god design*. Det kanske låter som en självklar princip, men det är lätt att vilja plocka bort funktionalitet i subklasser och det finns många exempel som bryter mot principen.

10.3 Funktioner som ger information om klasser

Två användbara inbyggda funktioner när det gäller klasser är `isinstance` och `issubclass`. Den första funktionen om ett givet *objekt* är en *instans* av en klass, och den andra om en *klass* är en subklass av en annan *klass*, dvs om en klass ärver från en annan.

Om vi använder oss av klassdefinitionera (`A, B, C, D`) och instanserna (`a, b, c, d`) ovan, så kan vi t.ex. göra:

```
isinstance(a, A) # a is an instance of the class A
isinstance(d, A) # d is an instance a class that inherits from A
isinstance(d, C) # d is not an instance of a class that inherits from C

issubclass(A, A)
issubclass(D, A)
issubclass(D, C)
```

```
True
True
False

True
True
False
```

Lägg märke till att det inte bara är den *direkta* basklassen som kollas med funktionerna, utan även vidare tillbaka i "arvsledet".

I programmeringsspråk som stödjer multipelt arv (som Python, mer om det i kapitel 11) så kallas ett sådant "arvsled" *method resolution order* (förkortas vanligtvis som *MRO*). MRO specificerar i vilken ordning basklasserna (om flera) till en klass Python söker efter metoder. I Python har klasser en metod `.mro` som visar klassens MRO. T.ex:

```
A.mro()
D.mro()
```

```
[__main__.A, object]
[__main__.D, __main__.B, __main__.A, object]
```

Här kan vi se att `D` ärver från `B` (som är den första klassen efter `D` som söks för om en metod finns `D`) och vidare från `A`. Den sista elementet i listan, `object`, är den *fundamentala byggstenen* för typer i Python och är därför sist i listan och både `A` och `D` har detta som en del av sin MRO. När det gäller Python får man ofta höra *"Everything is an object in Python"*:

```
isinstance(a, object)
isinstance(print, object) # function print is an object
issubclass(str, object)
issubclass(int, object)
```

```
True
True
True
True
```

10.4 Exempel: Geometriska figurer

Som ett exempel på arv ska vi nu implementera klasser som representerar geometriska objekt (figurer) som samlar vanliga/typiska operationer som man kan vilja göra på dessa figurer. Meningen är att med dessa klasser:

- Illustrera värdet av arv
- Illustrera *överlagring av metoder* (eng. *method overriding/overloading*)
- Illustrera värdet av *enkapsulering* (samling av data-attribut och de metoder som agerar på datat)
- Rita klassdiagram

Vi vill skriva kod som implementerar Python-objekt som representerar olika geometriska figurer som består av (raka) sidor. För att följa Liskovs substitutionsprincip, så vill vi ha en basklass som representerar det mest allmänna fallet: en polygon. Subklasserna ska lägga till data och funktionalitet som representerar en specifik typ av geometrisk figur.

Vi börjar med att definiera basklassen `Polygon`, som är menad att vara en klass som täcker det allmänna fallet men med vissa begränsningar.

```
class Polygon:
    def __init__(self, n_sides, n_unique_sides=None):
        self.n_sides = n_sides
        if n_unique_sides:
            self.n_unique_sides = n_unique_sides
        else:
            self.n_unique_sides = n_sides
        if self.n_sides % self.n_unique_sides:
            raise ValueError(
                "The total number of sides must be an integer"
                " multiple of the number of unique sides")
        self.sides = None
        self.__name__ = 'Polygon'

    # "Private" methods
    def _check_sides(self):
```

```
        if (not self.sides or
            not all(map(lambda x: x > 0, self.sides))):
            print("Invalid side lengths")
            return False
        else:
            return True

    def _get_side(self, side_no):
        s = float(input("Enter side " + str(side_no + 1) + ": "))
        return s

    # "Public" methods
    def input_sides(self):
        print("Please enter the " + str(self.n_unique_sides) +
              " unique side lenghts for a " + self.__name__)
        unique_sides = [self._get_side(i) for i in
                        range(self.n_unique_sides)]
        self.sides = unique_sides * (self.n_sides
                                     // self.n_unique_sides)
        if not self._check_sides():
            print("Please provide positive side lenghts")
            self.input_sides()

    def display_sides(self):
        if self._check_sides():
            for i in range(self.n_sides):
                print("Length of side", i + 1, ":", self.sides[i])
```

Klasskonstruktorn tar två argument, ett obligatorist argument `n_sides` som specificerar hur många sidor polygonen skall ha och ett nyckelordsargument `n_unique_sides` som har standardvärdet `None`. Vi vill här ge möjligheten att kunna specifiecera att en polygon har `n_sides` *antal* sidor men bara har ett visst antal unika sidlängder, t.ex så skulle `n_unique_sides=1` betyda att alla sidor i polygonen är lika långa.

I konstruktorn så sätts instansattributen till de värden som de formella parameterarna har; om `n_uniqe_sides` inte har specificerats så får instansattributet samma värde som `n_sides`, dvs att alla sidor anses unika. Vi vill dock begränsa hur en användare kan specificera antalet unika sidor i en polygon: det totala antalet sidor måste vara en heltalsmultipel av antalet unika sidor. Vi kollar att detta är uppfyllt genom att se om villkoret `n_sides % n_unique_sides` utvärderas till `True`, dvs om det finns en rest efter divisionen (ett tal skiljt från `0`) så lyfter vi ett `ValueError` (kom ihåg att alla tal som är skiljda från `0` utvärderas till `True` i Python).

Huvudmetoden i `Polygon` är `input_sides`, som när den kallas först skriver ut ett meddelande och sedan m.h.a en listomfattning läser in `n_unique_sides` antal sidlängder. Inläsningen av *varje enskild* sidlängd görs av den privata metoden `_get_sides`. Eftersom vi i konstruktorn redan sett till att `_n_unique_sides` har ett tillåtet värde, så kan vi multiplicera listan med (unika) inlästa sidlängder med `n_sides // n_unique_sides` för att slutligen få en lista som innehåller *alla* sidlängder (dvs `len(sides) == n_sides`): i fallet att vi har färre unika sidlängder än totalt antal sidlängder, så repeteras listan med de unika sidlängderna ett lämpligt antal gånger. Slutligen försäkrar vi oss att alla sidlängder är tillåtna med den privata metoden `_check_sides`, som med `map` och en anonym funktion (**`lambda`**) kontrollerar att alla sidor är positiva. Om de inte är det skrivs ett felmeddelande ut. Den första delen av villkoret för **`if`**-satsen

i `_check_sides` kontrollerar att `sides` är satt, dvs om `sides == None` så exekveras satsen *oavesett* om det andra villkoret är uppfyllt eller ej. Denna sats är av värde då `map` skulle ge ett fel om `sides == None`, eftersom `None` inte går att iterera över. Genom att uvärdera `if not sides` först så exekveras **`if`**-satsen om `sides == None` (och `map` exekveras aldrig) *eller* så är `sides` satt (vi antar att det är en lista) och `map` kan iterera över `sides`.

Metoden `display_sides` skriver ut alla sidlängder om `sides` innehåller giltiga sidlängder.

Nu när vi har en basklass, så kan vi lätt genom arv definiera subklasser som representerar specifika polygoner. T.ex., så kan en triangel-typ representeras av:

```
class Triangle(Polygon):
    def __init__(self):
        super().__init__(3)
        self.__name__ = 'Triangle'

    # method specific to triangles
    def get_area(self):
        a, b, c = sorted(self.sides)
        if c > (a + b):
            print("Non-valid side lengths for a triangle.")
            return
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        return (s * (s - a) * (s - b) * (s - c))**0.5
```

Här ser vi att `Triangle` ärver från `Polygon` och att vi i `Triangles` konstruktor kallar *basklassens* konstruktor med ett argument (`3`). Att vi explicit kallar konstruktorn med `super` gör att `Polygon` instantieras med `n_sides=3, n_unique_sides=None` och att en instans av `Triangle` ärver både instansattribut och metoder från `Polygon`. Vi överlagrar instansattributet `__name__` så att namnet för klassinstansen ges som `Triangle`. Vi *utökar* basklassens funktionalitet genom att lägga till en metod `get_area` som beräknar en triangels area om triangeln har giltiga sidlängder.

På ett liknande sätt kan vi definiera en rektangel. Rektanglar har fyra sidor, men endast två *unika* sidlängder. Det här löses enkelt genom att definiera en klass `Rectangle` som ärver från `Polygon` men som kallar basklassens konstruktor med argument som specificerar just detta, dvs en polygon som har 4 sidor men bara 2 unika sidlängder.

```
class Rectangle(Polygon):
    def __init__(self):
        super().__init__(4, n_unique_sides=2)
        self.__name__ = 'Rectangle'

    def get_area(self):
        return self.sides[0] * self.sides[1]

    # method specific for rectangles
    def get_diagonal(self):
        return (self.sides[0]**2 + self.sides[1]**2)**0.5
```

`Rectangle` definierar också metoder som är specifika för rektanglar: `get_area` och `get_diagonal`.

Slutligen så definierar vi ett specialfall av en rektangel, nämligen en kvadrat `Square`. En kvadrat är en

rektangel, så både area och diagonal beräknas på samma sätt som i `Rectangle`. Det är därför lämpligt att ärva funktionaliteten av `Rectangle`.

```
class Square(Rectangle):
    def __init__(self):
        super(Rectangle, self).__init__(4, n_unique_sides=1)
        self.__name__ = 'Square'

    def get_corner_coordinates(self):
        s = self.sides[0]/2
        return ((-s, -s), (-s, s), (s, s), (s, -s))
```

Skillnaden mellan en kvadrat och en rektangel är att det bara finns *en unik* sidlängd. I dett fall kan vi *inte* kalla på basklassens konstruktor i `Square` för att ärva instansattribut, eftersom basklassens (`Rectangle`) konstruktor inte tar några argument utan implicit sätter `n_sides = 4` och `n_unique_sides = 2`. Vi skulle vilja kalla konstruktorn till `Rectangles` basklass (dvs `Polygon`). Detta kan åstadkommas genom att vi i `Squares` konstruktor använder oss av `super` med ett argumenten `Rectangle` och `self` för att säga att vi vill initiera basklassen *till* `Rectangle`, men att det ska gälla för klassinstansen av `Square` (`self`). Vi ärver fortfarande från `Rectangle` (t.ex. metoderna `get_area` och `get_diagonal`), men konstruktorn körs som `Polygon(4, n_unique_sides=1)` för att begränsa antalet unika sidlängder) till *en* men fortfarande ha (totalt) fyra sidor.

Man kan nu använda sig av ovan definierade klasser:

```
t = Triangle()
t.input_sides()
print('The area of the triangle is:', t.get_area())
t.display_sides()

r = Rectangle()
r.input_sides()
print('The area of the rectangle is:', r.get_area())
print('The diagonal of the rectangle is:', r.get_diagonal())

s = Square()
s.input_sides()
print('The area of the square is:', s.get_area())
print('The diagonal of the square is:', s.get_diagonal())
print('The coordinates of the square corners are:\n',
      s.get_corner_coordinates())
```

```
Please enter the 3 unique side lengths for a Triangle
Enter side 1 : 1
Enter side 2 : 2.5
Enter side 3 : 2
The area of the triangle is: 0.9499177595981665
Length of side 1 : 1.0
Length of side 2 : 2.5
Length of side 3 : 2.0
Please enter the 2 unique side lengths for a Rectangle
Enter side 1 : -1
```

```
Enter side 2 : 2
Invalid side lengths.
Please provide positive side lenghts.
Please enter the 2 unique side lengths for a Rectangle
Enter side 1 : 1.2
Enter side 2 : 1.9
The area of the rectangle is: 2.28
The diagonal of the rectangle is: 2.247220505424423
Please enter the 1 unique side lengths for a Square
Enter side 1 : 6
The area of the square is: 36.0
The diagonal of the square is: 8.48528137423857
The coordinates of the square corners are:
 ((-3.0, -3.0), (-3.0, 3.0), (3.0, 3.0), (3.0, -3.0))
```

10.4.1 Implementera jämförelser

Som vi gick igenom i kapitel 9, så kan man genom att använda sig av specifika privata metoder implementera jämförelser för sin egen typ (klass). Dessa metoder fungerar väldigt bra att ärva från en basklass när någon logisk jämförelse mellan objekt kan göras.

Antag t.ex. att vi vill implementera <, >, och == baserat på arean av ett objekt. För de geometriska figurerna `Triangle`, `Rectangle` och `Square` så kan vi ju genom `get_area` räkna ut arean av ett objekt. Vi kan lägga till nedan funktioner i `Triangle` och `Rectangle`:

```
def __lt__(self, other):
    return self.get_area() < other.get_area()

def __gt__(self, other):
    return self.get_area() > other.get_area()

def __eq__(self, other):
    return self.get_area() == other.get_area()
```

Lägg märke till att vi *inte* behöver lägga till detta i `Square`, då denna klass ärver från `Rectangle`.

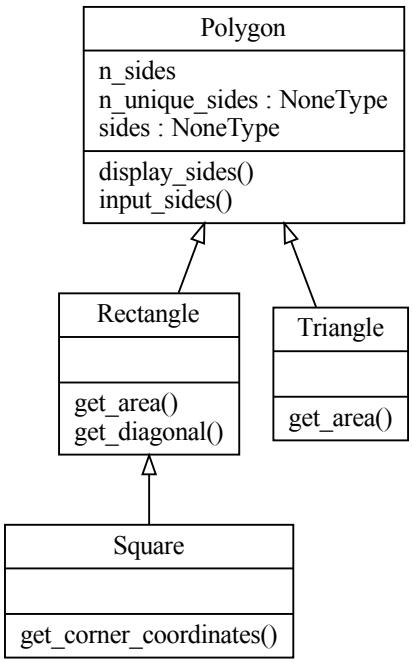
I fig. 10.2 illustrerar vi klassdiagram för det här exemplet.

10.5 Exempel: Monster Go

Följande exempel är inspirerat av Pokemon Go och är ett lite större exempel. Exemplet är lite konstruerat, men det är för att framhäva viktiga poänger som illustrerar värdet av objekt-orienterad programmering.

Målet med detta exempel är att:

- Illustrera värdet av arv
- Illustrera värdet av abstraktion: dölja implementationsdetaljer.



Figur 10.2: Klassdiagram för geometriska figurer.

10.5.1 En basklass: ProgMon

Klassen `ProgMon` innehåller attribut och metoder som ska vara gemensamma för alla objekt. Vi försöker alltså även här följa Liskovs substitutionsprincip.

Vi kommer i följande exempel att använda oss mycket av privata metoder och attribut (både de som används av Python och börjar/slutar med `__`, t.ex. `__str__` och de som är privata för vår implementation och börjar med `_`, t.ex. instansattributen `_attack`).

```
class ProgMon():
    def __init__(self):
        self._attack = 0.0
        self._defense = 0.0
        self._caffeinated = False
        self._unit_testing = False

    def __str__(self):
        return "<ProgMon object>"

    def get_attack(self):
        if self._caffeinated:
            return 2 * self._attack
        else:
            return self._attack

    def get_defense(self):
        if self._unit_testing:
            return 2 * self._defense
        else:
            return self._defense

    def fight(self, other_progmon):
        '''
        Attack is the best form of defense!
        Returns (True, False) if self wins, (False, True) if other_progmon
        wins and (False, False) otherwise.
        '''
        if self.get_attack() > other_progmon.get_defense():
            return (True, False)
        elif other_progmon.get_attack() > self.get_defense():
            return (False, True)
        else:
            return (False, False)
```

10.5.2 Tre subklasser

I subklasserna `Hacker`, `Newbie`, och `Guru` lägger vi in specialisering (data och beteende som är specifika för klassen). Vi gör detta genom att ändra värden för de privata attributen och definiera `__str__`.

```
class Hacker(ProgMon):
    def __init__(self):
        super().__init__()  # Call the superclass constructor!
        self._attack = 0.5
        self._defense = 0.25

    def __str__(self):
        return "<Hacker A=" + str(self._attack) + ">"

class Newbie(ProgMon):
    def __init__(self):
        super().__init__()
        self._attack = 0.15
        self._defense = 0.1

    def __str__(self):
        return "<Newbie>"

class Guru(ProgMon):
    def __init__(self):
        super().__init__()
        self._attack = 1.0
        self._defense = 1.0
```

Provkör med:

```
h = Hacker()
n = Newbie()
g = Guru()
n.fight(h)
g.fight(h)
```

```
(False, True)
(True, False)
(True, False)
```

Obs: Gemensam funktionalitet i basklassen gör att vi undviker duplicering av kod.

10.5.3 Anpassning

Antag att man inte gillar tanken på att en `Guru` försöker vinna kamper. Vi kan implementera en "Don't win, don't lose" för en `Guru` som initierar en "fight", genom att definiera en ny version av `Guru`:

```
class Guru(ProgMon):
    def __init__(self):
        super().__init__()
        self._attack = 1.0
        self._defense = 1.0

    # method overloading
```

```
    def fight(self, other_progmon):
        '''
        Don't win, don't loose.
        '''
        # same type of return value as in the super class
        return (False, False)
```

Provkör med ny definition: `g.fight(h)` ger `(False, False)`.

Obs: vi kan ändra beteendet hos en subklass genom att definiera om en metod i subklassen.

10.5.4 Abstraktion undviker detaljkunskap

Basklassen `ProgMon` definierar ett *gränssnitt* som ger en praktisk abstraktion: du behöver inte kunna några *detaljer* om basklassen eller dess subklasser om du använder gränssnittet.

10.5.5 Exempel: en hacker dojo

Antag att vi skapar en klass för en "Hacker Dojo", där ProgMon:s samlas för att träna. Man ska kunna utmana en Hacker Dojo, vilket kräver att man vinner en kamp mot varje medlem för att kunna säga att man vunnit. Förlust mot en medlem ger förlust av utmaningen.

```
class HackerDojo():
    def __init__(self):
        self._members = []

    def add_member(self, m):
        self._members.append(m)

    def challenge(self, pm):
        for monster in self._members:
            win, loose = pm.fight(monster)
            if loose:
                # Lost against one member, challenge failed.
                return False
        # Won against all members of the dojo
        return True
```

Utprovning:

```
dojo = HackerDojo()
dojo.add_member(h)
dojo.add_member(n)
dojo.challenge(g)
dojo.challenge(n)
```

```
True
False
```

Obs: det enda vi använder i implementationen är gränssnittet från `ProgMon`. Vi får automatiskt den lite annorlunda specialiseringen från `Guru`. Eller snarare: vi behöver inte oroa oss för sådana detaljer.

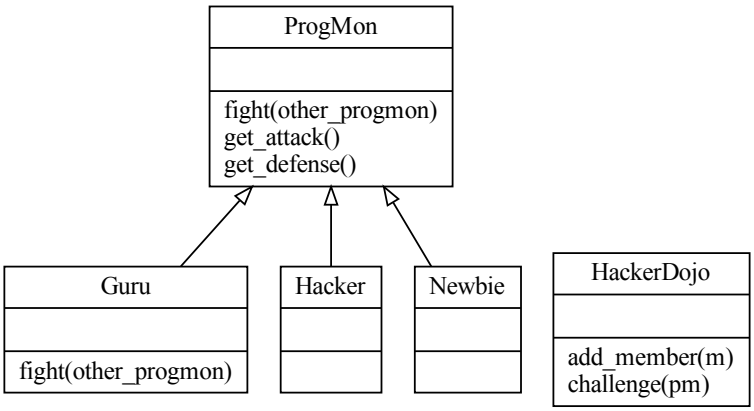
10.5.6 Ändrade detaljer

En av de stora vinsterna med objektorientering (men inte unikt för OO) är abstraktion. Genom att sätta ihop bra gränssnitt (dvs väl valda metoder) "framtidssäkrar" vi ett program: det blir lätt att ändra detaljer i en del av programmet, utan att andra delar påverkas. Detta underlättar ofantligt mycket i större kodprojekt som kan ha tusentals eller miljoner rader kod.

Antag att vi inser att `_attack` inte är ett bra attribut. Styrkan ges av erfarenhet, level, och andra attribut som blir specifika för subklasserna.

- Vilka ändringar behövs göras, och vilka klasser påverkas?
- Är `get_attack` en bra metod?

I fig. 10.3 illustrera vi det här exemplet med ett klassdiagram.



Figur 10.3: Klassdiagram för Pokemon Go.

10.6 Uppgifter

1. Skapa en klass `Equilateral` som ärver från `Triangle` i exemplet ovan och metodöverlagrar `get_area`.

   Tips: Arean för en liksidig triangel är $\sqrt{3}a^2/4$, där a är sidlängden. Man kan importera från standardmodulen `math` importera funktionen `sqrt`.

2. Skapa en klass `Student` som ärver från `ProgMon` (även instansattribut), men som lägger till ett instansattribut `is_learning` av typen `bool` och metodöverlagrar både `get_attack` och `get_defence` i `ProgMon`. Klassen `Students` metoder `get_attack` och `get_defence` skall använda

sig av `ProgMon` `get_attack` och `get_defence` men ska multiplicera styrkan med två om `Student` lärs, dvs om `is_learning` är `True`. Du kan bestämma vad instansattributen `_attack` och `_defense` instansieras till i klassen själv.

3. I uppgift 5 och 6 i kapitel 9 implementerade vi en class `Dog`. Implementera en klass `Tournament` som

   1. rankar hundarna baserat på antal tricks de kan, och
   2. skriver ut resultaten.

   Tips: hämta inspiration från hur `HackerDojo` relaterar till `ProgMon`.

4. Skriv en basklass `Vehicle`, som innehåller instansattributen `wheels`, `wings` och `sound`. Utöver detta ska `Vehicles` metod `__str__` skriva ut information om fordonet (hur många hjul, etc., fordonet har). Skapa tre subklasser `Car`, `Motorcycle` och `Plane` som alla ärver från `Vehicle`, men har rätt antal hjul/vingar och gör "rätt" ljud.

   Exempel-användning och output skulle vara:

```
car = Car()
mc = Motorcycle()
plane = Plane()
print(car)
print(mc)
print(plane)
```

```
A Car has:
4 wheels
and makes a AAARRRRRRRR! sound

A Motorcycle has:
2 wheels
and makes a VROOM VROOM! sound

A Plane has:
2 wings
and makes a WHOOSH! sound
```

5. Skriv en egen sträng-klass som "hanterar" subtraktion och division. Med *subtrahera* en sträng menar vi här att i strängen man subtraherar från (vänster om `-` operatorn) den första förekomsten *från höger* av strängen som subtraheras (höger om `-` operatorn) tas bort. Med *division* menas att alla förekomster av strängen till höger om `/` operatorn tas bort från huvudsträngen.

   Tips: skapa en subklass `MySring` till `str`, och läs på om vilka privata metoder som behövs implementeras för `-` och `/`.

   Exempel:

```
s1 = MyString("This is my own string!")
s2 = MyString("My, oh my, oh Oh.... oh")
subtract = MyString('my')
div = MyString('oh')
print(s1 - subtract)
print(s1 / div)
```

```
print(s2 - subtract)
print(s2 / div)
print(s1 - 'hello')
print(s1 / 'hello')
```

```
This is  own string!
This is my own string!
My, oh , oh Oh.... oh
My,  my,  Oh....
This is my own string!
This is my own string!
```

Kapitel 11

Objektorientering 3: mer om arv

11.1 Olika typer av arv

I förra kapitlet såg vi olika exempel på arv i Python. Olika former av arv har olika namn inom objektorientering:

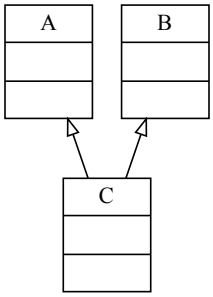
1. Enkelt arv (eng. *single inheritance*): klass `B` ärver endast från klass `A`.
2. Mångnivå arv (eng. *multilevel inheritance*): klass `B` ärver från `A` och klass `C` ärver sen från klass `B`.
3. Hierarkiskt arv (eng. *hierarcical inheritance*): klass `A` är superklass till `B`, `C`, `D`...
4. Multipelt arv (eng. *multiple inheritance*): klass `C` ärver från *både* `A` och `B` (men de ärver inte från varandra).
5. Hybrid arv (eng. *hybrid inheritance*): en mix av två eller fler typer av arv.

Vi såg även att man kan illustrera olika typer av arv med hjälp av klassdiagram. Förra kapitlet innehöll exempel på enkelt, mångnivå, hierarkiskt och hybrid arv, så vi ska nu titta på multipelt arv.

Det är inte alla programmeringsspråk som tillåter multipelt arv, men det gör Python. Betrakta följande exempel:

```
class A:
   pass    # some code

class B:
   pass    # some more code

class C(A, B):
   pass    # even more code
```

Vi kan illustrera detta med hjälp av ett klassdiagram:



Figur 11.1: Klassdiagram för `A`, `B` och `C`

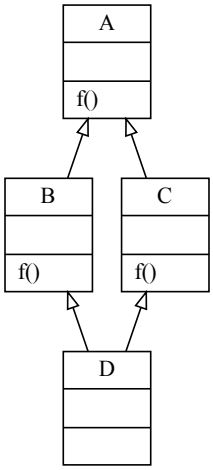
Här har `C` alltså två superklasser, `A` och `B`, vilka inte ärver från varandra. Detta kan verka oskyldigt och man kan undra varför inte alla språk stödjer det. Men rent generellt är det lätt hänt att multipelt arv ger kaos på grund av så kallade *diamantproblem*.

11.1.1 Diamantproblem

Betrakta följande kod:

```
class A:
    def f(self):
        print("f of A called")

class B(A):
    def f(self):
        print("f of B called")

class C(A):
    def f(self):
        print("f of C called")

class D(B,C):
    pass

d = D()
d.f()
```

Detta är ett typiskt diamantproblem då om man ritar upp klassdiagrammet får man en romb:



Figur 11.2: Klassdiagram för `A`, `B`, `C` och `D`

Kör vi koden ovan får vi

```
f of B called
```

Men om vi istället kör följande där arvet är `D(C,B)` får vi:

```
class A:
    def f(self):
        print("f of A called")

class B(A):
    def f(self):
        print("f of B called")

class C(A):
    def f(self):
        print("f of C called")

class D(C,B):
    pass

d = D()
d.f()
```

```
f of C called
```

Så utdata har nu ändrats då man ärver i en annan ordning! Detta är ett exempel på ett "diamantproblem" som kan uppstå med multipelt arv. Man kan lätt föreställa sig hur kod snabbt blir oöverskådlig om man har många klasser med en massa metoder och attribut vilka ärver från varandra på detta sätt. Denna typ

av problem är vad som lätt orsakar kaos i kod med multipelt arv och varför många språk inte tillåter det.

11.2 Exempel: Algebraiska uttryck

Vi ska nu titta på ett till exempel på arv. Poängen med detta exempel är att illustrera två saker:

1. Hur man kan använda klasser för att *strukturera* data.
2. Hur man kan använda rekursion i kombination med klasser.

Denna kombination är väldigt kraftfull och låter oss lösa många olika typer av programmeringsproblem på ett enkelt sätt. Vi börjar med att skriva ett program för att representera och modifiera algebraiska uttryck. Dessa uttryck kan innehålla tal, variabler, `+`, `*` och *matchade* paranteser. Ett exempel på ett algebraiskt uttryck är `x * y + 7` och ett annat är `x * (y + 7)`.

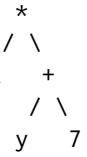
Hur ska vi tänka för att representera dessa i Python? En naiv lösning skulle kunna vara att helt enkelt använda strängar, så `x * y + 7` skulle representeras som strängen `"x * y + 7"`. Men låt oss säga att vi vill byta ut `x` till `3` och `y` till `2` och sedan beräkna värdet av uttrycket… Detta blir **väldigt** komplicerat med strängar!

Ett annat problem med strängar är hur man ska hantera paranteser. Enligt konventionen att `*` binder starkare än `+` så ska ju `x * y + 7` tolkas som `(x * y) + 7` och inte `x * (y + 7)`. Har man inte koll på detta kommer man ju få fel svar när man utvärderar uttryck. Detta är även det komplicerat att hålla redan på med strängar! Ett mycket smidigare sätt är att använda **träd**. På så sätt kan vi representera algebraiska uttryck med två dimensioner istället för bara en!

Ett träd är en datastruktur där ett element antingen är ett löv eller en nod med ett antal grenar som leder till "underträd" (eng. *subtree*). Så `(x * y) + 7` kan representeras entydigt som ett träd med översta noden `+` och två underträd som innehåller `x * y` och `7`. Värdet `7` är ett löv i trädet då det inte har några underträd. Uttrycket `x * y` är även det ett träd, denna gång med noden `*` längst upp och `x` och `y` som löv. Vi kan rita detta som:

Obs: dataloger ritar träd upp-och-ned jämfört med riktiga träd!

Uttrycket `x * (y + 7)` är istället följande träd:



Denna typ av träd kallas **binära** träd då alla noder har två eller noll underträd. Låt oss nu representera denna typ av träd med algebraiska uttryck i Python! Vi börjar med en tom superklass för uttryck som i sin tur har 3 subklasser (en för siffror, en för variabler och en för `+`):

```
class Expr:
    pass

class Constant(Expr):

    def __init__(self, value):
        self._value = value

    def __str__(self):
        return str(self._value)

class Var(Expr):

    def __init__(self, name):
        self._name = name

    def __str__(self):
        return self._name

class Plus(Expr):

    def __init__(self, left, right):
        self._left = left
        self._right = right

    def __str__(self):
        return str(self._left) + " + " + str(self._right)

print(Plus(Constant(3),Var("x")))
```

```
3 + x
```

Notera att anropet till `str` är *rekursivt* i `__str__` för `Plus`. Här anropas `str` metoden för vänstra och högra underträdet. Vilken funktion som sen kommer köras beror på vad dessa är.

Låt oss nu lägga till `*`, detta kan göras genom:

```
class Times(Expr):

    def __init__(self, left, right):
        self._left = left
        self._right = right

    def __str__(self):
        return str(self._left) + " * " + str(self._right)

print(Plus(Times(Var("x"),Var("y")),Constant(7)))
```

```
x * y + 7
```

Men denna definition är ju exakt samma som `Plus`, bortsett från att vi skriver ut `*` istället för `+`. Kopiera

och klistra in kod är farligt då man lätt introducerar buggar! En bättre lösning är att istället introducera en ny klass för binära operatorer vilken `Plus` och `Times` sen ärver ifrån:

```
class BinOp(Expr):

    def __init__(self, left, right):
        self._left = left
        self._right = right

class Plus(BinOp):

    def __str__(self):
        return str(self._left) + " + " + str(self._right)

class Times(BinOp):

    def __str__(self):
        return str(self._left) + " * " + str(self._right)

print(Plus(Times(Var("x"),Var("y")),Constant(7)))
```

```
x * y + 7
```

Detta verkar ju fungera bra, men det blir fel om vi skriver:

```
print(Times(Var("x"),Plus(Var("y"),Constant(7))))
```

```
x * y + 7
```

Vi får exakt samma som ovan! Detta är knas då vi glömt bort att sätta in paranteser. En naiv lösning på detta problem är att alltid sätta in parenteser runt underuttrck i `Plus` och `Times`:

```
class BinOp(Expr):

    def __init__(self, left, right):
        self._left = left
        self._right = right

class Plus(BinOp):

    def __str__(self):
        return "(" + str(self._left) + ") + (" + str(self._right) + ")"

class Times(BinOp):

    def __str__(self):
        return "(" + str(self._left) + ") * (" + str(self._right) + ")"

print(Plus(Times(Var("x"),Var("y")),Constant(7)))

print(Times(Var("x"),Plus(Var("y"),Constant(7))))
```

```
((x) * (y)) + (7)
(x) * ((y) + (7))
```

Detta är ju rätt, men vi har satt in onödigt många paranteser! Vad vi egentligen vill få ut är:

```
print(Plus(Times(Var("x"),Var("y")),Constant(7)))

print(Times(Var("x"),Plus(Var("y"),Constant(7))))
```

```
x * y + 7
x * (y + 7)
```

Hur kan vi lösa detta?

Lösning: dekorera alla uttryck med hur hårt de binder. Med denna extra information kan vi avgöra om vi ska skriva ut paranteser runt underuttryck eller inte. Vi gör detta genom att lägga till en klassvariabel `prec` (för *precedence*) till `Expr` klassen och sen skriva över den i `Plus` och `Times`. Vi har sedan en hjälpfunktion `parens` som avgör om paranteser ska skrivas ut runt ett uttryck eller inte.

```
class Expr:

    # default precedence very high so that we don't put parantheses
    prec = 1000

class Constant(Expr):

    def __init__(self, value):
        self._value = value

    def __str__(self):
        return str(self._value)

class Var(Expr):

    def __init__(self, name):
        self._name = name

    def __str__(self):
        return self._name

class BinOp(Expr):

    def __init__(self, left, right):
        self._left = left
        self._right = right

# Function for inserting parentheses around a string if precedence p1
# is smaller than precedence p2
def parens(p1,p2,s):
    if p1 < p2:
        return "(" + s + ")"
```

```
    else:
        return s

class Plus(BinOp):

    prec = 1

    def __str__(self):
        s1 = parens(self._left.prec,self.prec,str(self._left))
        s2 = parens(self._right.prec,self.prec,str(self._right))
        return s1 + " + " + s2

class Times(BinOp):

    prec = 2

    def __str__(self):
        s1 = parens(self._left.prec,self.prec,str(self._left))
        s2 = parens(self._right.prec,self.prec,str(self._right))
        return s1 + " * " + s2


print(Plus(Times(Var("x"),Var("y")),Constant(7)))

print(Times(Var("x"),Plus(Var("y"),Constant(7))))

print(Times(Var("x"),Times(Var("y"),Plus(Constant(3),Var("z")))))
```

```
x * y + 7
x * (y + 7)
x * y * (3 + z)
```

11.3 Exempel: Binära träd

Vi kan generalisera datatypen för algebraiska uttryck till godtyckliga binära träd. På detta sätt kan vi skriva våra funktioner en gång för alla och kommer inte behöva skriva om dem för varje typ av binära träd vi kan tänkas behöva.

Ett binärt träd är antingen en förgrening (med nodvärde och vänster/höger underträd) eller ett löv (med värde). Vi lägger in följande funktionalitet i dessa:

- `__str__`: konvertera till en sträng.
- `member(x)`: testa om `x` finns i trädet.
- `map_tree(f)`: applicera funktionen `f` på alla värden i trädet (notera att detta är en högre ordningens funktion då den tar en funktion som argument).
- `linearize()`: konvertera trädet till en lista.

I detta exempel är huvudklassen `BinTree` tomt, men det är ändå bra att ha med om man en dag vill lägga in något i den som vi gjorde med `prec` värdet för algebraiska uttryck.

```
class BinTree:
    pass

class Branch(BinTree):

    def __init__(self, node, left, right):
        self._node = node
        self._left = left
        self._right = right

    def __str__(self):
        n , l , r = str(self._node) , str(self._left) , str(self._right)
        return "(" + n + "," + l + "," + r + ")"

    def member(self, x):
        return self._node == x or self._left.member(x) or self._right.member(x)

    def map_tree(self,f):
        return Branch(f(self._node),self._left.map_tree(f),self._right.map_tree(f))

    def linearize(self):
        return self._left.linearize() + [self._node] + self._right.linearize()

class Leaf(BinTree):

    def __init__(self, val):
        self._val = val

    def __str__(self):
        return "(" + str(self._val) + ")"

    def member(self, x):
        return self._val == x

    def map_tree(self, f):
        return Leaf(f(self._val))

    def linearize(self):
        return [self._val]

t = Branch(-32,Leaf(2),Branch(1,Branch(23,Leaf(4),Leaf(-2)),Leaf(12)))

print(t)
print(t.member(12))
print(t.member(-123))
```

```
print(t.map_tree(lambda x: 0))
print(t.map_tree(lambda x: x ** 2))

print(t.linearize())
```

```
(-32,(2),(1,(23,(4),(-2)),(12)))
True
False
(0,(0),(0,(0,(0),(0)),(0)))
(1024,(4),(1,(529,(16),(4)),(144)))
[2, -32, 4, 23, -2, 1, 12]
```

Notera att alla funktioner som vi definierat är rekursiva. Detta exempel visar hur vi på ett enkelt sätt kan konvertera kombinera klasser och rekursion för att representera och modifiera strukturerad data.

11.4 Uppgifter

1. Vilka av exemplen i förra kapitlet är exempel på enkelt, mångnivå, hierarkiskt och hybrid arv?
2. Flytta `__str__` metoden för `Plus` och `Times` till `BinOp` på lämpligt sätt så att vi inte har någon kodduplikation. Exemplen ska fortfarande skrivas ut på rätt sätt.
3. Lägg till en klass `Exp` för exponentiering av algebraiska uttryck. Den ska binda hårdare är både `+` och `*`, samt skrivas ut som operatorn `**` (dvs på samma sätt som i Python).
4. Lägg till en funktion `big_constant_sum(n,e)` där `n` är ett positivt heltal och `e` ett algebraiskt uttryck. Funktionen ska skapa ett uttryck där `e` adderas med sig själv `n` gånger. Om `n` är `0` så ska konstanten `0` returneras. Så funktionen ska alltså returnera summan

$$\sum_{i=0}^{n-1} e$$

5. Lägg till en funktion `big_sum(n,e)` där `n` är ett positivt heltal och `e` ett funktion från heltal till algebraiska uttryck. Funktionen ska skapa ett uttryck som representerar

$$\sum_{i=0}^{n} e_i$$

6. Definera evaluering av algebraiska uttryck givet en uppslagstabell `d` från variabler till siffror. För att göra detta ska ni lägga till en metod `evaluate(d)` till `Constant`, `Var`, `Plus`, `Times` och `Exp`. Ni kan anta att alla variabler i uttrycket finns definierade i `d`.
7. † Implementera roträd. Dessa är en typ av träd där varje nod innehåller ett värde och en lista med fler roträd. Ett löv med värdet `5` representeras genom `RoseTree(5,[])`. Implementera en klass för roträd som har samma metoder som `BinTree`.

Kapitel 12

Defensiv programmering

Den som har lärt sig programmera har nog också blivit tvungen att lära sig felsöka. Alla programmerare gör fel, råkar förlita sig på olämpliga antaganden, och missförstår dokumentation. Fel kan göra att programmet avbryter precis där misstaget är, men det kan också uppstå långt senare. Om division med noll uppstår så kanske det är ett symptom på ett tidigare misstag. En variabel kanske har blivit satt till 0, trots att det är orimligt, men var gjordes det och varför? Det kan vara arbetsamt att leta upp orsaken till felet. Programmeringsstilen *defensiv programmering* syftar dels till att hitta fel tidigt, så nära det egentliga misstaget, för att undvika arbetsamma felsökningssessioner, och dels till att tydliggöra sina antaganden i koden.

Defensiv programmering avser att man förväntar sig att funktioner kommer att användas på fel sätt, algoritmer blir implementerade fel, kombinationer av variablers värden uppstår som man inte har tänkt på, och att andra konstigheter sker under programutveckling. Genom att *koda* sina förväntningar på variablers värden med hjälp av `assert` på många platser i ett program kan man upptäcka felen tidigare, närmare felkällan, och därmed förenkla felsökningen.

12.1 Assert

Instruktionen `assert condition` anger att uttrycket `condition` förutsattes vara sant. Om uttrycket är sant händer inget, det är det förväntade uppträdandet, men om det är falskt avbryter programmet med hjälp av ett särfall, `AssertionError`. Denna typ av instruktion finns i många språk och är ett hjälpmedel för felsökning, utveckling, och underhåll av program.

Några exempel:

```
assert len(data_list) > 0
assert x != 0
assert x > 0 and y > 0
```

Dessa rader ställer tre tänkbara krav på tillståndet i ett program. Instruktionen `assert` ska utläsas som “detta gäller”.

12.1.1 Avbrott och felmeddelande

Säg att vi har skrivit en divisionsfunktion på följande sätt:

```
def divide(x,y):
    assert y != 0
    return x / y
```

Om vi nu anropar `divide` med `0` som andra argument

```
print(divide(10,2))
print(divide(10,0))
```

så kommer ett felmeddelande skrivas ut:

```
Traceback (most recent call last):
  File "f12.py", line 6, in <module>
    print(divide(10,0))
  File "f12.py", line 2, in divide
    assert y != 0
AssertionError
```

Notera att särfallet `AssertionError` inte är tänkt att fångas med **`except`**; man använder **`assert`** för att hitta allvarliga programmeringsfel, inte för att hantera körfel (som t.ex., namngiven fil finns ej, minnet räcker inte, man har rekurserat för djupt, o.s.v.).

12.1.2 Generellt

Det generella fallet kan skrivas

```
assert condition [, comment_string]
```

där `condition` är ett boolskt uttryck och `comment_string` är en möjlig förklarande text. Man kan alltså skriva villkoren ovan som:

```
assert len(data_list) > 0, 'User submitted empty data'
assert x != 0, 'x==0 must be avoided early in the application!'
assert x > 0 and y > 0, 'Both x and y has to be positive!'
```

för att hjälpa till att förstå felen som uppstår.

Exempel:

```
def divide(x,y):
    assert y != 0, "Second argument cannot be 0 when dividing"
    return x / y

print(divide(10,0))
```

ger

```
Traceback (most recent call last):
  File "f12.py", line 6, in <module>
    print(divide(10,0))
```

```
  File "f12.py", line 2, in divide
    assert y != 0, "Second argument cannot be 0 when dividing"
AssertionError: Second argument cannot be 0 when dividing
```

12.1.3 Tumregler för `assert`

`assert` är implementerat med hjälp av särfall, men syftet är att uttrycka antaganden i koden, inte att hantera oväntade tillstånd.

- **`assert`** är för problem under kodning/utveckling.
- **`raise`** och **`try`**/**`except`** är för oväntade effekter under körning.

Man kan säga att **`assert`** används för att fånga oväntade tillstånd i koden, **`raise`** används för att signalera eventuella förväntade fel och **`try`**/**`except`** används för att fånga mer eller mindre förväntade fel. Ett program ska inte vara beroende av `assert`, utan fungera minst lika bra utan. I allmänhet går ett program långsammare med **`assert`** eftersom det är mer kod att exekvera, men man ska inte dra sig för att lägga till **`assert`** i sin kod. Om snabbhet är viktigt kan man starta sitt program med `python -O kod.py` istället för `python kod.py`; flaggan `-O` deaktiverar alla **`assert`**-instruktioner.

12.1.3.1 Exempel 1: `remove_trailing_spaces`

Nedan följer ett exempel på **`assert`** i `remove_trailing_spaces`:

```
def remove_trailing_spaces(s, spaces=" \t"):
    assert len(spaces) > 0
    i = len(s)
    while i > 0 and s[i-1] in spaces:
        i -= 1
    return s[0:i]
```

12.1.3.2 Exempel 2: `newton_raphson`

Nedan följer ett exempel på **`assert`** i en funktion för att beräkna roten av en funktion med hjälp av Newton-Raphsons algoritm:

```
def newton_raphson(f, f_prime, x, precision):
    assert isinstance(x, float)
    assert isinstance(f(x), float)
    assert isinstance(f_prime(x), float)
    assert  0.0 < precision < 1.0, 'precision must be in the interval (0,1)'

    ready = False
    while not ready:
        x_next = x - f(x)/f_prime(x)      # Main line of code to be repeated
        ready = abs(x-x_next) < precision # Test for stopping criterion
        x = x_next                        # Preparing the next iteration

    return x_next
```

Som du ser kan vi testa typen för ett objekt med hjälp av `isinstance` . Funktionen `type`, som returnerar objektets typ, är också användbar, men `isinstance` är mer generell; `isinstance` kan som andra argument ta en tupel av klasser och returnerar om det första argumentet är en instans av *någon* av klasserna *eller* dess subklasser.

Observera även att **`assert`** kan ses som extra dokumentation kring de antagande som görs i programmet.

12.1.4 Villkor och invarianter

Vi kan dela upp **`assert`**-användandet i tre sorter:

- **förvillkor**: uttryck som ska vara sanna när en funktionskropp exekveras (eng. *preconditions*)
- **eftervillkor**: uttryck som ska vara sanna när en funktion returnerar (eng. *postconditions*)
- **loop invarianter**: uttryck som ska vara sanna i en loop (eng. *loop invariants*)

12.1.4.1 Exempel 3

Låt oss säga att vi har en databas som kopplar unika identifierare (representerade som `int`) med namn (representerade som `str`). För att det ska gå snabbt att slå upp någon genom antingen dess identifierare eller namn har vi två uppslagstabeller, en från identifierare till namn och en från namn till identifierare. För att detta ska fungera måste ju nycklarna i den ena vara samma som värdena i den andra, och vice versa.

```
class MyDB:

    def __init__(self):
      self._id2name_map = {}
      self._name2id_map = {}

    def add(self, id, name):
      self._name2id_map[name] = id
      self._id2name_map[id] = name

    def by_name(self, name):
      id = self._name2id_map[name]
      return id
```

En första sak som kan bli fel här är ju att användaren helt enkelt skriver över en av `_id2name_map` och `_name2id_map`, eller att någon annan metod vi senare lägger till har en bugg som bryter mot invarianten för klassen. Vi kan lägga in en **`assert`** i `by_name` där vi dubbelkollar att invarianten ej har blivit bruten:

```
class MyDB:

    def __init__(self):
      self._id2name_map = {}
      self._name2id_map = {}

    def add(self, id, name):
      self._name2id_map[name] = id
      self._id2name_map[id] = name
```

```
    def by_name(self, name):
      id = self._name2id_map[name]
      assert self._id2name_map[id] == name
      return id
```

Om vi nu lägger in en buggig metod eller en användare använder klassen på fel sätt så kommer `by_name` generera ett `AssertionError`. En annan invariant är ju att ett id ska vara en `int` och namn `str`. I många programmeringsspråk med striktare typsystem kan vi specificera detta i signaturen för klassen eller konstruktorn. I Python kan vi lösa detta genom att lägga in en **`assert`** tillsammans med `isinstance`:

```
class MyDB:

    def __init__(self):
      self._id2name_map = {}
      self._name2id_map = {}

    def add(self, id, name):
      assert isinstance(id, int), ("id is not an integer: " + str(id))
      assert isinstance(name, str), ("name is not a string: " + str(name))
      self._name2id_map[name] = id
      self._id2name_map[id] = name

    def by_name(self, name):
      id = self._name2id_map[name]
      assert self._id2name_map[id] == name
      return id


db = MyDB()
db.add(2,2)
db.add("hej","du")
```

12.2 Testning

Det är svårt att garantera att ens kod är korrekt. Man kan resonera matematiskt om kod och algoritmer, men även då är det lätt att det blir fel. Oavsett hur mycket man planerar, tänker och granskar, så är det bra att också noggrant testa sin kod. Men hur gör man det på bästa sätt?

Erfarenheten visar att det varken är tillräckligt eller praktiskt att testa om ett helt program, själva *produkten*, fungerar som det ska. Om man vill upptäcka problem tidigt och skydda sig mot fel på grund av ändringar i koden, då bör man testa programmets funktioner och metoder.

Ett vanligt arbetssätt är att (1) skriva lite kod, kanske en funktion, (2) verifiera att koden fungerar, och sen fortsätta skriva mer kod. Det kan vara enkelt att verifiera koden: man väljer några parametrar till sin funktion och ser om resultatet blir som det var tänkt. Ett scenario som i praktiken är ännu vanligare är följande:

1. Ändra i befintlig kod, för att lägga till ny funktionalitet.
2. Verifiera den nya funktionaliteten.

3. Verifiera den gamla funktionaliteten.

Punkt 3 blir lätt bortglömd, men den kan också vara lite jobbig att ta hand om. Det är ju på den nya funktionaliteten man har sitt fokus, inte den gamla, och hur var det nu vi testade den koden? Om du ändrar i en funktion `fun`, kommer du då också ihåg att testa andra delar av din kod som är beroende av `fun`? Allteftersom tiden går och ett projekt växer blir det allt svårare verifiera gammal funktionalitet. Och om man jobbar med andras kod blir det förstås ännu svårare att komma på hur man bör testa de funktioner som finns. Så hur gör man?

Det som behövs är *systematisk* och *automatisk* testning, vilket man åstadkommer genom att koda testningen, och metodiken för detta kallas *enhetstestning* (eng. *unit testing*). Idag finns det så pass bra stöd för testning i vanliga programmeringsspråk att det ofta är enkelt att inkludera tester i sitt projekt. I det här avsnittet tittar vi på några olika sätt att arbeta med tester.

Ni kan läsa mer på: https://realpython.com/python-testing/

12.2.1 Testning med `assert`

Kodtestning kan göras både manuellt och automatiskt. För enkel manuell testning kan vi använda **`assert`**.

I labb 2 var uppgiften att skriva funktioner manipulerade eller utvärderade listrepresentationer av polynom. En del av uppgiften var att testa dessa funktioner, vilket gjordes genom att manuellt undersöka outputen av en given input. Ett bättre sätt hade varit om instruktionerna för labb 2 hade innehållit följande fil:

```
from labb2 import *

p = [2,0,1]
q = [-2,1,0,0,1]
p0 = [2,0,1,0]
q0 = [0,0,0]

assert (poly_to_string(p) == '2 + x^2')
assert (poly_to_string(q) == '-2 + x + x^4')
assert (poly_to_string([]) == '0')
assert (poly_to_string([0,0,0]) == '0')
assert (drop_zeroes(p0) == [2, 0, 1])
assert (drop_zeroes(q0) == [])
assert (eq_poly(p,p0))
assert (not eq_poly(q,p0))
assert (eq_poly(q0,[]))
assert (eval_poly(p,0) == 2)
assert (eval_poly(p,1) == 3)
assert (eval_poly(p,2) == 6)
assert (eval_poly(q,2) == 16)
assert (eval_poly(q,-2) == 12)
assert (eq_poly(add_poly(p,q),add_poly(q,p)))
assert (eq_poly(sub_poly(p,p),[]))
assert (eq_poly(sub_poly(p,neg_poly(q)),add_poly(p,q)))
assert (not eq_poly(add_poly(p,p),[]))
```

```
assert (eq_poly(sub_poly(p,q),[4, -1, 1, 0, -1]))
assert (eval_poly(add_poly(p,q),12) == eval_poly(p,12) + eval_poly(q,12))
```

Denna fil kommer endast att köra klart om samtliga tester går igenom! Vi behöver varken ha med testerna eller de polynom vi bara har definierat för tester (dvs `p`, `q`, `p0` och `q0`) i huvudfilen för laborationen `labb2.py`.

På detta sätt kan vi använda **`assert`** för att testa och specificera våra program. Ett problem är dock att vi bara ser resultatet av det första **`assert`** som inte håller och inte för alla andra som inte går igenom. Ett bättre sätt för testning av mer komplexa program är *enhetstester*.

12.2.2 Enhetstester med `unittest`

Modulen `unittest` har funnits med i Pythons standardbibliotek sen version 2.1 och används av många projekt, både kommersiella och open-source. För att använda `unittest` måste vi:

1. Gruppera alla tester i klasser där testerna är metoder.
2. Använda specifika assert-metoder från `unittest.TestCase` istället för de inbyggda **`assert`** uttrycken.

För att skriva om labb 2 exemplet ovan så att det använder `unittest` gör vi följande:

- Importera `unittest`
- Skapa en klass `TestPoly` som ärver från `TestCase` klassen
- Skriv om alla **`assert`** till metoder i den här klassen som använder `self.assertEqual()`
- Ändra så att `unittest.main()` körs när filen laddas

Så här kan filen se ut:

```
import unittest

import labb2

p = [2,0,1]
q = [-2,1,0,0,1]
p0 = [2,0,1,0]
q0 = [0,0,0]

class TestPoly(unittest.TestCase):

    def test_poly_to_string_p(self):
        self.assertEqual(labb2.poly_to_string(p),'2 + x^2', \
                                        "Should be 2 + x^2")

    def test_poly_to_string_q(self):
        self.assertEqual(labb2.poly_to_string(q),'-2 + x + x^4',\
                                        "Should be -2 + x + x^4")

    def test_poly_to_string_empty(self):
```

```
        self.assertEqual(labb2.poly_to_string([]),'0', "Should be 0")

    def test_poly_to_string_empty2(self):
        self.assertEqual(labb2.poly_to_string([0,0,0]),'0', "Should be 0")

    def test_drop_zeroes_p0(self):
        self.assertEqual(labb2.drop_zeroes(p0),[2, 0, 1])

    def test_drop_zeroes_q0(self):
        self.assertEqual(labb2.drop_zeroes(q0),[])

    def test_eq_poly_pp0(self):
        self.assertEqual(labb2.eq_poly(p,p0),True)

    def test_eq_poly_qp0(self):
        self.assertNotEqual(labb2.eq_poly(q,p0),True)

    def test_eq_poly_q0(self):
        self.assertEqual(labb2.eq_poly(q0,[]),True)

    # ...

if __name__ == '__main__':
    unittest.main()
```

Och om vi kör den får vi:

```
.........
----------------------------------------------------------------------
Ran 9 tests in 0.001s

OK
```

Om vi har en bugg, som exempelvis att `poly_to_string` alltid råkar returnera tomma strängen, blir resultatet:

```
.....FFFF
======================================================================
FAIL: test_poly_to_string_empty (__main__.TestPoly)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittestlabb2.py", line 19, in test_poly_to_string_empty
    self.assertEqual(poly_to_string([]),'0', "Should be 0")
AssertionError: '' != '0'
+ 0 : Should be 0

======================================================================
FAIL: test_poly_to_string_empty2 (__main__.TestPoly)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittestlabb2.py", line 22, in test_poly_to_string_empty2
```

```
    self.assertEqual(poly_to_string([0,0,0]),'0', "Should be 0")
AssertionError: '' != '0'
+ 0 : Should be 0

======================================================================
FAIL: test_poly_to_string_p (__main__.TestPoly)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittestlabb2.py", line 13, in test_poly_to_string_p
    self.assertEqual(poly_to_string(p),'2 + x^2', "Should be 2 + x^2")
AssertionError: '' != '2 + x^2'
+ 2 + x^2 : Should be 2 + x^2

======================================================================
FAIL: test_poly_to_string_q (__main__.TestPoly)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "unittestlabb2.py", line 16, in test_poly_to_string_q
    self.assertEqual(poly_to_string(q),'-2 + x + x^4', "Should be -2 + x + x^4")
AssertionError: '' != '-2 + x + x^4'
+ -2 + x + x^4 : Should be -2 + x + x^4

----------------------------------------------------------------------
Ran 9 tests in 0.002s

FAILED (failures=4)
```

Vi ser alltså att 4 av testerna felar och vi kan nu börja debugga vår kod. Genom att skriva bra tester som vi kör varje gång vi gör några ändringar kan vi alltså vara säkra på att vår kod fortsätter fungera som vi förväntar oss trots att vi förändrar den. Detta är väldigt användbart i större projekt där man är många som samarbetar. Många system för att arbeta på kod tillsammans (exempelvis Github) har stöd för denna typ av automatisk testning för varje ändring som någon vill göra till kodbasen (denna utvecklingsteknik kallas för "kontinuerlig integration" (eng. *continuous integration*).

12.2.3 Slumptestning (á la QuickCheck):

Det är svårt att skriva bra tester för hand. Ett alternativ som kan vara väldigt smidigt är att använda ett bibliotek som genererar tester åt en automatiskt, dvs genom så kallad "slumptestning". För denna typ av testning skriver man en specifikation, t.ex. `add_poly(p,q) == add_poly(q,p)`, och biblioteket genererar sedan slumpmässig indata (dvs, polynom i detta fall) för att testa så att specifikationen håller.

Python har stöd för detta genom hypothesis biblioteket.

12.3 Bra kod

Det är svårt att skriva bra kod. Med "bra" kod menas kod som är lätt att läsa, förstå sig på, använda och underhålla. Genom att skriva bra kod kan vi även undvika att introducera buggar.

Det är dock svårt att säga exakt vad som gör kod bra, men förhoppningsvis har ni fått se lite exempel på både bra och mindre bra kod när ni rättat varandras lösningar. Många programmerare har olika definitioner på vad som gör kod bra (på samma sätt som om det är väldigt subjektivt vad som är "bra konst" eller "bra litteratur").

Vi har tidigare pratat om hur man strukturerar sitt program. På kurshemsidan finns också en text med titeln *Tumregler för programmering* (se kurshemsidan) med diverse tips och reflektioner kring vad som gör kod bättre.

Ni kan även hitta en massa specifika Python tips på: https://docs.python-guide.org/writing/style/. Det finns en stilguide, med det stela namnet PEP-8, som är skriven av Guido van Rossum, skaparen av Python. När du börjar programmera i andra språk är det en bra idé att se vilka principer som gäller där.

Det finns verktyg för automatisk granskning av kod. Ett sådant finns på nätet: https://www.pythonchecker.com/. Det finns också IDE:er med liknande stöd, och bibliotek som `pylint` som man kan köra i ett terminalfönster.

Här kommer en lista (i ingen speciell ordning) över saker som vi tycker gör kod bättre:

- Undvik långa rader, mindre än 79 tecken per rad är bra. Om en rad är för lång är det väldigt svårt att läsa den. PEP-8 kräver denna begränsning.
- Undvik långa funktioner, max 1 sida (helst kortare) per funktion. Annars svårt att få överblick och hålla reda på allt funktionen gör.
- En kodsnutt per rad. Följande kod är dålig:

```
print('one'); print('two')

if x == 1: print('one')

if <complex comparison> and <other complex comparison>:
    # do something
```

  Följande kod är mycket bättre:

```
print('one')
print('two')

if x == 1:
    print('one')

cond1 = <complex comparison>
cond2 = <other complex comparison>
if cond1 and cond2:
    # do something
```

- Dela upp kod i funktioner som gör **en** sak och gör det bra. Detta gör det mycket lättare att läsa, testa och förstå koden.
- Tänk först på hur programmet bör struktureras och vilka abstraktioner som bör användas innan ni börjar koda. Gruppera funktionalitet och attribut i klasser—på så sätt uppnås abstraktion och det blir lättare att ändra i programmet senare.

- Dela upp i små enheter, dvs dela upp i lämpliga funktioner och klasser. Detta kallas för "modulär" kod då varje logisk del i koden kan studeras som en egen liten modul som sedan kombineras med andra moduler för att skapa komplexa program.
- Funktioner bör returnera något. Det blir då möjligt att testa dom och det är lättare att förstå vad de gör eller förväntas göra.
- Returnera **inte** värden av typ `int` eller `bool` som strängar, dvs som `'2'` eller `'True'`. Det gör det svårare att testa och använda funktionerna senare. Det är även lätt att introducera buggar på så sätt, exempelvis

```
def f(x):
    return '2'

if f(1) == 2:
    print("This is not printed!")
else:
    print("This is printed!")
```

- Separera **beräkning** från **interaktion**: funktioner som gör beräkningar bör inte ha `print`/`input`. Funktioner som hanterar interaktion bör inte göra komplexa beräkningar.
- Informativa namn på funktioner, klasser, variabler och konstanter gör kod mycket lättare att läsa.
- Variabler/konstanter är bättre än hårdkodade litteraler. Till exempel om vi skriver ett grafiskt program och vill ha ett fönster med storlek 600 x 600, då är det bättre att först skriva

  ```
  width , height  = 600 , 600
  ```

  istället för att upprepa 600 överallt. På så sätt blir det lättare att ändra programmet i framtiden om vi vill ha andra dimensioner på fönstret. Det blir också lättare att läsa koden om det står `width` istället för ett godtyckligt nummer.
- Undvik globala variabler då de ofta leder till spagettikod.
- Dokumentera funktioner och klasser med "docstrings".
- Inkludera tester för att både testa och specificera hur program är tänkta att fungera. Detta är lättare att göra om programmet är uppdelat i mindre funktioner som gör **en** sak och **returnerar** den.
- Enkel kod är lättare att läsa. Undvik därför att skriva onödigt komplicerade kodsnuttar. Till exempel är "`if b == True:`" mer komplicerat än bara "`if b:`". Det är som med vanligt svenska: det är dålig stil att stapla många små ord på varandra. På samma sätt som enkla meningar blir lätta att förstå, så är enkel kod lätt att förstå.

  Här är ett exempel på ett onödigt komplicerat program:

```
def make_complex(args):
    x, y = args
    d = dict()
    d['x'] = x
    d['y'] = y
    return d
```

  Mycket bättre:

```
def make_complex(x,y):
    return {'x': x, 'y': y}
```

Ett bra knep är att alltid läsa igenom sin kod flera gånger och reflektera över hur den kan förenklas.

- Var försiktiga med muterbara datatyper (många buggar uppstår då listor råkas tömmas genom exempelvis `pop()`!). Speciellt farligt är det att ha muterbara typer som globala variabler eller klassattribut!
- Dela upp större projekt i logiska enheter i olika filer/mappar.
- Kommentarer är bra, men ska inte ersätta enkel kod med bra namn på identifierare. Om koden behöver kommenteras för att vara förståelig så betyder det troligtvis att man gör något komplicerat. Bra användning av kommentarer är att förklara varför man använder en specifik datastruktur (t.ex. varför man använder ett `set` istället för `list`, etc.) eller hur det är tänkt att en funktion ska användas.

  Kommentarer som inte bidrar något mer än vad koden redan säger bör undvikas. Till exempel är följande en helt redundant kommentar:

  ```
  rows = []       # Create an empty list
  ```

  Alla som kan lite Python förstår ju att kodraden skapar en tom lista, så varför skriva det i en kommentar?

12.3.1 Tecken på mindre bra kod

- Saknas parametrar och **`return`** i funktioner som ska göra beräkningar? Då gör de troligtvis en massa konstigheter.
- Djup indentering: om man har väldigt nästlade **`if`**-satser så är koden svårläst och bör delas upp i mindre delar.
- "Copy-paste kod": det är lätt att introducera buggar om man kopierar rader och klistrar in dom igen. Programmet blir även svårläst om det blir väldigt långt. Kan man introducera en loop eller funktion istället?
- Är det svårt att skapa bra tester? Då har du nog inte delat upp ditt program i små delar som gör **en** specifik sak. Tänk över designen och introducera bättre funktioner och abstraktioner.

12.4 Uppgifter

1. Jaccard-index (J) är ett mått på hur lika två mängder `A` och `B` är och är definierat som

   $$J := \frac{|A \cap B|}{|A \cup B|}$$

   där $\cap$ och $\cup$ är *snittet* och *unionen* av mängderna `A` och `B` (se Wikipedia). Implementera denna formel som funktionen `jaccard_index(l1,l2)` i Python där `l1` och `l2` är två listor med element.

Tips: Konvertera listorna till mängder och använd deras metoder. Till exempel ska funktionen kunna användas så här

```
print(jaccard_index(['dog', 'cat', 'mouse'], ['dog', 'cat', 'mouse']))
print(jaccard_index(['dog', 'cat'], ['dog', 'cat', 'mouse']))
print(jaccard_index(['dog', 'cat'], ['mouse', 'rat']))
```

```
1.0
0.6666666666666666
0.0
```

2. Använd modulen `unittest` för att skriva enhetstester för din funktion i uppgift 1.

3. En begränsning med Jaccard-index är att det inte tar hänsyn till repeterade element. Till exempel resulterar koden nedan i samma Jaccard-index som ovan, även fast vi kanske skulle beskriva dessa två *multimängder* (dvs samma element kan förekomma flera gånger) som "mer olika".

```
print(jaccard_index(['dog', 'dog', 'dog', 'cat', 'mouse'],
                    ['dog', 'cat', 'cat', 'cat', 'cat',  'mouse']))
print(jaccard_index(['dog', 'cat','cat', 'cat'],
                    ['dog', 'cat', 'mouse', 'mouse']))
```

```
1.0
0.6666666666666666
```

Implementera kod för att beräkna viktat Jaccard-index. Låt A och B vara två *multimängder* och A_z och B_z beteckna hur många gånger elementet z förekom i A respektive B. Då definierar vi viktat Jaccard-index J_w som

$$J_w := \frac{\sum_{z \in (A \cup B)} \min(A_z, B_z)}{\sum_{z \in (A \cup B)} \max(A_z, B_z)}$$

Till exempel:

```
print(weighted_jaccard_index(['dog', 'dog', 'dog', 'cat', 'mouse'],
                             ['dog', 'cat', 'cat', 'cat', 'cat', 'mouse']))
print(weighted_jaccard_index(['dog', 'cat', 'cat', 'cat'],
                             ['dog', 'cat', 'mouse', 'mouse']))
```

```
0.375
0.3333333333333333
```

När du implementerar detta, tänk på tips om struktur och kod från delkapitlet "Bra kod". Till exempel: separera kod i små funktioner som gör en sak, ha inte för långa uttryck på samma rad. Tänk på hur koden bör struktureras.

4. Använd modulen `unittest` för att skriva enhetstester för dina funktioner i uppgift 3.