Linnæus University

Grunderna i Python, del 2

Tobias Andersson Gidlund

Agenda

- Text och strängar
- Listor
- Funktioner

Texter och listor

Text

- Både text och listor har mycket gemensamt
  - En text, dvs en *sträng*, är en lista av tecken
- Det innebär att om man skapar en variabel som är av texttyp (`str`) så kan man komma åt delarna
- En teckensträng i Python kan innehålla all tecken som finns i UTF-8
  - Det går att utöka det, men UTF-8 är standard
- Om man kör i en terminal är det dock viktigt att även teckensnittet där hanterar ett specifikt tecken för att det ska synas

Exempel

```
text = 'ääö är inga problem'
print(text)
text = '€ och ê samt ẽ fungerar också'
print(text)
text = '\u2663 \u2664 \u2665 \u2666' # Unicode-kod för spelkort
print(text)
text = '😀🙃🤯 fungerar också'
print(text)
```

➤ Körning:

```
ääö är inga problem
€ och ê samt ẽ fungerar också
♣ ♤ ♥ ♦
😀🙃🤯 fungerar också
```

Indexering

- En sträng består *element*, där varje element är ett tecken
- Varje element har ett *index* som är dess position i strängen
- Indexeringen börjar från 0, dvs det första elementet i strängen har index 0
- För att plocka ut ett element från strängen så används strängens namn med indexpositionen inom hårda parenteser
- Det är upp till programmeraren att hålla sig inom längden för strängen
  - Det går givetvis att fråga en sträng om dess längd

Exempel

```
strang = 'May the Force be with you!'

print('Första elementet:', strang[0])
length = len(strang) # Fråga om längden

print('Längden:', length)
print('Sista elementet:', strang[length - 1])

print('Element 8 till 12: ', end='')
for i in range(8, 13):
  print(strang[i], end='')
```

- Körning:

```
Första elementet: M
Längden: 26
Sista elementet: !
Element 8 till 12: Force
```

Skivor

- Den sista delen i exemplet tar ut en delmängd av en sträng
- Det kan göras smidigare med hjälp av *skivor* (slices)
- På samma sätt som man kan ange ett index, kan man också ange ett intervall
  - Det görs genom att använda ett kolon mellan start och slut
  - På precis samma sätt som för iteration

Exempel

```
text = 'May the Force be with you!'
print(text[8:13])
print(text[22:])  # Från tecken 22 och fram till slutet
print(text[:3])   # Från början till tecken tre
print(text[::2])  # Varannat tecken
print(text[::-1]) # Baklänges
```

- Körning:

```
Force
you!
May
MyteFreb ihyu
!uoy htiw eb ecroF eht yaM
```

Mer om strängar

- Det finns mycket mer att lära om strängar och mängder med funktioner för dem
- Det är möjligt att kontrollera om ett eller flera tecken finns i en sträng med `in`
- Med `*` är det möjligt att multiplicera strängar och med `+` kan man slå samman dem
- Viktast att veta om strängar är att de är *oföränderliga*, dvs när de väl är skapade kan man inte *ändra* innehållet
- Det är dock möjligt att *skriva över* hela innehållet med en ny sträng

Exempel

```
antal_o = 0
text = 'Wookiees are know to do that'

if 'o' in text:
  print('Jo, det finns "o" i texten "' + text + '"')

for i in text:
  if i == 'o':
    antal_o += 1

print('Antal "o":', antal_o)
```

- Körning:

```
Jo, det finns "o" i texten "Wookiees are know to do that"
Antal "o": 5
```

Funktioner för strängar

- Det finns två viktiga funktioner som fungerar för strängar, `ord()` och `chr()`
- `ord()` returnerar ordningstalet för tecknet, dvs vilket heltal som motsvarar bokstaven
  - Eftersom ASCII är en delmängd av UTF-8 (som Python använder), så är de 128 första samma
  - För andra tecken kommer heltalet som representerar bokstaven i UTF-8 att användas
- `chr()` fungerar på andra hållet, det vill säga att det tar ett tal och returnerar symbolen
  - Skicka alltså in det heltal som du vill ha ett tecken för

Exempel för strängfunktioner

- Notera hur 9827 returneras för ♣ medan \u2663 används för skriva det
  - Det är för att UTF-8-koder är i hexadecimalformat snarare än decimal
  - Det vill säga att de använder basen 16

```
print(ord('A'))
print(ord('Ö'))
print(ord('♣'))
print(ord('🐍'))
```

```
65
214
9827
128013
```

Fler exempel

- Här visas samma tecken men skrivs ut med `chr()`

```
print(chr(65))
print(chr(214))
print(chr(9827))
print(chr(128013))
```

```
A
Ö
♣
🐍
```

- Båda funktionerna är användbara till exempel när man vill jämföra tecken eftersom jämförelseoperatorerna fungerar på det värde de får från `ord()`
  - Det innebär också att man måste vara lite noggrann när man jämför mot till exempel svenska tecken

”Escapesekvenser”

- Strängar kan, som har visats, innehålla nästan alla möjliga tecken
- Det finns dock några besvärliga tecken som man behövers ”escapa”
  - Svårt att översätta, men betyder att man åsidosätter den vanliga utskriften med en speciell utskrift
- Traditionellt sett behöver tecken som `"`, `'` samt osynliga tecken som nyrad eller tab ”escapas”
  - I Python kan man många gånger kringgå det genom att använda olika typer av cittattecken
- Inte minst när man ska läsa eller skriva till fil är det viktigt att ta hand om den här typen av tecken

Skriva ut med hjälp av ”escapesekvenser”

- En sådan här sekvens inleds med teckent `\` (bakvänt snedstreck)
- För att hoppa till en ny rad, använd `\n`

```
print('This is the first line\nAnd this is the second')
```

```
This is the first line
And this is the second
```

- Eftersom alla sådana här sekvenser börjar med `\` så måste även det tecknet specialbehandlas

```
print('You escape with the \\ character')
```

```
You escape with the \ character
```

Några vanliga tecken som specialbehandlas

Sekvens	Beskrivning
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\n`	Newline
`\t`	Horizontal tab
`\r`	Carriage return
`\b`	Backspace
`\v`	Vertical tab
`\uhhhh`	Unicodetecken med hexvärdet `hhhh`

Konvertera till sträng

- Det är ganska vanligt att ett tal behöver omvandlas till en sträng
- För att göra det, används funktionen `str()` som tar en variabel av någon annan datatyp och returnerar den som en sträng
  - Det fungerar även för boolska värden och returnerar då antingen `True` eller `False`

```
everything = str(42)
print(everything)
print(type(everything))
```

```
42
<class 'str'>
```

Strängkonkatenering

- Strängar kan också slås ihop (kallas att *konkatenera* dem) med andra strängar
- Vi har tidigare sett det med hjälp av `,`
- Det är också möjligt att slå ihop strängar med hjälp av `+`
  - Det är många gånger smidigare eftersom plustecknet inte introducerar ett mellanslag mellan strängarna
- Sammanslagningen fungerar enbart för *strängar*, så vill man slå ihop med ett tal så måste det först konverteras med `str()`

```
answer = 42
q_1 = 'life'
q_2 = 'the universe'
q_3 = 'everything'
print(str(answer) + ' is the answer to ' + q_1 + ', ' + q_2 + ' and ' + q_3)
```

Kontrollera och jämföra strängar

- Det är möjligt att kontrollera om en sträng är del av en annan sträng genom att använda `in` (eller `not in`)

```
text = 'May the Force be with You'

if 'Force' in text:
    print('Found it!')
```

```
Found it!
```

- Strängars likhet, det vill säga om de är helt lika, jämförs med `==`

```
text = 'Linux'

if text == 'Linux':
    print('They are equal')
```

```
They are equal
```

Jämföra storhet

- Det är också möjligt att jämföra om två strängar är större (eller mindre) än varandra
- Jämförelsen görs lexikaliskt, baserat på tecken
  - Det vill säga, inte på hur *långa* olika strängar är
- Det numeriska värdet används för varje enskilt tecken
- Precis som i ett uppslagsverk testas de olika tecknen en efter en tills en relation kan bedömmas

```
print('A' < 'B') # B is later in the alphabet and therefore larger
print(str(ord('A')) + ', ' + str(ord('B')))
print('Pink Floyd' > 'Beatles')
```

```
True
65, 66
True
```

Strängar är oföränderliga

- Strängar i Python kan inte ändras efter att de har skapats
  - Det innebär att de är *oföränderliga* (eller *immutable* på engelska)
- På grund av det så är det bara möjligt att *läsa* ett element med `[]`, inte att skriva till den positionen
- Det är fortfarande möjligt att *skriva över* en sträng med nytt innehåll

```
text = 'Pink Floyd'
text = text[0] + text[5] # Read elements 0 and 5 and overwrite text
print(text)
```

```
PF
```

Varför är strängar *oföränderliga*?

- Strängar är oföränderliga i de flesta programmeringsspråk
- Genom att göra det omöjligt att förändra dem, kan de göras betydligt mycket mer effektiva
  - Enklare att skicka till olika delar i ett program eftersom de är ganska enkla
  - Säkrare eftersom det inte går att manipulera dem
- Om en sträng behöver förändras, så är det enklast att bara skriva över dem (som i de exempel som har visats)

Sammanfattning av strängar

- Text är något som används i många, om inte de flesta, program
  - Detta oavsett om det är en terminalapplikation, en webbapp eller ett skrivbordsprogram
- Det är därför viktigt att känna till hur strängar fungerar och vad som kan göras med dem
- Python har ett bra stöd för att göra mycket med strängar, mer finns i boken och på internet

Listor

Listor

- Något som ligger nära hur strängar fungerar är *listor*
  - En sträng är egentligen bara en lista av just tecken
- En generell list kan innehålla vilken datatyp som helst
  - Den kan till och med innehålla flera *olika* datatyper
- Till skillnad från strängar kan man även ändra (och lägga till) listor
- Listor skapas lättast med `[]` och med en uppräkning av elementen
  - Det finns även en klass kallad `list()` som kan användas
- Till skillnad från strängar så är listor *föränderliga*, det vill säga att värdena i listan kan ändras

Exempel

- En tom lista kan även skapas genom att skriva `lst = []`

```
enLista = ['Luke Skywalker', 'Leia Organa', 'Han Solo']

print(enLista)
print(enLista[0])
print(enLista[len(enLista) - 1])
```

- Körning:

```
['Luke Skywalker', 'Leia Organa', 'Han Solo']
Luke Skywalker
Han Solo
```

Fler exempel

- Exemplen visar några av de funktioner som finns för listor

```
annanLista = ['Darth Vader']
annanLista.append('Palpatine')
annanLista.insert(1, 'Jabba the Hutt')

print(annanLista)

annanLista[2] = 'Kejsaren'
print(annanLista)
```

- Körning:

```
['Darth Vader', 'Jabba the Hutt', 'Palpatine']
['Darth Vader', 'Jabba the Hutt', 'Kejsaren']
```

Listor och iteration

- Listor är enkla att använda i iteration
  - I stället för att ange ett intervall, använder man bara listan

```
jedi = ['Yoda', 'Obi-Wan Kenobi', 'Mace Windu']

for j in jedi:
  print(j,'', end='')
```

- Körning:

```
Yoda Obi-Wan Kenobi Mace Windu
```

Funktioner för listor

- Många av de funktioner som finns för tal och strängar, finns även för listor
- Till exempel `len()`, `min()`, `max()` och så vidare

```
integer_list = [83, 27, 67, 21, 34, 99, 76, 45, 27, 51]

print(f'Length of the list: {len(integer_list)}')
print(f'Minimum value in the list: {min(integer_list)}')
print(f'Maximum value in the list: {max(integer_list)}')
print(f'Sum of all values: {sum(integer_list)}')
```

```
Length of the list: 10
Minimum value in the list: 21
Maximum value in the list: 99
Sum of all values: 530
```

Skivor av listor

- Precis som för strängar så är det möjligt att skapa en *skiva* av en lista
- För att återupprepa, skivor är alltså en delmängd av en lista och har följande syntax:
  - `list[start : stopp : steg]`
- Start kan tas bort om man börjar från noll
- Elementet vid positionen för `stopp` kommer inte att tas med
- `steg` kan tas bort om endast ett steg tas varje gång
- Viktigt att komma ihåg är att en *ny* lista returneras, den gamla förändras inte

Exempel på skivor

- Precis som tidigare används -1 för att gå baklänges

```
characters = ['Luke Skywalker', 'Han Solo',
    'Princess Leia', 'C-3PO', 'R2-D2', 'Chewbacca']

print(characters[2:4])
print(characters[::2])
print(characters[3:])
print(characters[::-1])
```

```
['Princess Leia', 'C-3PO']
['Luke Skywalker', 'Princess Leia', 'R2-D2']
['C-3PO', 'R2-D2', 'Chewbacca']
['Chewbacca', 'R2-D2', 'C-3PO', 'Princess Leia', 'Han Solo', 'Luke
Skywalker']
```

Listförståelse

- Ett kraftfullt koncept för listor är *listförståelse* eller vanligare *list comprehension*
- I grund och botten handlar det om att skapa en ny lista från en existerande lista
- Det som är så kraftfullt är att man kan utföra något på varje enskilt element på vägen
- Många gånger kan det också producera kod som är lättare att läsa
  - Och själva utförande är också effektivt

Exempel

- Följande exempel skapar en ny lista genom att ta första bokstaven i listan som först skapas

```
star_wars_movies = ["A New Hope", "The Empire Strikes Back", "Return of the Jedi",
"The Phantom Menace", "Attack of the Clones", "Revenge of the Sith"]

first_letters = [movie[0] for movie in star_wars_movies]
print(first_letters)
```

- Körning:

```
['A', 'T', 'R', 'T', 'A', 'R']
```

Ytterligare exempel

- Beräkningarna kan vara ännu mer komplexa

```
siffror = [1, 2, 3, 4, 5]
nya_siffror = [x * x for x in siffror]

print(nya_siffror)
```

- Körning:

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

- Varje element multipliceras med sig själv

Med selektion

- En selektion kan även läggas till själva skapandet

```
siffror = [1, 2, 3, 4, 5]
nya_siffror = [x * x for x in siffror if x % 2 == 0]

print(nya_siffror)
```

- Körning:

```
[4, 16]
```

- I den nya listan görs endast multiplikationen *om* elementet är ett jämnt tal

Listvariabler pekar på ett minnesområde

- Listvariabler är vad som brukar kallas *referensvariabler*
- Det innebär att variabeln inte håller *värdena* för innehållet i listan, utan bara *var i minnet* som listan finns
  - Det gör det snabbare och effektivare att skicka runt en pekarvariabel eftersom inte innehållet behöver flyttas
- I förlängningen innebär det att två, eller fler, variabler kan ”peka” på samma minnesområde
- Om de gör det, så kan innehållet ändras genom båda variablerna!

Peka och ändra

- I det här exemplet kommer båda variablerna att peka på samma lista

```
lst1 = ['Pink Floyd', 'David Gilmour', 'Roger Waters']
lst2 = lst1
print(f'Content of second list: {lst2}')

lst2.append('Nick Mason')
lst2.append('Richard Wright')

print('Content of both lists:')
print(lst1)
print(lst2)
```

```
Content of second list: ['Pink Floyd', 'David Gilmour', 'Roger Waters']
Content of both lists:
['Pink Floyd', 'David Gilmour', 'Roger Waters', 'Nick Mason', 'Richard
Wright']
['Pink Floyd', 'David Gilmour', 'Roger Waters', 'Nick Mason', 'Richard
Wright']
```

Kopiera listor

- Det här beteendet innebär också att det inte räcker med att göra en tilldelning för att kopiera en lista (till en annan variabel)
  - Till skillnad från hur det fungerar för heltal och strängar
- Om elementen från en lista behöver kopieras till en annan, måste de kopieras *en och en*
- Då behöver man iterera genom den ena listan och lägga till varje enskilt element till den andra
- Det kan göras på många olika sätt

Manuell kopiering

- Här används iteration för att kopiera varje värde från en lista till en annan

```
prg_lang = ['Python', 'Java', 'Dart', 'Rust']
copy_prg_lang = []

for pl in prg_lang:
    copy_prg_lang.append(pl)

copy_prg_lang.append('Go')

print('Both lists:')
print(prg_lang)
print(copy_prg_lang)
```

```
Both lists:
['Python', 'Java', 'Dart', 'Rust']
['Python', 'Java', 'Dart', 'Rust', 'Go']
```

Använda listförståelse

- I det här exemplet används listförståelse för att göra det enklare att kopiera

```
prg_lang = ['Python', 'Java', 'Dart', 'Rust']
copy_prg_lang = [x for x in prg_lang]

copy_prg_lang.append('Go')

print('Both lists:')
print(prg_lang)
print(copy_prg_lang)
```

```
Both lists:
['Python', 'Java', 'Dart', 'Rust']
['Python', 'Java', 'Dart', 'Rust', 'Go']
```

Använda skivor

- Det är också möjligt att använda skivor för att tala om vilka element som ska kopieras

```
prg_lang = ['Python', 'Java', 'Dart', 'Rust']
copy_prg_lang = prg_lang[:]

copy_prg_lang.append('Go')

print('Both lists:')
print(prg_lang)
print(copy_prg_lang)
```

```
Both lists:
['Python', 'Java', 'Dart', 'Rust']
['Python', 'Java', 'Dart', 'Rust', 'Go']
```

Funktionen `copy()`

- Slutligen finns det en funktion som kan användas för att göra samma sak

```
prg_lang = ['Python', 'Java', 'Dart', 'Rust']
copy_prg_lang = prg_lang.copy()

copy_prg_lang.append('Go')

print('Both lists:')
print(prg_lang)
print(copy_prg_lang)
```

```
Both lists:
['Python', 'Java', 'Dart', 'Rust']
['Python', 'Java', 'Dart', 'Rust', 'Go']
```

Listor av listor

- En lista kan innehålla vilken annan datatyp som helst, även andra listor
  - Kallas *flerdimensionella listor*
- En lista kan ha hur många dimensioner som helst, men två (tabell) eller 3 (3D) är vanligast
- Precis som för en nästlad iteration gäller att *varje* element i den yttre listan har en rad med *alla* i den inre
  - Det är också med hjälp av nästlade iterationer man kan komma åt varje element

Exempel

- Notera hur vi måste avläsa längden på den inre listan när vi skriver ut asteriskerna

```
slott = [['*', ' ', ' ', ' ', '*'],
         ['*', ' ', ' ', ' ', '*'],
         ['*', ' ', '*', ' ', '*'],
         ['*', '*', '*', '*', '*'],
         ['*', '*', '*', '*', '*'],
         ['*', '*', '*', '*', '*'],
         ['*', '*', '*', '*', '*']]

for i in range(0, len(slott)):
  for j in range(0, len(slott[i])):
    print(slott[i][j], end='')
  print()
```

- Kanske mer som ett torn än ett slott...

```
*   *
*   *
* * *
*****
*****
*****
*****
```

Sammanfattning

- Med det har vi tittat på det som behöver finnas i ett programmeringsspråk!
- Resten som kommer att tas upp är *nice to have* för att kunna vara produktiv
  - Helt ärligt så skulle det vara mycket svårt att använda ett programmeringsspråk utan vad som kommer
- Det är också viktigt att säga att vi bara har tittat på grunderna *mycket översiktligt*
  - Det finns mycket kvar att lära

Funktioner

Funktioner

- För att kunna skriva större program behövs en möjlighet att dela upp koden
  - Det blir lättare att hitta och att felsöka
- Det blir också enklare att återanvända saker man gör ofta
  - Till exempel att räkna ut medelvärdet av en lista av heltal
- Funktioner *anropas* och kan antingen *utföra* något eller *returnera* ett eller flera värden
  - Den kan ses som en *svart låda* som man kallar på för att få något gjort

Problemlösning

- Tanken är alltså att man delar upp koden i mindre delar, där varje del utför en *specific* uppgift
- Varje sådan uppgift bör vara tydligt avgränsad och väldefinierad – och endast den uppgiften ska göras
- Det blir en del av en problemlösningsstrategi
  - Om ett problem är för stort för att lösa som det är, se vilka delar det består av och lös dem i stället
- Det vill säga: det är sällan en bra idé att försöka lösa hela problemet på en gång, utan bättre att dela upp det
  - Funktioner är ett bra sätt att hantera sådana uppdelningar

Inbyggda funktioner

- Flera inbyggda funktioner har redan använts
  - till exempel `max()`, `min()` och `chr()`
- Det innebär att ni redan vet en del om funktioner eftersom ni har använt dem och egenskapade funktioner fungerar på samma sätt
- Funktionerna som har använts har ofta haft några parametrar, det vill säga, data som skickas till dem
  - Vill man veta det största värdet av 1 och 2 så *kallar* vi på funktionen med `max(1, 2)`
- Sedan hanterar vi resultatet genom att, till exempel, lägga det i en variabel

```
maximum = max(1, 2)
```

Definiera funktioner

- Funktioner placeras (av praxis) högst upp i källkodsfilen
- Varje funktionsdefinition inleds med nyckelordet `def` följt av ett namn och parenteser inom vilka *parametrar* anges
  - En parameter är en variabel som man vill skicka in till funktionen
  - Varje funktion kan ha noll eller flera parametrar
- Alla satser i funktionen indenteras så att man ser tydligt till vilken funktion de tillhör

Inledande exempel

- Följande program gör inget annat än skriver ut `hej` (och är ganska meningslöst)

```
def hej():          # Start av funktionen
  print('Hej')     # Innehållet

# Här börjar programmet
hej()               # Här anropas funktionen
```

- Körning:

```
Hej
```

Parametrar

- En funktion blir mer användbar om man skickar in något till den
- Inom parentes skriver man hur många variabler man vill skicka in
  - Viktigt att tänka på är att det som skickas in *kopieras* till funktionen
  - Det innebär att det inte behöver vara samma namn som där man anropar
- Eftersom Python bestämmer datatyp baserat på vad man använder, så behöver ingen datatyp anges

Exempel med parameter

```
def salutation(meddelande):
  print('Hej', meddelande)

def flera_hej(antal):
  for i in range(antal):
    print('Hej! ', end='')

def flera_hejsan(antal, meddelande):
  for i in range(antal):
    print(meddelande, '', end='')

# Här börjar programmet
namn = 'Anakin Skywalker'
salutation(namn)
flera_hej(3)
print()
flera_hejsan(2, 'Salut!')
```

```
Hej Anakin Skywalker
Hej! Hej! Hej!
Salut! Salut!
```

Returvärde

- Det är sällan en bra idé att låta ett program skriva ut något direkt i en funktion
  - Det låser funktionen till bara konsollapplikationer
- Därför är det oftast bättre att *returnera* ett (eller flera) värden från funktionen
- Det görs genom att avsluta funktionen med nyckelordet `return`
  - Det måste inte ligga sist i en funktion, men funktionen avslutas när man når `return`
- Programmet fortsätter därefter varifrån anropet gjordes

Exempel med returvärde

```
def salutation(meddelande):
  return 'Hej ' + meddelande + '!'

def medelvarde(lista):
  sum = 0
  for i in lista:
    sum = sum + i

  return sum/len(lista)

# Här börjar programmet
namn = 'Darth Vader'
print(salutation(namn)) # Här måste vi göra utskrift
varden = [2, 4, 3, 6, 3, 6, 2]
print(medelvarde(varden))
```

- Körning:

```
Hej Darth Vader!
3.7142857142857144
```

Kopia och referens

- Som tidigare nämndes är det som skickas in en kopia
- Det gäller dock inte om det som skickas är en lista (eller senare, objekt)
- Då är det möjligt att ändra innehållet i listan
  - Det är möjligt då det som skickas till funktionen är en kopia på *var i internminnet* listan finns
- Det är viktigt att komma ihåg eftersom det lätt leder till fel som är svåra att hitta annars

Listor och funktioner

- I övrigt så kan listvariabler användas både som returtyp och som parametrar till en funktion
  - De fungerar precis som alla andra datatyper
- Enda skillnaden är alltså att innehållet i en lista kan ändras
- Listan är dessutom nåbar även utanför själva funktionen och det gör att den kan ändras i flera olika funktioner
- Det är därför relativt vanligt att man i stället för att ändra i en lista i en funktion, skickar tillbaka en *ny* lista från funktionen

Exempel

```
def byta(a, b):
  tmp = a              # Kan göras smidigare i Python
  a = b
  b = tmp

def byta_lista(lista):
  tmp = lista[0]
  lista[0] = lista[1]
  lista[1] = tmp

# Här börjar programmet
a = 1
b = 2
tal = [1, 2]
byta(a, b)
byta_lista(tal)
print('a: ' + str(a) + ', b:' + str(b)) # tal -> sträng
print(tal)
```

```
a: 1, b:2
[2, 1]
```

Exempel på *sökning* i en lista

- Det finns inbyggd funktionalitet för att söka i en lista, men här visas en funktion som gör just det

```
def is_in(lst, value):
    for e in lst:
        if e == value:
            return True

    return False

def main():
    planets = ['Tatooine', 'Hoth', 'Endor']
    print(f'Hoth in list? {is_in(planets, "Hoth")}')
    print(f'Dantooine in list? {is_in(planets, "Dantooine")}')

main()
```

```
Hoth in list? True
Dantooine in list? False
```

Flera returvärden

- En sak som gör Python relativt unik är möljligheten att returnera mer än ett värde
- Vad som sker är egentligen att en tupel skickas tillbaka men man kan sedan ta emot den med flera variabler

```
def forst_sist(namn):
  return namn[0], namn[len(namn) - 1]

# Här börjar programmet
f, s = forst_sist('Chewbacca')
print(f, s)
```

- Körning:

```
C a
```

Lokala variabler

- En funktion har ett kodblock (sin sekvens) och i den kan nya variabler deklareras
- De nya variablerna är *endast* tillgängliga i funktionen
  - De kan inte läsas eller skrivas till utanför funktionen
  - Om det finns variabler utanför funktionen med samma namn så är det *andra* variabler
- Det här ska ses som något bra eftersom det annars är lätt att funktioner får *sidoeffekter*
  - Med det menas att ändringar kan göras på variabler som man ”egentligen” inte ska ha tillgång till

Exempel på lokala variabler

```
def knights_who_say_ni():
    # Local variable 'ni' is only accessible within this function
    ni = 'Ni!'
    message = 'We are the knights who say ' + ni
    return message

def the_forest_of_death():
    # Local variable 'shrubs' is only accessible within this function
    shrubs = 'We demand a shrubbery!'
    return 'To pass through the forest, you must bring us a shrubbery.'

print(knights_who_say_ni())
print(the_forest_of_death())

# Uncommenting the next two lines will raise an error
# print(ni)
# print(shrubs)
```

```
We are the knights who say Ni!
To pass through the forest, you must bring us a shrubbery.
```

En `main()` funktion

- I de exempel som har visats så har bara en kommentar används för att markera var själva programmet startar
- Python, precis som många andra språk, har möjligheten att ha en funktion som *startar* själva exekveringen
- Den kallas ofta `main()`
- I Python behöver man dock anropa den direkt för att den ska köras
- Den ”formella” deklarationen av en `main()`-funktion i Python ser ut så här:

```
if __name__ == '__main__':
    main()
```

En tillräcklig `main()`

- För de allra flesta tillfällen så räcker följande utmärkt:

```
def hello():
    print('Hello')

def main():
    hello()

main()  # Start of program
```

```
Hello
```

- Det kan dock vara en bra idé att faktiskt använda någon form av `main()`-funktion för att tydliggöra var programmet startar

Mycket mer om funktioner

- Det finns mycket mer om funktioner
  - Parameterar kan ha standardvärden (om inget skickas)
  - Funktioner kan anropa sig själva (rekursion)
- Det vi har tagit upp här är dock en bra början!
- Funktioner är en viktig del i att dela upp ett problem i mindre delar