Listor

Listor är ett av de viktigaste verktygen i Python med ett stort användningsområde.

Det innefattar bland annat det som i många andra språk kallas för *arrayer* men är mer generellt än dessa brukar vara.

Endimensionella listor

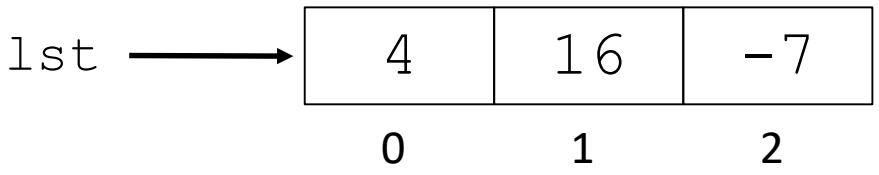
En lista består ***element*** och varje element har en plats som anges av ett ***index***. Det första elementet har alltid index 0. Elementen kan vara av olika typ, även blandade. Exempel kod, där alla elementen är av typen `int`.

```
lst2 = []          # Variabeln lst2 tilldelas en tom lista
lst = [4,16,-7]    # lst tilldelas en lista med tre element
print(lst)         # Skriv ut variabeln lst
print(type(lst)) # Skriv ut typen för variabeln lst
```

Utskrift:

```
[4, 16, -7]
<class 'list'>
```

Kan illusteras på följande sätt:



Variabeln `lst` har tre element, `lst[0], lst[1], lst[2],` alla är av typen `int`

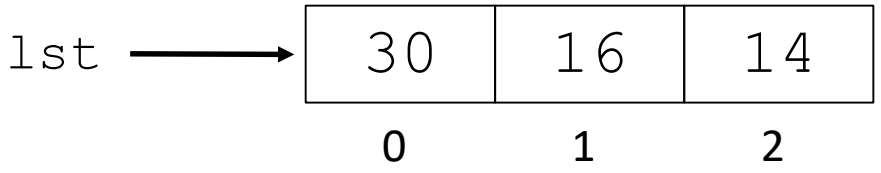
Hantering av enskilda element i en lista:

```
lst = [4,16,-7]
print(lst[1])          # skriver ut element nr 2, har index=1
lst[2]=14              # uppdatera element nr 3.
print(lst)             # Skriv ut lst
lst[0]=lst[1]+lst[2] # uppdatera första elementet
print(lst)             # Skriv ut lst
```

Utskrift:

```
16
[4, 16, 14]
[30, 16, 14]
```

Så här ser `lst` ut nu:



Legala index i en lista:

```
lst = [4,16,-7]   # Legala index är 0…2, ty lst har 3 element
lst[3]=99         # Men …
```

Ger felmeddelande:

```
    lst[3] = 99
IndexError: list assignment index out of range
```

Listans längd, dvs antal element, ges av pythons inbyggda funktion **`len`**:

```
print(len(lst))   # Ger utskrift: 3
```

Ändra sista elementet, utan att "veta" listans längd:

```
lst[len(lst)-1]=8   # Alt 1
lst[-1]=8           # Alt 2, lite enklare
```

`len`, inbyggd funktion i Python, returnerar listans längd

Ändra näst sista elementet, utan att "veta" listans längd:

```
lst[len(lst)-2]=8   # Alt 1
lst[-2]=8           # Alt 2, lite enklare
```

Operatorerna * och + för att skapa nya listor:

```
lst = [0]*5  #Läses: gör en lista med 5 st noll-element

# Ger [0,0,0,0,0]

x = [1,2,3]
a =7
lst = [a]*5 + x + [3*a]*4    # Lägg ihop tre listor

# Ger [7, 7, 7, 7, 7, 1, 2, 3, 21, 21, 21, 21]


lst = 3*x  # Ger [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Exempel: Operatorn * som vanligt för multiplikation

```
k = 7
lst = [k, 2*k, 3*k, 4*k] # Ger [7, 14, 21, 28]
```

Skivningar, mkt fiffigt

Allmänt: `[forst:sist:steg]` kallas *skivning*

```
lst = [7, 14, 21, 28]
lst2 = lst[1:]

lst2 = lst[1:3]

lst3 = lst2 + lst[0:1]

lst = [1, 2, 3, 4]
lst2 = lst[3:0:-1]

# Skiva i vänsterledet
lst[0:2] = [14, 99]

y = [1, 2, 3, 4]
z = y[-1:0:-1]
```

```
# Ger [14, 21, 28]

# Ger [14, 21]

# Ger [14, 21, 7]

# Ger [4, 3, 2]

# lst blir: [14, 99, 3, 4]

# Ger [4, 3, 2]
```

Listor och strängar är båda exempel på sekvenser och har vissa likheter när det hanteringen.

Först mkt lite om strängar här, mer om strängar i Fö5…

```
s = 'Torsten'  # Tilldela ett strängvärde till en variabel
type(s)        # ger: <class 'str'>
len(s)         # är 7     Funktionen len
s[2]           # är 'r'.  Indexeringen...
s2 = s[-1::-1] # s2 är 'netsroT'. Skivning
s3 = s*2       # s3 är 'TorstenTorsten'
```

Listor med strängar som element

```
lst = ['abc', 'Zyxw']
print(lst)                    # ['abc', 'Zyx']
```

Kommer åt de enskilda teckan
Elementet `lst[0]` är `'abc'`
Elementet `lst[1]` är `'Zyxw'`

Kan komma åt de enskilda tecken som strängen vore en lista.

```
s = lst[0][0]    # är 'a'   Notera dubbelindex [0][0]
s = lst[0][1]    # är 'b'
len(lst[1])      # är 4
```

Pythons inbyggda metoder för listor.

Metoder anropas med *punktnotation*, dvs listans namn, följt av punkt, därefter metodens namn.

Några exempel:

```
v1 = [9, 14, 6]
v1.append(23)       # v1 blir [9, 14, 6, 23]
v1.insert(1, 17)    # v1 blir [9, 17, 14, 6, 23]
x = v1.pop()        # ger x=23 och v1=[9, 17, 14, 6]
```

Notera vänsterledet kan undvaras:

```
v1.pop()

v1.pop(-2)          # v1 blir [9, 17, 6]
v1.extend([15,3]) # v1 blir [9, 17, 6, 15, 3]
v1.pop(2)           # v2 blir[9, 17, 15, 3]
v2 = v1.copy()      # v2 blir [9, 17, 15, 3]
v2.sort()           # v2 blir [3, 9, 15, 17]
```

forts metoder…

```
names = ['Kim', 'Ed', 'Torsten', 'Agda']
names.sort()                 # ['Agda', 'Ed', 'Kim', 'Torsten']
names.sort(reverse=True) # ['Torsten', 'Kim', 'Ed', 'Agda']
```

Sortering baserad på fknvärde, längden på elementen, i detta fall strängens längd

```
names.sort(key=len)          # ['Ed', 'Kim', 'Agda', 'Torsten']
```

Baserad på egendefinierad funktion, ”värdet” för 2:a tecknet för elementen

```
def f(e):
    return e[1]
```

Vad blir resultatet om istället:
`return e[-1]`

```
names = ['Kim', 'Ed', 'Torsten', 'Agda']
names.sort(key=f)      # ['Ed', 'Agda', 'Kim', 'Torsten']
```

Notera: *Metoder* anropas med punktnotation, ex. `v1.pop()`

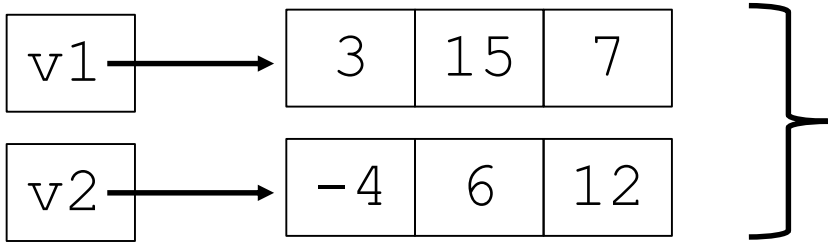
Pythons inbyggda funktioner för exvis listor, några exempel:

```
v3 = list('kim')  # ['k', 'i', 'm'] Gör en lista av strängen
v1 = [9,14,6]
x = max(v1) # 14
x = min(v1) # 6
x = sum(v1) # 29
v2 = sorted(v1) # v2 blir [6,9,14]
```

Notera: *Funktioner* anropas utan punktnotation

Listor och referenser

```
v1 = [3, 15, 7]
v2 = [-4, 6, 12]
```



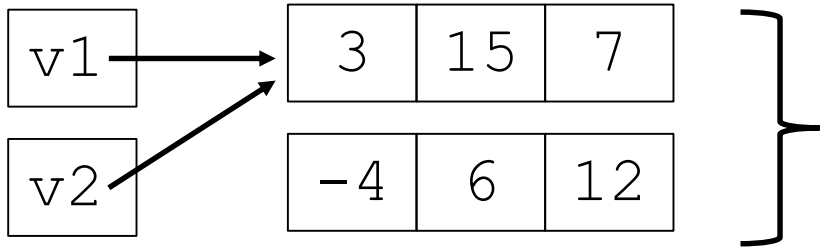
`v1` och `v2` refererar till varsin lista

I python finns den inbyggda funktionen **`id`** som returnerar en variabels s.k. identitet. Vi kan för enkelhets skull säga att det är den adress i datorns minne som variabeln refererar till. **`id`**-värdet för variablerna är:

```
id(v1)      #  51367336
id(v2)      #  51452392
```

Om vi nu gör följande tilldelning:

```
v2 = v1
```

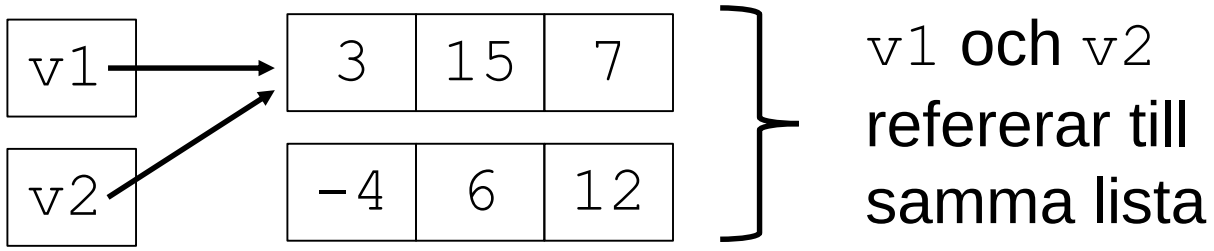


`v1` och `v2` refererar till samma lista

```
id(v2)      # 51367336
```

Dvs samma id som `v1`

Forts…

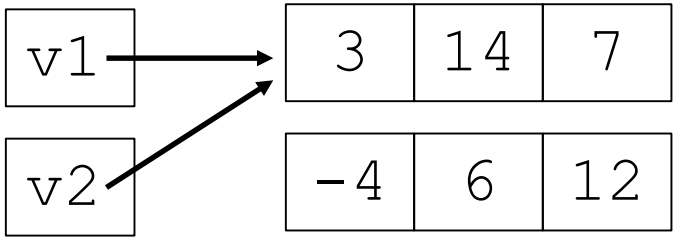


`v1` och `v2` refererar till samma lista

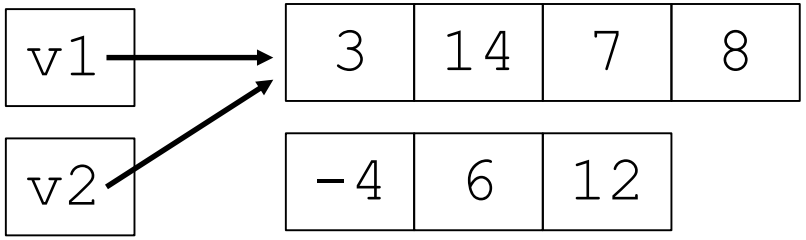
Kommer man åt listan `[-4,6,12]` ?

Nej. Vi har tappat bort listan som `v2` tidigare refererade till

`v1[1]=14`



`v2.append(8)`



Loopa (iterera) igenom en lista:

```
lst = [4,16,-7]
for el in lst:  # För varje element (el) i lst
    print(el)   # Skriv ut el
```

Utskrift:

```
4
16
-7
```

Alt 2 (mha inbyggda funktionen enumerate, får två värden, räknare och element)

```
for i, el in enumerate(lst): # För varje i och element
    print(i,el)   # Skriv ut i och element
```

Utskrift:

```
0 4
1 16
2 -7
```

Funktionen enumerate returnerar två värden för varje iteration

Alt 3:

```
for i, el in enumerate(lst,100): # Starta med i=100
    print(i,el)
```

Utskrift

```
100 4
101 16
102 -7
```

Alt 4:

```
for tpl in enumerate(lst): # För varje tuple (index,element)
    print(tpl)   # Skriv ut tuple (parvärdet)
```

Utskrift:

```
(0, 4)
(1, 16)
(2, -7)
```

Alternativ print-sats:

```
print(tpl[0], tpl[1])
```

→

Utskrift:

```
0 4
1 16
2 -7
```

Mer om tuplar i annat dokument

List comprehension (listbyggare). Mkt, mkt fiffigt

Skapa en lista `v2` genom att utgå från en befintlig lista `v1` och utföra ett visst uttryck på alla element i listan `v1`.

```
v1 = [1,2,3,4]
v2 = [x*x for x in v1]
# v2 blir [1,4,9,16]
```

För varje element `x` i listan `v1`: Beräkna `x*x` som bygger upp en lista. Högerledet, dvs den färdiga listan tilldelas till variabeln `v2`

```
v2 = [x*x for x in v1 if x>2]
# v2 blir [9,16]
```

```
v2 = [x+4 for x in v1 if x%2==0]
# v2 blir [?]
```

Tänk igenom steg för steg…

Svar: `[6,8]`

```
v2 = [i*x for i,x in enumerate(v1)]
# v2 blir [0, 2, 6, 12]
```

För varje index (i) och element (x): Beräkna i*x

Listbyggare forts, utöka lista, har vi sett tidigare… hoppa över…

```
a = [2, 3, 4]
x = 3
b = [0]*x + b[1:3]+ [0]*2
# b blir [0, 0, 0, 3, 4, 0, 0]
```

Inläsning till lista

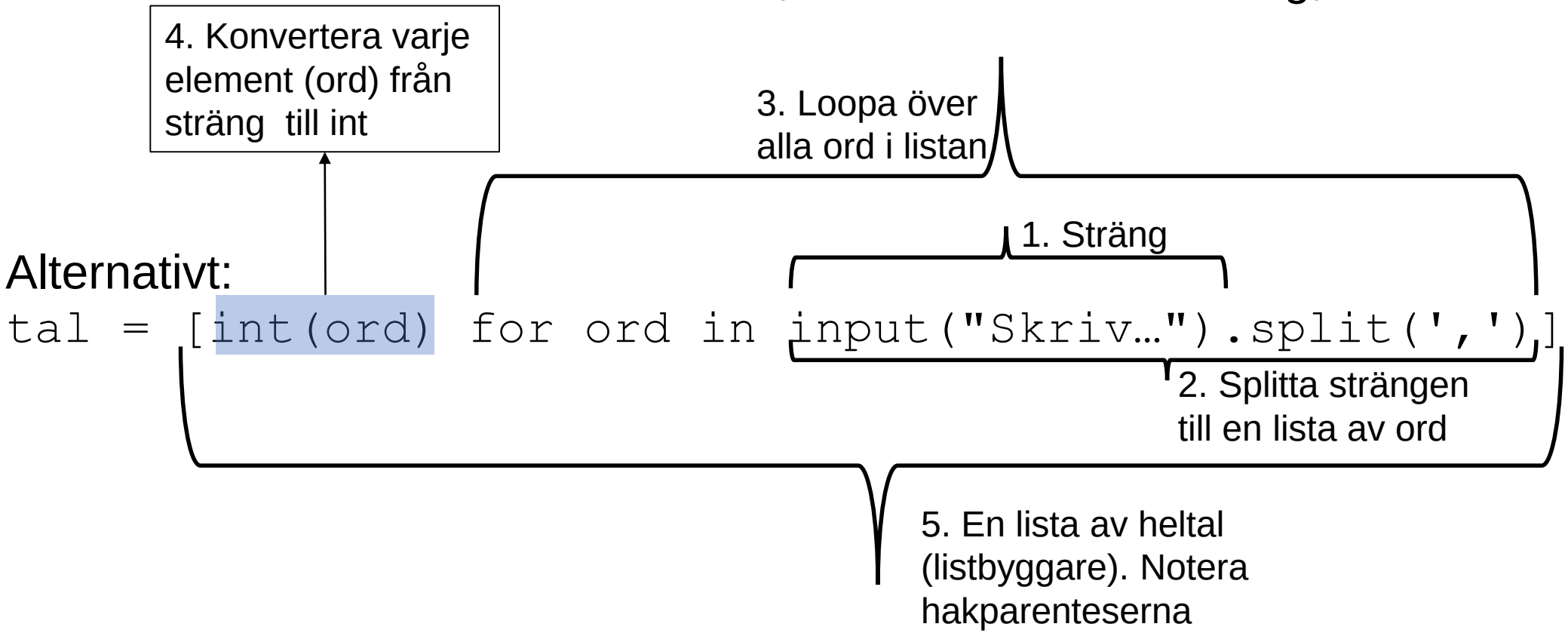
Ett program som fungerar på följande sätt:

```
Skriv ett antal heltal, kommatecken mellan: 2, 1, 4, 3
Du skrev 4 tal
Max av talen: 4
Summan av talen: 10
```

Exempel på kod:

```
s = input('Skriv ett antal heltal, kommatecken mellan:')
# s blir alltid en sträng, exvis '1, 2, 3, 4'
s2 = s.split(',') # Inbyggd metod
# Splitta s till en lista s2, exvis ['1', '2', '3', '4']
# Gör om alla ord (strängar) till heltal och lägg i en lista
tal = [int(ord) for ord in s2] # listbyggare
print('Du skrev', len(tal), 'tal')  # Använder de inbyggda
print('Max av talen:', max(tal))     # funktionerna len,
print('Summan av talen:', sum(tal))  # max och sum
```

Det hela kan skrivas som EN sats, där det utförs i fem steg, 1-5.




```
print("Du skrev", len(tal), "tal")
print("Max av talen", max(tal))
print("Summan av talen", sum(tal))
```

Skriva egna funktioner för listor

Tre olika scenarior:

```
lst1 = fA()       # fA returnerar en lista

lst2 = fB(lst1)   # fB får en lista som argument, returnerar en lista

fC(lst1)          # fC får en lista som argument, returnerar inget.
```

Funktionerna finns i programmet `listFunctions.py`
Vi kikar på koden och diskuterar de olika lösningarna.

I koden används den inbyggda funktionen **`id`** som returnerar en variabels s.k. identitet.
Vi kan för enkelhets skull säga att det är den adress i datorns minne som variabeln refererar till.

Tag upp ex: f(xlist), ylist = f(xlist), xlist = f(xlist), append ändrar ej referensen, ej heller pop. Öppna två instanser av Thonny, ett med fkndef, det andra med anropen.

Listor kan ha element av godtyckliga typer

```
a = [1, 6, 9]                    # heltal
b = [3.24, -9.0, 1.34e10]        # flyttal
c = ['abba', 'aBc', '14']        # strängar
d = [ [1,2,3], [99,14] ]         # listor
```

Lista med paddor

```
# Skapa två paddor, lägg dem i en lista
family = [turtle.Turtle(), turtle.Turtle()]
# Hantera alla paddor i listan
for t in family:
    t.left(random.randint(-45,45))
    t.forward(random.randint(1,100))
```

Blandade typer i en lista:

```
e = [ 2, 'abba', [-4, 17], turtle.Turtle() ]
```

Exempel

Antag att vi har två listor med strängar

```
swedish= ['hund', 'hej', 'katt']
english= ['dog', 'hello', 'cat']
```

Vi vill skapa listan `longest` som väljer de längsta orden i varje par från de två listorna, dvs

```
longest = ['hund','hello','katt']
```

Vi har följande funktion till hjälp:

```
def longestString(str1, str2):
    """Returnerar den längsta strängen av två."""
    if len(str1) > len(str2):
       return str1
    else:
       return str2
```

Lösning 1, enkel loop (fortsättning från föregående sida)

```
longest = []
for i in range(len(swedish)): # För varje par i listorna
    longest.append(longestString(swedish[i], english[i]))
```

- Behöver ett index (`i`) när vi faktiskt bara vill iterera över hela listan
- Lite problem att vi har TVÅ listor vi vill iterera över samtidigt

Lösning 2, med zip (inbyggd python funktion)

```
longest = []
for (s1, s2) in zip(swedish, english):
    longest.append(longestString(s1, s2))
```

- `zip` skapar en iteration över par (tuplar) från två itererbara objekt (tänk tänderna i ett blixtlås som möts)

Lösning 3, listbyggare (fortsättning från föregående sida)

```
longest = [longestString(s1,s2) for (s1,s2) in \
                                    zip(swedish,english)]
```

Lösning 4

```
longest = list(map(longestString, swedish, english))
```

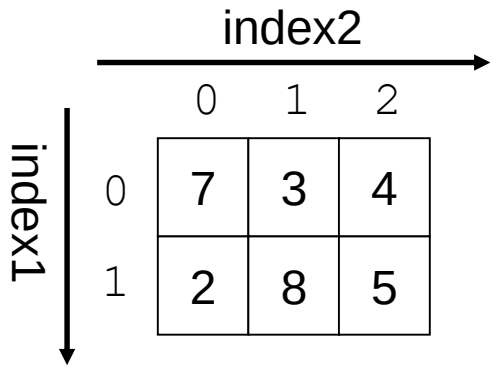
- *`map` är inbyggd funktion i python som kan ta flera itererbara objekt samtidigt (`swedish` och `english`) och var och en av dem bidrar då med en parameter till funktionen (`longestString`)*

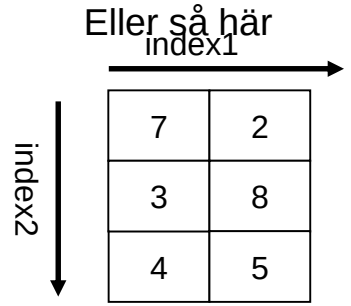
- Att använda färdiga verktyg från Python gör koden
  - Kortare
  - Mer entydig
  - Lättare att förstå (när man känner igen mönstren)

”Tvådimensionella” listor, har två index

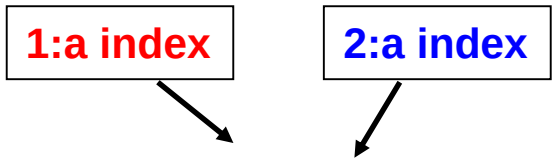
```
t = [ [7,3,4], [2,8,5] ] #Notera: Yttre och inre par av hakparenteser
print(t)  # ger [[7, 3, 4], [2, 8, 5]]
```

Kan ses som en tabell (matris):

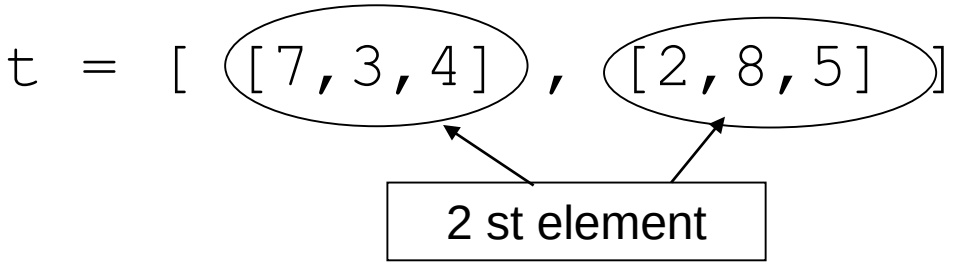




Elementen är int:

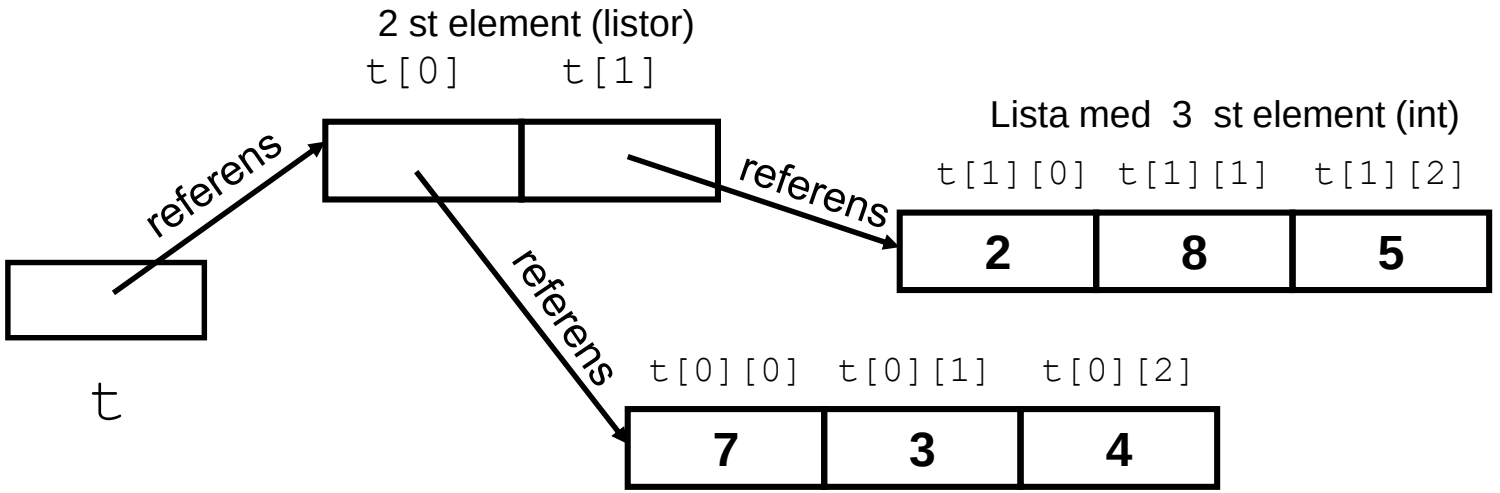



```
t[0][0]=7, t[0][1]=3, t[0][2]=4,
t[1][0]=2, t[1][1]=8, t[1][2]=5.
```



Listan `t` har två ”element” `t[0]` resp. `t[1]` , där varje sådant element är en lista med ett index som i sin tur refererar till tre element som är heltal.

Så här kan det illusteras:



Skriv ut listan `t:`

```
print(t)        # [[7, 3, 4], [2, 8, 5]]
print(t[0])     # [7, 3, 4]
print(t[1])     # [2, 8, 5]
print(t[0][1]) # 3
```

Dubbelloop, loopa igenom alla element:

```
for i1 in range(0,2): # radloop
    for i2 in range(0,3): # kolumnloop
        print(t[i1][i2], end =' ')
    print() # Radbyte
```

Ger:

```
7 3 4
2 8 5
```

Alt dubbelloop:

```
for x in t: # rad
    for y in x: # kolumn
        print(y, end = ' ')
    print()
```

```
t = [ [7,3,4], [2,8,5] ]
```

Summera elementen

```
print(sum(t))  # Ger avbrott, kan ej summera 2D-listor
print(sum(t[0])# Ger 14, som är summan av första raden

s = 0
for e1 in t:
    for e2 in e1:
        s = s + e2
```

s blir 29

```
s = 0
for x in t:
    s = s + sum(x) # summera summan av varje rad

s = sum( [sum(x) for x in t] )
```

Bygg upp 1D-listan: `[14,15]`.
Därefter summan av dessa

Sudoku
En variabel som representerar en sudoku-spelplan (9x9 rutor):

Alla element sätts till noll, tomma rutor är noll.

```
sud = [[0]*9, [0]*9, … ]
```

Alt:

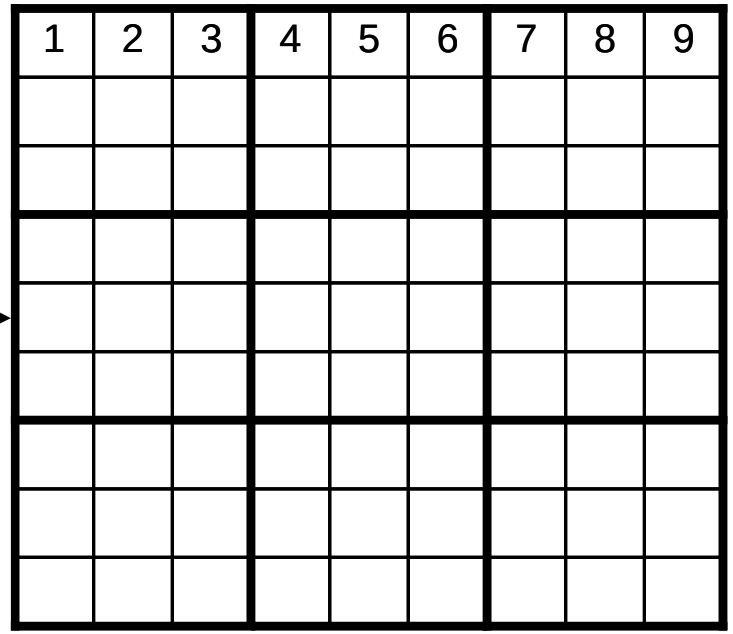
```
sud = [[0 for x in range(9)] for y in range(9)]
```

Sätt värdena 1-9 i översta raden:

```
for x in range(9):
    sud[0][x] = x+1
```

Funktion som skriver ut

```
def printSud(sud):
   for r in range(9):
        print(sud[r])
```

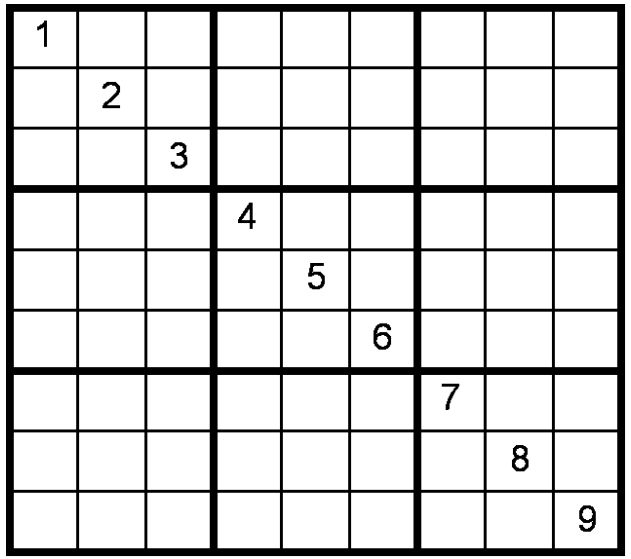


Låtsas att det är nollor i alla tomma rutor i figuren till vänster.

Alt modifiera `printSud` så den skriver ut blanktecken istf nolla.

Fundera på:

Fyll diagonalen med värdena 1-9.



Låtsas att det är nollor i alla tomma rutor i figuren till vänster

```
for x in range(9):
    sud[x][x] = x+1
```

Fundera på:
Lägg in värden 1-9 i tre av delar av spelplanen enligt nedan.

1	2	3						
4	5	6						
7	8	9						
			1	2	3			
			4	5	6			
			7	8	9			
						1	2	3
						4	5	6
						7	8	9

Att fylla **en** kvadrat är **ett** delproblem. Att fylla de tre kvadraterna är egentligen att göra samma sak (lösa samma problem) tre gånger, vilket kan uttryckas med en loop som snurrar tre varv, där varje varv består av en kvadratfyllning.

```
# Initiera alla element till noll
sud = [[0 for x in range(9)] for y in range(9)]
#  Fyll kvadranter
fillQuad(0,0,sud)
fillQuad(3,3,sud)
fillQuad(6,6,sud)
```

Skriv funktionen `fillQuad`

Lös ett sudoku

a) Skapa en sudoku-spelplan som fylls med slumpmässiga siffror 1-9 i alla rutor. Gör ingen kontroll av siffrorna.

b) Bygg på programmet: Räkna hur många rader som är godkända dvs innehåller alla siffror 1-9. (tips på räknare, se nästa sida)

c) Bygg på programmet: Räkna hur många kolumner som är godkända, dvs innehåller alla siffror 1-9.

d) Bygg på programmet: Räkna hur många av de 9 st 3x3 rutorna (kvadraterna) som är godkända.

e) Med räknarna i b)-d) kan du kontrollera ifall hela sudoku't är korrekt.

Gissningsvis är det ytterst sannolikt att det blir någon rad, kolumn eller kvadrat som blir godkänd.

Testa därför programmet genom att listan istället har fixa värden för alla element, istf slumpvalda, så att någon rad, kolumn eller kvadrat blir godkänd.

Tips på räknare för sudoku

Alt 1:

```
count = [0]*9 # Alla nio element får värdet 0.
```

`count` kan användas för att räkna antal element med värdet 1-9, om `count[0]` är antalet 1:or, `count[1]` är antalet 2:or, etc

Exempel:

```
dig=4;
count[dig-1] = count[dig-1] + 1;
```

Ovan sats ökar räknaren med ett för värdet 4

Alt 2 (fiffigare för sudoku):

```
exists = [False]*9 # Alla element är false.
```

Variabeln `exists` kan användas för att bokföra huruvida ett siffra (1-9) redan är valt. Om `exist[0]` är `true` betyder det att siffran 1 redan är vald.

Exempel:

```
int dig=4;
if (exists[dig-1]==true)…
```

så är siffran 4 redan vald

Annars så väljer vi siffran och gör:

```
exists[dig-1]=true
```

Dvs sätter att siffran 4 nu är vald.

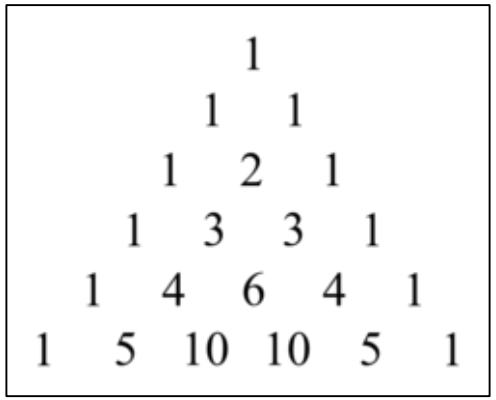
*-uppgifter

4. Skriv ett program som fyller ett från början tomt sudoku med nummer efter nummer på följande sätt:
   Börja med att slumpa en siffra (dig1) och placera dig1 i övrevänstra positionen, dvs i första raden, första kolumnen. Därefter slumpa nästa siffra (dig2), som ej får vara dig1, och placera i första raden, andra kolumnen. Fortsätt enligt samma princip så att den översta raden är fylld.
   Fortsätt därefter med nästa rad, men förutom att testa att unika nummer fås i den raden, måste även numret vara unikt med avseende på kolumnen och aktuell kvadrat.
   Fortsätt med nästa rad…

   Notera: Detta program kommer att ta lång tid, kommer kanske aldrig att sluta, men gör lämpliga utskrifter så att du ser hur programmet löpande fyller sudoku't.

Ex. Pascals triangel, `(a+b)ⁿ` binomialsatsen

Koefficienterna i Pascals triangel (de sex första raderna)



1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	
1	5	10	10	5	1

```
p = [ [1], [1,1], [1,2,1], [1,3,3,1], [1,4,6,4,1], [1,5,10,10,5,1] ]
```

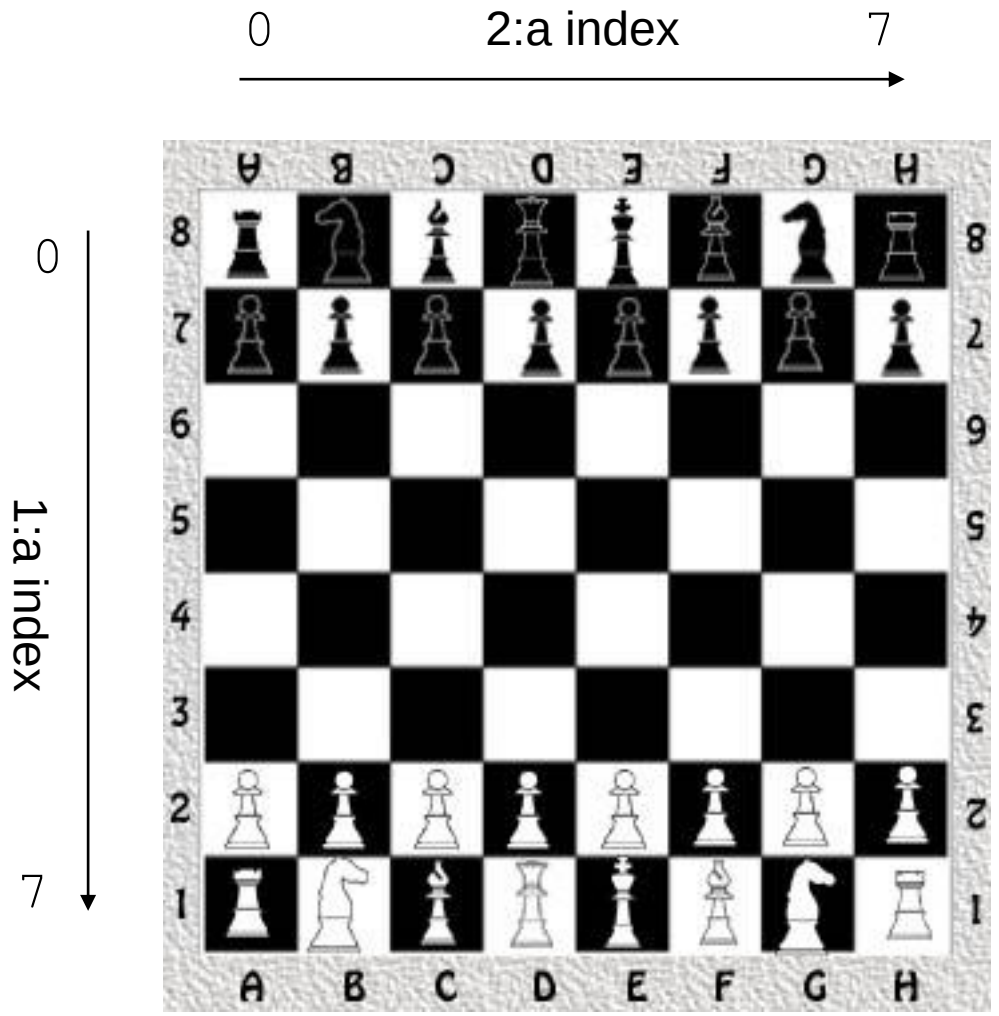
`len(p)` har värdet 6
`len(p[0])` har värdet 1
Utskrift av `p`:

```
for el in p:
    print(el)
```

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
```

Skapa nästa rad i `p`, dvs elementet `[1, 6, 15, 20, 15, 6, 1]`
Skapa näst, nästa rad, etc

Ex. schackspel



Schack fortsätter→

Listor: (Torsten Andersson, IT-inst, Uppsala Universitet)

Ex, hur ett schackbräde kan representeras

Skapar en tvådimensionell lista, där varje element är en sträng bestående av ett tecken, dvs 8 ggr 8 element, 8 rader, 8 kolumner. Varje tecken är första bokstaven i pjäsens namn:
Torn, Springare, Löpare, Dam, Kung, Bonde
Versaler är svarta pjäser. Gemener är vita pjäser.
Låt tecknet `'_'` (understrykningstecknet) betyda tom plats.

```
board =  [
           ['T','S','L','D','K','L','S','T'],
           ['B','B','B','B','B','B','B','B'],
           ['_','_','_','_','_','_','_','_'],
           ['_','_','_','_','_','_','_','_'],
           ['_','_','_','_','_','_','_','_'],
           ['_','_','_','_','_','_','_','_'],
           ['b','b','b','b','b','b','b','b'],
           ['t','s','l','d','k','l','s','t']
         ]
```

Schack fortsätter→

Vi skriver ut startbrädet

```
for s in board:
    print(s)
}
```

Ger följande resultat:

```
['T', 'S', 'L', 'D', 'K', 'L', 'S', 'T']
['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B']
['_', '_', '_', '_', '_', '_', '_', '_']
['_', '_', '_', '_', '_', '_', '_', '_']
['_', '_', '_', '_', '_', '_', '_', '_']
['_', '_', '_', '_', '_', '_', '_', '_']
['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b']
['t', 's', 'l', 'd', 'k', 'l', 's', 't']
```

Versaler = svarta pjäser
Gemener = vita pjäser

Schack fortsätter→

Listor med trippla index

Exempel på tredimensionella data:

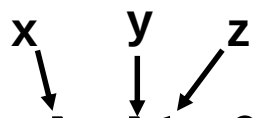
- Bilder (medicinska bilder med *voxlar,* jmfr *pixlar*)
- Spel (ex. 3D luffarschack, 3D sudoku)
- 2D-data med flera lager

  t.ex. tidslager med sudoku, schack

Lista med 3 dimensioner

En variabel med tre index kan betraktas som en variabel som representerar data med tre dimensioner, t.ex. en volym med x,y och z.

Exempel: En variabel som representerar ett rätblock med bredden (x=0-1), höjden (y=0-1) och djupet (z=0-2), dvs 2x2x3 diskreta element (värdena är 1-12).




```
cube = [ [ [1,2,3], [4,5,6] ], [ [7,8,9], [10,11,12] ] ]
```

x=0
y=0
z=0: 1
z=1: 2
z=2: 3

x=0
y=1
z=0: 4
z=1: 5
z=2: 6

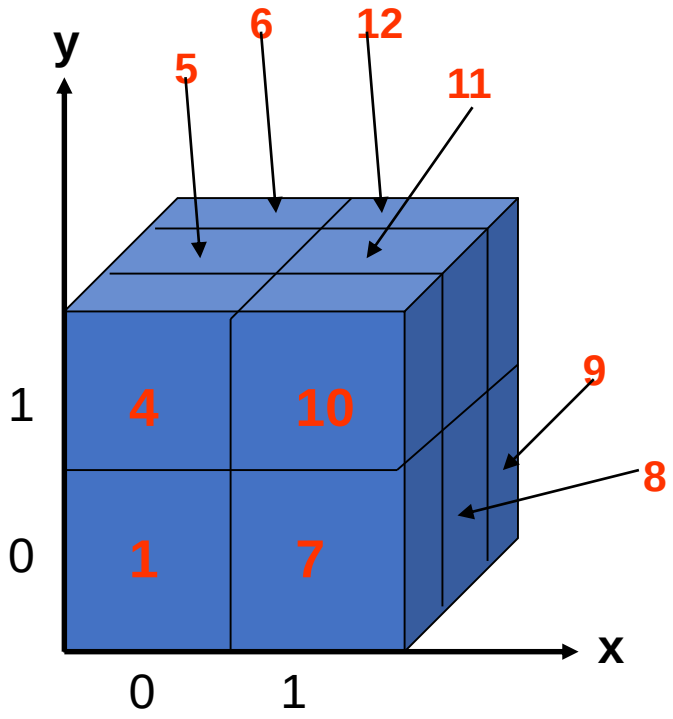
x=1
y=0
z=0: 7
z=1: 8
z=2: 9

x=1
y=1
z=0: 10
z=1: 11
z=2: 12

Vad är `cube[0]` och `cube[1]`?

Forts. →

Så här kan variabeln `cube` visualiseras



Elementen `cube[0][0][1]=2` och `cube [0][0][2]=3` syns ej i bilden ovan, eftersom dessa element är skymda (i den valda projektionen)