729G46 Informationsteknologi och programmering

Tema 3, Föreläsning 3.1-3.2

Johan Falkenjack, johan.falkenjack@liu.se

Dugga 24/10, kl. 14.00-16.00

- Anmälan via LiU-appen eller studentportalen.
- **Ingen anmälan, ingen dugga.** Sista anmälningsdatum 10 dagar innan duggan.
- Kurslitteratur tillåten (bok/utskrift) + Kapitel från Pythondokumentationen som PDF (se kurshemsidan)
- Kod/Föreläsningsbilder/anteckningar **ej tillåtet**

Dugga 24/10, kl. 14.00-16.00

- **Kom i tid.** Duggan börjar 14.00, prick!
- Innan måste ni ha legitimerat er, hängt av er, loggat in, fått ev. böcker kontrollerade.

Dugga 24/10, kl. 14.00-16.00

- I SU-sal, ni använder Visual Studio Code + terminal
- Personliga inställningar kommer inte vara tillgängliga
- Se till att ni vet hur ni testar era program i via terminalen (**OBS!** *Inte* Playknappen i VSCode)
- Uppgifter liknar de som ni gjort i Pythonuppgifter 1-3
- Kommunikation (chat) med Johan samt inlämning sker via tenta-klient
- Assistent för hjälp med tekniska problem kommer finnas på plats

`for`-loop eller `while`-loop?

- `for`-loopar är bra på att iterera genom sekvenser utan att vi behöver ta hand om en massa logistik.

```
for value in values:
    if value == 42:
        print("I found the answer!")
```

- Du behöver inte ta hand om någon logistik vid användning av `for`-loop, dvs `i` är onödig i koden nedan.

```
i = 0
for value in values:
    if value == 42:
        print("I found the answer!")
    i += 1
```

`for`-loop eller `while`-loop?

In [3]:
```
import random

# while-loopar är bra om man vill fortsätta loopa under vissa
# förutsättningar - villkor
hemligt_ord = random.choice(["apelsin", "banan", "citron"])
gissat_ord = None
antal_försök = 0

while gissat_ord != hemligt_ord:
    if gissat_ord != None:
        print("Fel!")
    gissat_ord = input("Gissa vilket ord jag tänker på: ")
    antal_försök += 1

print("Rätt! Det tog " + str(antal_försök) + " försök.")
```

```
Gissa vilket ord jag tänker på: apelsin
Fel!
Gissa vilket ord jag tänker på: banan
Fel!
Gissa vilket ord jag tänker på: citron
Rätt! Det tog 3 försök.
```

Hitta felen

```
# Funktionen word_exists(words, words) ska returnera True om word finns
i
# listan words. Om word inte finns i listan words ska funktionen
returnera
# False
def word_exists(word, words):
    while word in words:
        if word == words:
            return True
        else:
            return False
```

Hitta felen

1. Blandar syntax för for-loop med while vilket resulterar i något helt annat.
    - Syntax: `while <uttryck som ger sanningsvärde>` , dvs uttrycket är `word in words` , där `in` är pythons medlemsoperator. Om while-loop används för att iterera över en lista behöver man en explicit variabel för index som ökar med 1 varje iteration, samt använda `words[index]` för att titta på värdena i listan.
    - Om man hade skrivit `for word in words` istället hade `for` -loopens variabel `word` "skrivit över" funktions-argumentet `word`
2. `word` kommer aldrig vara lika med `words` om `word` är ett enskilt värde och `words` är en lista
3. returnerar `False` i loopen - så fort `if` -grenen är falsk!

Hitta felen - med for

In [12]:
```python
# Funktionen word_exists(words, words) ska returnera True om word finns i
# listan words. Om word inte finns i listan words ska funktionen returnera
# False

# FEL. OBS! Nedanstående betyder inte det ni tror att det betyder
# else som hör ihop med for tolkas som "if no break", dvs kommer köras om
# ingen break inträffar i loopen.
def word_exists1(word, words):
    for trg_word in words:
        if word == trg_word:
            result = True
    else:
        result = False
    return result

# Se längre fram i denna föreläsning om f-stringar
print(f"{word_exists1('a', ['b', 'c', 'd'])=}")
print(f"{word_exists1('a', ['b', 'c', 'a', 'd'])=}")

# Korrekt
def word_exists2(word, words):
    for trg_word in words:
        if word == trg_word:
            return True
    return False
```

```
print(f"{word_exists2('a', ['b', 'c', 'd'])=}")
print(f"{word_exists2('a', ['b', 'c', 'a', 'd'])=}")
```

```
word_exists1('a', ['b', 'c', 'd'])=False
word_exists1('a', ['b', 'c', 'a', 'd'])=False
word_exists2('a', ['b', 'c', 'd'])=False
word_exists2('a', ['b', 'c', 'a', 'd'])=True
```

Hitta felen - korrekt lösning med while

In [14]:
```python
# Funktionen word_exists(words, words) ska returnera True om word finns i
# listan words. Om word inte finns i listan words ska funktionen returnera
# False
def word_exists3(word, words):
    index = 0
    while index < len(words):
        if word == words[index]:
            return True
        index += 1
    return False

print(f"{word_exists3('a', ['b', 'c', 'd'])=}")
print(f"{word_exists3('a', ['b', 'c', 'a', 'd'])=}")
```

```
word_exists3('a', ['b', 'c', 'd'])=False
word_exists3('a', ['b', 'c', 'a', 'd'])=True
```

Föreläsningsöversikt, FÖ 3.1-3.2

- Tema 3: Verktyget Python
    - Vad kan man använda Python till?
    - Python-paket
    - Lite mer om terminalen, kommandon och skalet

Föreläsningsöversikt, FÖ 3.1-3.2

- Python
    - argument till pythonprogram, läsa från stdin
    - strängformattering
    - repetition: scope, lokala och globala variabler
    - moduler, importera fil som modul
    - föränderliga och oföränderliga värden (datatyper)
    - referenser till värden - två variabler kan referera till samma värde
    - ny datatyp: dictionary
    - nästlade strukturer
    - rekursion

Tema 3: Verktyget Python

Python som verktyg för kognitionsvetare?

- **Artificiell intelligens:** maskininlärning, kunskapsrepresentation, programmera robotar
- **Lingvistik:** språkteknologiska tillämpningar, korpuslingvistik
- **Psykologi:** utforska data, räkna statistik, visualisera data
- **Neurovetenskap:** simulering, modellering, processa t.ex. fMRI-bilder
- **Allmänt:** automatisering, databearbetning, kognitionsvetenskapliga experiment, webbtjänster, IoT m.m.

Pythonpaket

- Pythonpaket tillhandahåller moduler som innehåller funktionalitet som inte finns med i Python "från början"
- Importeras på samma sätt som inbyggda moduler som `math`, `random` och `os`

Pythonpaket, exempel

- `pycodestyle` och `pyflakes` : kontrollera pythonkod för fel och stilbrott
- `requests` : enklare (att använda) HTTP-requests (kommunikation med webbservrar)
- `beautifulsoup` : paket för hantering av HTML och XML (t.ex. extrahera information från webben)
- `matplotlib` : datavisualisering
- `SciPy` : databeararbetning, diagram och grafer (statistik, linjär algebra, m.m.)
- `NLTK` : naturligt språkbehandling
- `Django` och `Flask` : två ramverk för webbutveckling

Kort om att installera paket

- Kommandot `pip`
- I LiUs Linuxmiljö får användare *inte* installera paket på systemnivå.
- För att installera paket på kan vi använda modulen venv för att skapa en virtuell miljö som vi kan aktivera och sedan installera paket i.

  https://www.ida.liu.se/~729G46/tips/virtual_environment/

  https://docs.python.org/3/library/venv.html

Terminalen, skalet, kommandon, program

Skript skrivna i Python som del av andra program

Filosofin bakom Unix

- **Summerat av Peter H. Salus (1994)**

  Write programs that do one thing and do it well.

  Write programs to work together.

  Write programs to handle text streams, because that is a universal interface.

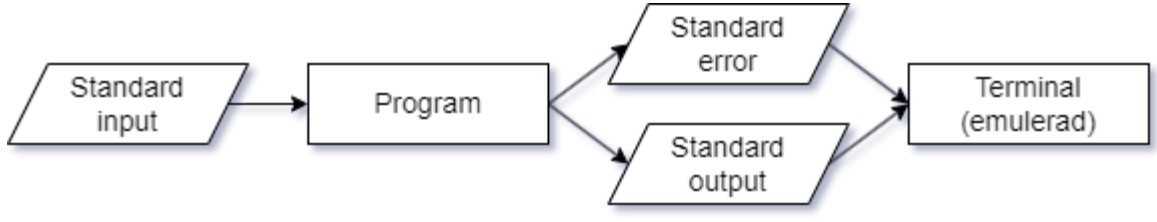
- **Det vill säga**

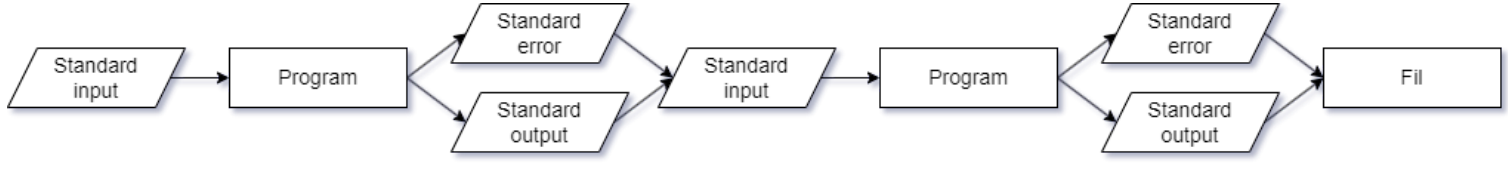
  Genom att skriva program som kan ta in text som input och ger text som output, kan man använda olika program med varandra genom att använda ett programs output som input till ett annat program.

  Genom att skriva program som endast gör en sak, så får vi byggstenar som vi kan använda efter våra behov.

Standard input, standard output, standard error

- Skalet hanterar input och output som strömmar (tänk kanaler/slangar/rör).
- Skalet kan *omdirigera* information som ett program skickar till stdout (standard output) så att det används som input till ett annat program.





Omdirigeringstecken

- Exempel på verktyg för detta, omdirigeringstecken
  - `>` skicka utdata från program (via stdout) till fil
  - `>>` skicka utdata från program (via stdout) till slutet på en fil
  - `<` skicka innehåll i fil till program via standard input
  - `|` skicka utdata från ett program som indata till annat program

Några exempel

- `sort` - sorterar de rader den får
- `uniq` - tar bort dubletter
- `cut` - plocka fram text från t.ex. en viss kolumn
- `grep` - leta efter rader som innehåller specificerad text

- `seq` - skriv ut en sekvens av siffror

- Fler exempel: https://developer.ibm.com/articles/au-unixtext/

Mönster i sökvägar i Unix

- `*` betyder 0 eller fler godtyckliga tecken
- `?` betyder *exakt ett* godyckligt tecken
- **Exempel**

  `ls *.py` visa alla filer som slutar på `.py`

  `cp data/*.csv csvfiler/` kopiera alla filer vars namn slutar på .csv från katalogen data till katalogen `csvfiler` (givet att `csvfiler` är en katalog)

  `cat sub1?.json` skriv ut innehållet i alla filer med namn som börjar på `sub1` följt av *ett* godtyckligt tecken följt av `.json`

Exekverbara filer i Unix

Exekverbara *textfiler* i unix

- Filen måste ha körrättighet

  ```
  chmod u+x filnamn
  ```

- Första raden *i filen* ska vara den tolk som ska användas för att tolka resten av textfilen.
- För Python 3 (pythonkod):

  ```
  #!/usr/bin/env python3
  ```

- För bash (skalkommandon)

  ```
  #!/bin/bash
  ```

Pythonprogram i Unix-miljö

Argument från terminalen

Exekverbara skript

Läsa från stdin

Pythonprogram som tar emot argument

- importera modulen `sys`
- variabeln `sys.argv` är en lista som innehåller allt som skrevs på kommandoraden (mellanslag som skiljetecken)
- Exempel

```
$ ./greetings.py Ada Bertil Caesar
sys.argv → ['./greetings.py', 'Ada', 'Bertil', 'Caesar']
```

f-strängar

Enklare strängformattering i Python

Enklare formattering av strängar

- f-strängar, en sträng med f som prefix

  `f"Jag är en f-sträng"`

- Kan innhålla uttryck innanför måsvingar, `{}`, som beräknas när strängen skapas.
- Exempel:

In [15]:
```
namn = "Guido"
f"Hej {namn}!"
```

Out[15]:
```
'Hej Guido!'
```

In [16]:
```
f"Jag vet att svaret är {21*2}"
```

Out[16]:
```
'Jag vet att svaret är 42'
```

Smidig utskrift av variabler

- Från Python 3.8 (3.8.10 används i LiUs Linuxmiljö)
- Lägg till `=` efter ett uttryck inom `{}` i en f-sträng, så följer även det icke-beräknade uttrycket med när strängen skapas
- Exempel:

In [17]:
```
my_value = 123
print(f"{my_value=}")
```

```
my_value=123
```

In [18]:
```
print(f"{40+2=}")
```

```
40+2=42
```

In [19]:
```
print(f"{my_value/2=}")
```

```
my_value/2=61.5
```

Bearbeta argument till pythonskript

med hjälp av `sys.argv`

Exempel, argv_demo.py

```python
#!/usr/bin/env python3
import sys
print(f"{sys.argv=}")
index = 0
while index < len(sys.argv):
    print(f"Argument {index}: {sys.argv[index]}")
    index += 1
```

```
$ python3 argv_demo.py apelsin banan citron
sys.argv=['argv_demo.py', 'apelsin', 'banan', 'citron']
Argument 0: argv_demo.py
Argument 1: apelsin
Argument 2: banan
Argument 3: citron
```

Läsa från standard input

med hjälp av `sys.stdin`

Pythonprogram som läser från stdin

- importera modulen `sys`
- läs rad för rad från `stdin`:

```
for line in sys.stdin:
    print(line)
```

- `line` kommer vara en sträng

Exempel, find.py

```
"""
Exempelanvändning som visar alla rader som innehåller ordet glass
# med pipe
$ cat alice.txt | python3 find.py glass
# med omdirigering
$ python3 find.py glass < alice.txt
"""
import sys

# för varje rad som fås via standard input (t.ex. från en pipe)
for line in sys.stdin:
    # om raden innehåller första argumentet som pythonprogrammet får
    if sys.argv[1].lower() in line.lower():
        # skriv ut raden
        print(line.strip())
```

```
$ cat alice.txt | python3 find.py glass
solid glass; there was nothing on it except a tiny golden key,
quite plainly through the glass, and she tried her best to climb
Soon her eye fell on a little glass box that was lying under
again, and the little golden key was lying on the glass table as
swim in the pool, and the great hall, with the glass table and
glass.  There was no label this time with the words `DRINK ME,'
and a crash of broken glass, from which she concluded that it was
Come and help me out of THIS!'  (Sounds of more broken glass.)
sounds of broken glass.  `What a number of cucumber-frames there
little glass table.  `Now, I'll manage better this time,'
```

Exempel, pretty_print.py

```python
#!/usr/bin/env python3
import sys

if len(sys.argv) != 2:
    print("Ange exakt ett hälsningsord som argument.")
    sys.exit(1)

for line in sys.stdin:
    line = line.split(";")
    print(f"{sys.argv[1]} användare {line[0]} ({line[1]} {line[2]})!")
```

```
$ grep "z" people.csv | ./pretty_print.py "Hejsan"
Hejsan användare jazsn103 (Jazmine Snyder)!
Hejsan användare thife781 (Thiago Fernandez)!
Hejsan användare izasa568 (Izaiah Savage)!
Hejsan användare loura901 (Louise Ramirez)!
Hejsan användare ezrgi707 (Ezra Giles)!
$ head people.csv | grep "z" | ./pretty_print.py "Hejsan"
Hejsan användare jazsn103 (Jazmine Snyder)!
Hejsan användare thife781 (Thiago Fernandez)!
```

Moduler i Python

Moduler

- Moduler tillhandahåller ytterligare funktioner m.m.
- Exempel: `random`, `sys`, `json`, `pickle`
- Vi kan använda de filer vi skriver som egna moduler

Importera en modul

- `import <modulnamn>`

  `<modulnamn>` används som prefix för att komma åt funktionalitet i modul
- `import <modulnamn> as <alias>`

  `<alias>` används som prefix för att komma åt funktionalitet i modul
- `from modulnamn import <namn_på_något_i_modulen>`

  importera `<namn_på_något_i_modulen>` direkt till den egna namnrymden (inget prefix behövs)

- `from <modulnamn> import *`

  importera allt från `<modulnamn>` till den egna namnrymden

- **OBS!** En import-sats per modul räcker. Samla alla importer högst upp i er fil.

Exempel - olika sätt att referera till funktion i en modul beroende på hur den importerats

In [20]:
```
# funktioner.py
print("Nu laddas funktionerna in.")

def print_twice(message):
    message = message.rstrip()
    print("1:", message)
    print("2:", message)

print("Funktionerna har laddats.")

if __name__ == "__main__":
    print("funktioner.py är \"huvudprogram\". __name__:", repr(__name__))
else:
    print("funktioner.py laddas in som en modul. __name__:", repr(__name__))
```

```
Nu laddas funktionerna in.
Funktionerna har laddats.
funktioner.py är "huvudprogram". __name__: '__main__'
```

Importera modulen med sin vanliga namnrymd

```python
# program1.py ###################################

# importera innehållet från filen funktioner.py
# lägger sig under modulnamnet funktioner
import funktioner

funktioner.print_twice("hejsan!")
```

```
$ python3 program1.py
Nu laddas funktionerna in.
Funktionerna har laddats.
funktioner.py laddas in som en modul. __name__: 'funktioner'
1: hejsan!
2: hejsan!
```

Importera modulen till en lokal namnrymd som vi döper själva

```python
# program2.py ###################################

# importera innehållet från filen funktioner.py
# lägger sig under modulnamnet f
import funktioner as f

f.print_twice("hejsan!")
```

```
$ python3 program2.py
Nu laddas funktionerna in.
Funktionerna har laddats.
funktioner.py laddas in som en modul. __name__: 'funktioner'
1: hejsan!
2: hejsan!
```

Importera allt i modulen till den lokala namnrymden

```python
# program3.py ###################################

# importera innehållet från filen funktioner.py
# lägger sig i den nuvarande namnrymnden
from funktioner import *

print_twice("hejsan!")
```

```
$ python3 program3.py
Nu laddas funktionerna in.
Funktionerna har laddats.
funktioner.py laddas in som en modul. __name__: 'funktioner'
1: hejsan!
2: hejsan!
```

Kod som körs när något läses in som modul

- Vid import körs koden som står i modulen
- Vi kan ta reda på om koden körs som "huvudprogram" eller om den importeras.
- Systemvariabeln `__name__` innehåller en sträng med den namnrymd (namespace) som gäller.
- Om programmet körs som huvudprogram gäller `__name__ == "__main__"`
- Dvs, vi kan skriva kod som endast kör när filen körs som huvudprogram, t.ex. tester

In [21]:
```python
# funktioner.py
print("Nu laddas funktionerna in.")

def print_twice(message):
    message = message.rstrip()
    print("1:", message)
    print("2:", message)

print("Funktionerna har laddats.")

if __name__ == "__main__":
    print("funktioner.py är \"huvudprogram\". __name__:", repr(__name__))
else:
    print("funktioner.py laddas in som en modul. __name__:", repr(__name__))
```

```
Nu laddas funktionerna in.
Funktionerna har laddats.
funktioner.py är "huvudprogram". __name__: '__main__'
```

Några varningar

- Att importera kan ***överskugga*** variabler och funktioner

  Speciellt riskabelt är det att importera allt i en modul till den lokala namnrymden, dvs att köra

  ```
  from min_modul import *
  ```

  Om `min_modul` innehåller t.ex. en funktion som heter `print` så kommer vi inte längre åt den vanliga `print` -funktionen
- En modul kan bara importeras **EN** gång

  Om vi kör `import min_modul` , och sedan gör ändringar i `min_modul` , så kan vi inte få tillgång till dessa ändringar i en pågående Python-session (t.ex. när vi kör i *interactive mode*) genom att köra `import min_modul` igen.

  Lösning 1: Starta om Python.

  Lösning 2: Använd funktionen `reload` från den inbyggda modulen `importlib`

Scope

lokala och globala variabler

Lokala och globala variabler

- nyckelordet `global`
- **lokala variabler**: variabler i funktioner är bara åtkomliga i funktionen
- **globala variabler**: variabler utanför funktioner kan kommas åt inifrån en funktion med nyckelordet `global`

In [22]:
```
my_variable = "spam"

print(f"Before calling my_function: {my_variable=}")

def my_function():
    print(f"Inside my_function: {my_variable=}")

my_function()

print(f"After calling my_function: {my_variable=}")
```

```
Before calling my_function: my_variable='spam'
Inside my_function: my_variable='spam'
After calling my_function: my_variable='spam'
```

In [23]:
```
my_variable = "spam"

print(f"Before calling my_function: {my_variable=}")

def my_function():
    my_variable = "eggs"
    print(f"Inside my_function: {my_variable=}")

my_function()

print(f"After calling my_function: {my_variable=}")
```

```
Before calling my_function: my_variable='spam'
Inside my_function: my_variable='eggs'
After calling my_function: my_variable='spam'
```

In [24]:
```
my_variable = "spam"

print(f"Before calling my_function: {my_variable=}")

def my_function():
    global my_variable
    my_variable = "eggs"
    print(f"Inside my_function: {my_variable=}")

my_function()

print(f"After calling my_function: {my_variable=}")
```

```
Before calling my_function: my_variable='spam'
Inside my_function: my_variable='eggs'
After calling my_function: my_variable='eggs'
```

Oföränderliga och föränderliga värden

eng. *immutable* vs. *mutable*

Oföränderliga värden

- I Python är strängar, heltal, flyttal, sanningsvärden, None och tupler oföränderliga (eng. *immutable*).
- Detta betyder att man inte kan ändra på dessa värden

    t.ex. så kan man inte ändra värdet 5 till värdet 1, värdet 5 är alltid 5.

    Python behandlar strängar på samma sätt; i Python kan vi inte ändra strängen "hej" till strängen "nej"

Föränderliga värden

- Av de datatyper som vi stött på så är tillhör listor och dictionaries (som vi stöter på strax) kategorin förändringsbara värden (eng. *mutable*).
- Detta betyder att vi kan ändra på dessa värden.
- Vi kan t.ex. byta ut första elemenetet i listan `[1, 2, 3]` till 5

In [25]:
```
l = [1, 2, 3] # lista
l[0] = 5
print(l)
```

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

Oföränderliga listor? `tuple`

- Datatypen `tuple` (*sve.* tupel) fungerar som en lista, förrutom att man inte kan byta dess element.
- Parenteser används för att skapa en tuple.
- **Obs!** att `tuple` i Python är vad vi i diskreta matematiken kallat -tupler, dvs de behöver inte ha specifikt två element

In [26]:
```
t = (1, 2, 3) # tupel
t[0] = 5
```

```
---------------------------------------------------------------------
--
TypeError                                 Traceback (most recent call las
t)
Cell In[26], line 2
      1 t = (1, 2, 3) # tupel
----> 2 t[0] = 5

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

Referenser till värden

Variabler som referenser till värden

- I de flesta fall är det bättre att se variabler som "etiketter" som refererar till värden.
- En variabeltilldelning `l1 = [1, 2, 3]` kan ses som att vi säger "låt etiketten `l1` referera till värdet ..."
- Vi kan låta olika etiketter referera till samma värde: `l2 = l1`

Konsekvenser av att olika variabler kan referera till samma värde

In [27]:
```
l1 = [1, 2, 3]
# låt l2 referera till samma värde som l1
l2 = l1
# ändra första elementet i värdet som l1 refererar till
l1[0] = "hoppsan"
# vad kommer vi få för utskrifter?
print(l1)
print(l2)
```

```
['hoppsan', 2, 3]
['hoppsan', 2, 3]
```

Operatorn + på listor returnerar en ny lista, .append() ändrar på existerande lista

In [28]:
```
frukter_1 = ["apelsin", "banan", "citron"]
frukter_2 = frukter_1
frukter_3 = ["druva"]
print(f"* start:\n {frukter_1=},\n {frukter_2=},\n {frukter_3=}")
```

```
* start:
 frukter_1=['apelsin', 'banan', 'citron'],
 frukter_2=['apelsin', 'banan', 'citron'],
 frukter_3=['druva']
```

In [29]:
```
# .append() -> lägg till existerande lista
frukter_1.append("fikon")
print(f'* efter frukter_1.append("fikon"):')
print(f" {frukter_1=}\n {frukter_2=}")
```

```
* efter frukter_1.append("fikon"):
 frukter_1=['apelsin', 'banan', 'citron', 'fikon']
 frukter_2=['apelsin', 'banan', 'citron', 'fikon']
```

Operatorn + på listor returnerar en ny lista, .append() ändrar på existerande lista

In [30]:
```
# OBS! Något att se upp för listor fungerar += som .append()
frukter_1 += ["oj"]
print(f'* efter frukter_1 += ["oj"]:')
print(f" {frukter_1=}\n {frukter_2=}")
```

```
* efter frukter_1 += ["oj"]:
 frukter_1=['apelsin', 'banan', 'citron', 'fikon', 'oj']
 frukter_2=['apelsin', 'banan', 'citron', 'fikon', 'oj']
```

In [31]:
```
# operatorn + skapar en ny lista från operanderna # (om de är listor)
frukter_1 = frukter_1 + frukter_3
print("* efter frukter_1 + frukter_3:")
print(f" {frukter_1=},\n {frukter_2=},\n {frukter_3=}")
```

```
* efter frukter_1 + frukter_3:
 frukter_1=['apelsin', 'banan', 'citron', 'fikon', 'oj', 'druva'],
 frukter_2=['apelsin', 'banan', 'citron', 'fikon', 'oj'],
 frukter_3=['druva']
```

Operatorn + på listor returnerar en ny lista, .append() ändrar på existerande lista

In [32]:
```
# Varför ändras inte frukter 2 här?
frukter_1 += ["varför"]
print(f'* efter frukter_1 += ["varför"]:')
print(f" {frukter_1=}\n {frukter_2=}")
```

```
* efter frukter_1 += ["varför"]:
 frukter_1=['apelsin', 'banan', 'citron', 'fikon', 'oj', 'druva', 'varfö
r']
 frukter_2=['apelsin', 'banan', 'citron', 'fikon', 'oj']
```

Föränderliga värden som skickas som argument till funktioner

In [33]:
```
def change_and_return_new_list(a_list):
    a_list.append("list was changed")
    return [1, 2, 3]

my_list = ["a", "b", "c"]

# låt also_my_list också referera till värdet som my_list refererar till
also_my_list = my_list

# låt my_list referera till det som change_and_return_new_list returnerar
my_list = change_and_return_new_list(my_list)

print(my_list)
print(also_my_list)
```

```
[1, 2, 3]
['a', 'b', 'c', 'list was changed']
```

Ny datatyp: dictionary

Dictionary - en mängd nyckel-värde-par

- Ett *dictionary* har *element* precis som en lista, men istället för index, kommer man åt elementen med via ***nycklar***.
- Varje **nyckel** (eng. *key*) är associerat med ett **värde** (eng. *value*).
- Alla datatyper som är oföränderliga (*immutable*) kan användas som nycklar, t.ex. flyttal, heltal, strängar, tupler.
- Alla datatyper kan lagras som värden.
- Ett dictionary omges av `{}` och består av nycklar och värden:

```
{ nyckel_1:värde1, nyckel_2:värde2, ... , nyckel_n:värde_n }
```

Hämta värde: istället för index använder vi nyckeln

- Jämfört med en lista så använder vi nyckeln istället för index för att "komma åt" (eng. *access*) värdena.

In [34]:
```
dictionary1 = {"nyckel 1":"värde 1", 345:"värde 2", (3):54 }

print(dictionary1["nyckel 1"])
```

```
värde 1
```

In [35]:
```
print(dictionary1[345])
```

```
värde 2
```

In [36]:
```
 print(dictionary1[(3)])
```

```
54
```

Vi kan lägga till element till ett dictionary

Till skillnad från att använda index som är större än längden på listan, så kan man *lägga till nya värden* i ett dictionary genom att *använda en ny nyckel*.

In [37]:
```
dictionary1 = {"nyckel1":"värde 1", 345:"värde 2", (3):54 }
dictionary1["ny nyckel"] = "nytt värde"
dictionary1["böcker"] = ["bok 1", "bok 2", "bok 3"]
```

In [38]:
```
print(dictionary1["ny nyckel"])
```

```
nytt värde
```

In [39]:
```
print(dictionary1["böcker"])
```

```
['bok 1', 'bok 2', 'bok 3']
```

In [40]:
```
print(dictionary1)
```

```
{'nyckel1': 'värde 1', 345: 'värde 2', 3: 54, 'ny nyckel': 'nytt värde',
'böcker': ['bok 1', 'bok 2', 'bok 3']}
```

Vi kan ändra värden genom att använda nyckeln

Precis som att man kan ange ett index vid tilldelning av ett värde på ett element i en lista, så kan man ange en nyckel vid tilldelning av ett associerat värde i en dictionary.

In [41]:
```
dictionary1 = {"nyckel1":"värde 1", 345:"värde 2", (3):54 }
print(dictionary1["nyckel1"])
```

```
värde 1
```

In [42]:
```
dictionary1["nyckel1"] = 1024
```

In [43]:
```
print(dictionary1["nyckel1"])
```

```
1024
```

Alla nycklar/värden/nyckel-värdepar för ett dictionary

- Punktnotation för att komma åt alla
  - nycklar: `dict.keys()`
  - värden: `dict.values()`
  - nyckel-värdepar: `dict.items()`
- `dict` refererar här till ett värde av typen dictionary, denna notation används även i pythondokumentationen
- Funktioner som anropas via punktnotation kallas för *metoder*. Mer om detta i Tema 4-6.
- Metoderna `dict.keys()`, `dict.values()` och `dict.items()` returnerar en **vy** (eng. *view*) in i dictionary-värdet
  - Ett *view*-objekt beter sig ungefär som en lista (men är ingen riktig lista). Se https://docs.python.org/3/library/stdtypes.html#dictionary-view-objects för mer information.

Loopa igenom ett dictionary

Repetition, for-loopen

```
# syntax
for element in sequence:
    # gör saker med element

# ok, men försök att undvika:
for index in range(len(sequence)):
    # gör saker med sequence[index]

# felaktigt (ingen användning av for-loopen)
index = 0
for element in sequence:
    # gör saker med sequence[index]
    index += 1
```

Ordning i ett dictionary

- Sedan Python 3.7 (vi använder 3.8.10 på LiUs datorer idag) följer elementen i ett dictionary ordningen de lades till.

  I Python 2.x och Python 3.1-6 fanns ingen *bestämd* ordning mellan elementen i ett dictionary och man kunde alltså inte räkna med samma beteende varje gång man loopade över ett dictionary.

  Många internetkällor säger fortfarande att dictionary är en oordnad datatyp och att man bör använda datatypen OrderedDict om ordningen är viktig, det är för att detta är en relativt ny förändring.
- Vyer in i dictionaries (från `dict.keys()`, `dict.values()` och `dict.items()`) följer samma ordning.

Loopa genom nycklar i ett dictionary

In [44]:
```
# loopa genom nycklar, explicit
for key in dictionary1.keys():
    print(f"{key=}")
    print(f"{dictionary1[key]=}")
```

```
key='nyckel1'
dictionary1[key]=1024
key=345
dictionary1[key]='värde 2'
key=3
dictionary1[key]=54
```

In [45]:
```
# loopa genom nycklar, implicit
for key in dictionary1:
    print(f"{key=}")
    print(f"{dictionary1[key]=}")
```

```
key='nyckel1'
dictionary1[key]=1024
key=345
dictionary1[key]='värde 2'
key=3
dictionary1[key]=54
```

Vi kan loopa genom värden i ett dictionary

In [46]:
```
for value in dictionary1.values():
    print(f"{value=}")
```

```
value=1024
value='värde 2'
value=54
```

Vi kan loopa genom par av nycklar och värden

- `dict.items()` returnerar en *vy* som kan användas som en lista av tupler med två element per tupel.

In [47]:
```
# loopa genom par av nycklar och värden explicit
for key, value in dictionary1.items():
    print(f"{key=}")
    print(f"{value=}")
```

```
key='nyckel1'
value=1024
key=345
value='värde 2'
key=3
value=54
```

In [48]:
```
# en lista av tupler med två element per tupel kan loopas igenom på samma sätt
list_of_tuples = [("ett", 1), ("två", 2), ("tre", 3)]
for word, number in list_of_tuples:
    print(f"{word=}")
    print(f"{number=}")
```

```
word='ett'
number=1
```

```
word='två'
number=2
word='tre'
number=3
```

Bearbetning av nästlade datastrukturer

Nästlade datastrukturer

- A är en nästlad datastruktur om

  A innehåller flera värden

  Ett av värdena i A i sin tur innehåller flera värden

In [49]:
```
list_of_lists = [["Ada Lovelace", 1815], ["Joan Clarke", 1917]]
list_of_dicts = [ { "name": "Ada Lovelace", "birthyear": 1815 }, { "name": "Jo
dict_with_some_list_value = { "name": "ditto", "abilities": ["imposter", "limb
```

Hur kommer vi åt nästlade värden?

In [50]:
```
lista1 = [["a", "b", "c"], ["d", "e", "f"]]
```

In [51]:
```
# första elementet i lista1
lista1[0]
```

Out[51]:
```
['a', 'b', 'c']
```

In [52]:
```
# första elementet i första elementet i lista1
lista1[0][0]
```

Out[52]:
```
'a'
```

In [53]:
```
# andra elementet i första elementet i lista1
lista1[0][1]
```

Out[53]:
```
'b'
```

Hur kommer vi åt nästlade värden?

In [54]:
```
dict1 = { "frukter": ["a", "b", "c"],
          "bilar": ["d", "e", "f"] }
```

In [55]:
```
# värdet associerat med nyckeln "frukter"
dict1["frukter"]
```

Out[55]:
```
['a', 'b', 'c']
```

In [56]:
```
# första elementet i listan associerad med nyckeln "frukter"
dict1["frukter"][0]
```

Out[56]:
```
'a'
```

In [57]:
```
# andra elementet i listan associerad med nyckeln "frukter"
dict1["frukter"][1]
```

Out[57]:
```
'b'
```

En nästlad loop för att bearbeta en nästlad datastruktur

In [58]:
```
yttre_lista = [ ["a", "b", "c"], ["d", "e", "f", "g"] ]

# om vi för varje inre lista i yttre_lista vill skriva ut den inre listans ele
yttre_index = 0
while yttre_index < len(yttre_lista):
    inre_lista = yttre_lista[yttre_index]

    # kod som skriver ut varje element i inre_lista
    inre_index = 0
    while inre_index < len(inre_lista):
        print(inre_lista[inre_index])
        inre_index += 1
    yttre_index += 1
```

```
a
b
c
d
e
f
g
```

En nästlad loop för att bearbeta en nästlad datastruktur

In [59]:
```
yttre_lista = [ ["a", "b", "c"], ["d", "e", "f", "g"] ]

# om vi för varje inre lista i yttre_lista vill skriva ut den inre listans ele
for yttre_element in yttre_lista:
    for inre_element in yttre_element:
        print(inre_element)
```

```
a
b
c
d
e
f
g
```

Nästlade strukturer med dictionaries

Bearbeta lista av dictionaries 1

In [60]:
```
pokemons = [ { "name": "bulbasaur", "abilities": ["chlorophyll", "overgrow"] }
             { "name": "squirtle", "abilities": ["rain-dish", "torrent"] } ]

def print_dictionaries(list_of_dictionaries):
    # gå igenom listan med dictionaries
    for dictionary in list_of_dictionaries:
        # gå igenom alla nycklar i aktuellt dictionary
        for key in dictionary.keys():
            print(f"The key {key} has the value: {dictionary[key]}")

print("pokemons:")
print_dictionaries(pokemons)
```

```
pokemons:
The key name has the value: bulbasaur
The key abilities has the value: ['chlorophyll', 'overgrow']
The key name has the value: squirtle
The key abilities has the value: ['rain-dish', 'torrent']
```

Bearbeta lista av dictionaries 1

In [61]:
```
books = [ { "title": "Introduction to Python", "pages": 314 },
          { "title": "Another introduction to Python", "pages": 413 } ]


print("books:")
print_dictionaries(books)
```

```
books:
The key title has the value: Introduction to Python
The key pages has the value: 314
The key title has the value: Another introduction to Python
The key pages has the value: 413
```

Bearbeta lista av dictionaries 2

In [62]:
```python
def print_dictionaries(list_of_dictionaries):
    # gå igenom listan med dictionaries
    for dictionary in list_of_dictionaries:

        # gå igenom alla nycklar i varje dictionary
        for key in dictionary:

            # i de fall som värdet för en nyckel är en lista
            if type(dictionary[key]) == list:
                print(f"Value of key {key} is a list:")

                # skriv ut varje värde i listan dictionary[key]
                for value in dictionary[key]:
                    print(f"- {value}")
            # när värdet i dictionaryt inte är en lista
            else:
                print(f"The key {key} has the value: {dictionary[key]}")
```

Bearbeta lista av dictionaries 2

In [63]:
```
print("pokemons:")
print_dictionaries(pokemons)
print("\nbooks:")
print_dictionaries(books)
```

```
pokemons:
The key name has the value: bulbasaur
Value of key abilities is a list:
- chlorophyll
- overgrow
The key name has the value: squirtle
Value of key abilities is a list:
- rain-dish
- torrent

books:
The key title has the value: Introduction to Python
The key pages has the value: 314
The key title has the value: Another introduction to Python
The key pages has the value: 413
```

Leta efter värde i nästlad lista

In [64]:
```
blandad_lista = [ "a", ["b", "c"], "d", "e", ["f", "g"] ]
def look_for_value(needle, haystack):
    # bearbeta yttre listan
    for value in haystack:
        # bearbetning av yttre värden som är listor
        if type(value) == list:
            for inner_value in value:
                if inner_value == needle:
                    return True
        # om yttre värde inte är en lista, kolla om det är det vi letar efter
        elif value == needle:
            return True
    return False
```

In [66]:
```
look_for_value("c", blandad_lista)
```

Out[66]:
```
True
```

Operatorn `in`

Operatorer: repetition

- Exempel på operatorer:

  +, *, -, /, and, or, not

- Operatorer kan ses som funktioner

```
plus(x, y)
minus(x, y)
funktionen_or(True, False)
funktionen_not(True)
```

- Argument till operatorer kallas för *operander*
- Operatorer är oftast *binära* (tar exakt två argument), eller *unära* (tar exakt ett argument)

Operatorerna `in` och `not in`

- Operatorn in är en binär operator som undersöker medlemsskap.
- Operanderna är ett värde och en behållare, t.ex. sekvenser eller dictionary-vyer.
- Exempel på sekvenser är: strängar, listor, tupler, dictionaries (implicit dess nycklar)
- Operatorn `in` returnerar `True` om värdet är en medlem i behållaren.
- Operatorn `not in` returnerar `True` om värdet inte är en medlem i behållaren.

Exempel

In [67]:
```
l = [10, "tre", 3.14]
print("tre" in l)
print("fyra" not in l)
```

```
True
True
```

In [68]:
```
wordlist = { "bil" : "car", "fisk": "fish" }
print("fisk" in wordlist)
print("fish" in wordlist)
```

```
True
False
```

Rekursion

Rekursiva funktioner

- En funktion är *rekursiv* om den anropar sig själv direkt eller indirekt.

Rekursion

- När en funktion anropar sig själv
- Vad tror ni händer nedan?

In [ ]:
```python
def skriv_ut_hej(heltal):
    print(f"Hej {heltal}")
    skriv_ut_hej(heltal + 1)
    print("Klar!")

skriv_ut_hej(0)
```

Slutvillkor - Stoppa rekursionen

In [70]:
```python
def skriv_ut_hej(heltal):
    if heltal == 10:
        print("Klar!")
    else:
        print(f"Hej {heltal}")
        skriv_ut_hej(heltal + 1)

skriv_ut_hej(0)
```

```
Hej 0
Hej 1
Hej 2
Hej 3
Hej 4
Hej 5
Hej 6
Hej 7
Hej 8
Hej 9
Klar!
```

Rekursiv problemlösning

- För att lösa ett problem rekursivt, behöver vi formulera eller dela upp det på ett sådant sätt så att delarna är **mindre varianter av det större problemet**.

- Om vi har problemet summera talen `1`, `2`, `75`, `6` och `7`. Att uttrycka det som `1 + 2 + 75 + 6 + 7` är att försöka lösa allt på en gång.

- Ett alternativt sätt är uttrycka det som `1` + *summan av de resterande talen*; `2, 75, 6, 7`
- Summan av de resterande talen i sin tur kan vi uttrycka på samma sätt, `2` + *summan av de resterande talen*; `75, 6, 7`
- Vi har hittat ett sätt att formulera problemet så att vi kan tillämpa samma lösning på mindre och mindre varianter av samma problem.

Ett första försök

In [71]:
```
numbers = [1, 2, 75, 6, 7]

def sum_list_rec(values):
    return values[0] + sum_list_rec(values[1:])

result = sum_list_rec(numbers)
print(result)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call las
t)
Cell In[71], line 6
      3 def sum_list_rec(values):
      4     return values[0] + sum_list_rec(values[1:])
----> 6 result = sum_list_rec(numbers)
      7 print(result)

Cell In[71], line 4, in sum_list_rec(values)
      3 def sum_list_rec(values):
----> 4     return values[0] + sum_list_rec(values[1:])

Cell In[71], line 4, in sum_list_rec(values)
      3 def sum_list_rec(values):
----> 4     return values[0] + sum_list_rec(values[1:])
```

```
    [... skipping similar frames: sum_list_rec at line 4 (3 times)]

Cell In[71], line 4, in sum_list_rec(values)
      3 def sum_list_rec(values):
----> 4     return values[0] + sum_list_rec(values[1:])

IndexError: list index out of range
```

Ett första försök - fel när `values == []`

- Vad bör anropet sum_list_rec([]) bli?
- Vad är summan av alla tal i en tom lista?
- Vad kan vi lägga till för att hantera det fallet?

In [72]:
```
numbers = [1, 2, 75, 6, 7]

def sum_list_rec(values):
    return values[0] + sum_list_rec(values[1:])

result = sum_list_rec(numbers)
print(result)
```

```
---------------------------------------------------------------------
--
IndexError                                Traceback (most recent call las
t)
Cell In[72], line 6
      3 def sum_list_rec(values):
      4     return values[0] + sum_list_rec(values[1:])
----> 6 result = sum_list_rec(numbers)
      7 print(result)

Cell In[72], line 4, in sum_list_rec(values)
      3 def sum_list_rec(values):
```

```
----> 4     return values[0] + sum_list_rec(values[1:])

Cell In[72], line 4, in sum_list_rec(values)
      3 def sum_list_rec(values):
----> 4     return values[0] + sum_list_rec(values[1:])

    [... skipping similar frames: sum_list_rec at line 4 (3 times)]

Cell In[72], line 4, in sum_list_rec(values)
      3 def sum_list_rec(values):
----> 4     return values[0] + sum_list_rec(values[1:])

IndexError: list index out of range
```

Summera alla tal i en lista

In [73]:
```
numbers = [1, 2, 75, 6, 7]

def sum_list_rec(values):
    # summan av värdena i en tom lista är 0
    if not values:
        return 0
    # summan av värdena i en icke-tom lista är första värdet + summan av
    # resten av värdena i listan
    return values[0] + sum_list_rec(values[1:])

result = sum_list_rec(numbers)
print(result)
```

```
91
```

Nytt problem: summan av alla tal från 0 till n?

- Summan av alla tal från 0 till n är samma sak som summan av alla tal från n till 0.
- Summan av alla tal från n till 0 är samma sak som n plus summan av alla tal från $n-1$ till 0
- Mer formellt:

$$\sum_{0}^{n} = n + \sum_{0}^{n-1} \ (givet\ att\ n > 0)$$

Försök 1: Summan av alla tal från n till 0

In [74]:
```
def sum_rec(n):
    return n + sum_rec(n-1)
```

In [75]:
```
print(sum_rec(3))
```

```
---------------------------------------------------------------------
--
RecursionError                            Traceback (most recent call las
t)
Cell In[75], line 1
----> 1 print(sum_rec(3))

Cell In[74], line 2, in sum_rec(n)
      1 def sum_rec(n):
----> 2     return n + sum_rec(n-1)

Cell In[74], line 2, in sum_rec(n)
      1 def sum_rec(n):
----> 2     return n + sum_rec(n-1)

    [... skipping similar frames: sum_rec at line 2 (2971 times)]

Cell In[74], line 2, in sum_rec(n)
      1 def sum_rec(n):
----> 2     return n + sum_rec(n-1)
```

Försök 1: Summan av alla tal från n till 0 - när ska vi sluta?

- Vi behöver identifiera när vi ska sluta...
- Vilket värde på n borde vara det sista vi tar hand om?

In [76]:
```
def sum_rec(n):
    return n + sum_rec(n-1)
```

Försök 1: Summan av alla tal från n till 0 - när ska vi sluta?

- Vi behöver identifiera när vi ska sluta...
- Vilket värde på n borde vara det sista vi tar hand om?

In [76]:
```
def sum_rec(n):
    return n + sum_rec(n-1)
```

In [77]:
```
def sum_rec(n):
    if n == 0:
        return 0
    return n + sum_rec(n-1)
```

In [78]:
```
print(sum_rec(100))
```

```
5050
```

Mönster för att skriva en rekursiv funktion

- Basfallet

  Vi letar efter det "enklaste" fallet av problemet som ska lösas, basfallet.

  Basfallet är det fall som är så enkelt att svaret är givet.
- Rekursionsfall

  Basfallet följs sedan av ett eller flera rekursiva anrop på en "enklare" variant av ursprungsproblemet.
- Se till att alla `return` -satser returnerar värde av samma datatyp

Mönster för att skriva en rekursiv funktion

- Vad är basfallet? När ska rekursionen avslutas?
- Vilka andra fall finns det?

    När ska något utföras på värdet?

    När ska värdet sparas?

    När ska värdet kastas?

    ...

Summera alla tal i en lista

In [79]:
```python
numbers = [1, 2, 75, 6, 7, 75, 6, 7, 75, 6, 7]

def sum_list_rec(values):
    # summan av värdena i en tom lista är 0
    if not values:
        return 0

    # summan av värdena i en icke-tom lista är första värdet + summan av
    # resten av värdena i listan
    return values[0] + sum_list_rec(values[1:])

result = sum_list_rec(numbers)
print(result)
```

```
267
```

Leta efter ett värde i en lista

In [81]:
```python
numbers = [1, 2, 75, 6, 7, 75, 6, 7, 75, 6, 7]

def look_for_value_rec(value, values):
    # Inget värde kan finnas i en tom lista
    if not values:
        return False

    # om första värdet är det värde vi letar efter kan vi sluta leta
    elif values[0] == value:
        return True

    # om inte första värdet var det vi letade efter så returnera
    # resultatet av att leta efter värdet i resten av listan
    else:
        return look_for_value_rec(value, values[1:])

print(f"{look_for_value_rec(314, numbers)=}")
print(f"{look_for_value_rec(7, numbers)=}")
```

```
look_for_value_rec(314, numbers)=False
look_for_value_rec(7, numbers)=True
```

Spara bara strängar

In [82]:
```
mixed_list = ["ett", 2, "sjuttiofem", 6, 7, 75, 6, "sju"]

def keep_strings_rec(values):
    # om listan är tom så resultatet en tom lista
    if not values:
        return []

    # om första värdet i listan är en sträng så är resultatet
    # en lista med den strängen + alla strängar i resten av values
    elif type(values[0]) == str:
        return [values[0]] + keep_strings_rec(values[1:])

    # om första värdet i listan inte var en sträng så är resultatet
    # alla strängar i resten av listan
    else:
        return keep_strings_rec(values[1:])

print(keep_strings_rec(mixed_list))
```

```
['ett', 'sjuttiofem', 'sju']
```

Varning för globala variabler

Den andra funktionen ger 0 poäng på duggan...

In [83]:
```
def keep_strings_good(values):
    if not values:
        return []
    elif type(values[0]) == str:
        return [values[0]] + keep_strings_good(values[1:])
    else:
        return keep_strings_good(values[1:])

answer = []
def keep_strings_bad(values):
    if len(values) > 0:
        if type(values[0]) == str:
            answer.append(values[0])
        keep_strings_bad(values[1:])
    return answer

# men det funkar ju eller?
print(keep_strings_good(["ett", 2, "sjuttiofem", 6, 7]))
print(keep_strings_bad(["ett", 2, "sjuttiofem", 6, 7]))
```

Den andra funktionen ger 0 poäng på duggan...

- kan vi använda funktionerna flera gånger?

In [84]:
```
print("Två anrop till keep_strings_good()")
print(keep_strings_good(["ett", 2, "sjuttiofem", 6, 7]))
print(keep_strings_good(["ett", 2, "sjuttiofem", 6, 7]))

print("\nTvå anrop till keep_strings_bad()")
print(keep_strings_bad(["ett", 2, "sjuttiofem", 6, 7]))
print(keep_strings_bad(["ett", 2, "sjuttiofem", 6, 7]))
```

```
Två anrop till keep_strings_good()
['ett', 'sjuttiofem']
['ett', 'sjuttiofem']

Två anrop till keep_strings_bad()
['ett', 'sjuttiofem', 'ett', 'sjuttiofem']
['ett', 'sjuttiofem', 'ett', 'sjuttiofem', 'ett', 'sjuttiofem']
```

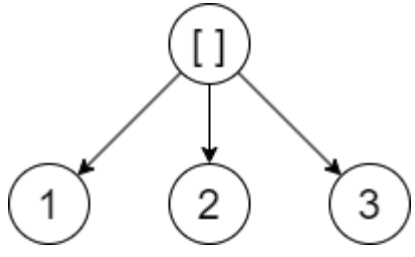
Räcker det att flytta in `answer` i funktionen?

In [85]:
```
def keep_strings_bad(values):
    answer = []
    if len(values) > 0:
        if type(values[0]) == str:
            answer.append(values[0])
        keep_strings_bad(values[1:])
    return answer


print(keep_strings_bad(["ett", 2, "sjuttiofem", 6, 7]))
```

```
['ett']
```

Platt lista som trädstruktur

- `[ 1, 2, 3]`
- **Problem**: Vad är summan av alla löv?
- Alla noder är löv, dvs alla noder har värden.
- Vi anropar `sum_rec([ 1, 2, 3])`



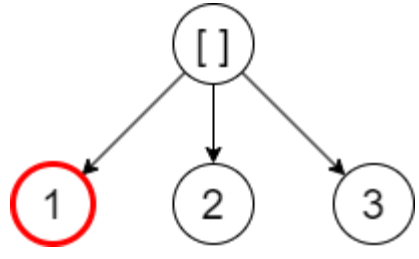

```
def sum_rec(values):
  if not values:
      return 0
  else:
      return values[0] + sum_rec(values[1:])
```

Platt lista som trädstruktur

- `[ 1, 2, 3]`
- **Problem**: Vad är summan av alla löv?
- Noden är ett löv.
- Beräkna 1 + summan av övriga värden.

`sum_rec` anropas med *resten* av listan som argument:

`sum_rec([ 2, 3])`

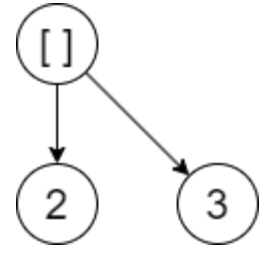



```
def sum_rec(values):
if not values:
  return 0
else:
  return values[0] + sum_rec(values[1:])
```

Platt lista som trädstruktur

- `[ 2, 3]`
- **Problem**: Vad är summan av alla löv?
- Alla noder är löv, dvs alla noder har värden.

```
def sum_rec(values):
  if not values:
      return 0
  else:
      return values[0] + sum_rec(values[1:])
```



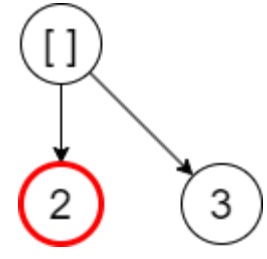
Platt lista som trädstruktur

- `[ 2, 3]`
- **Problem**: Vad är summan av alla löv?
- Noden är ett löv.
- Beräkna 2 + summan av övriga värden.

  `sum_rec` anropas med *resten* av listan som argument:

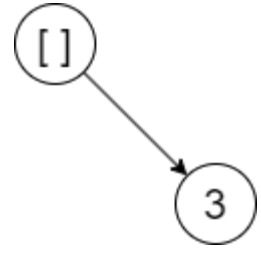
  `sum_rec([ 3])`

```
def sum_rec(values):
if not values:
  return 0
else:
  return values[0] + sum_rec(values[1:])
```



Platt lista som trädstruktur

- `[ 3]`
- **Problem**: Vad är summan av alla löv?
- Alla noder är löv, dvs alla noder har värden.



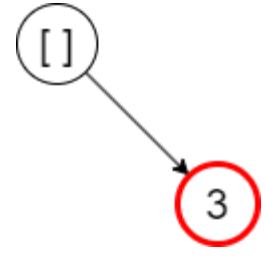

```
def sum_rec(values):
  if not values:
      return 0
  else:
      return values[0] + sum_rec(values[1:])
```

Platt lista som trädstruktur

- `[ 3]`
- **Problem**: Vad är summan av alla löv?
- Noden är ett löv.
- Beräkna 3 + summan av övriga värden.

`sum_rec` anropas med *resten* av listan som argument:

`sum_rec([ ])`




```
def sum_rec(values):
if not values:
  return 0
else:
  return values[0] + sum_rec(values[1:])
```

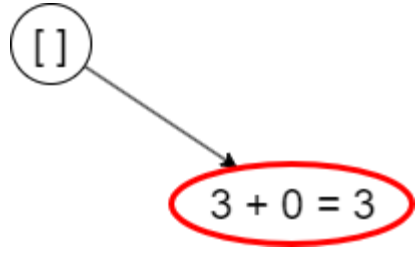
Platt lista som trädstruktur

- `[ ]`
- **Problem**: Vad är summan av alla löv?
- Inga noder finns.
- Summan av inga värden är 0 så vi returnerar 0.


```
def sum_rec(values):
  if not values:
      return 0
  else:
      return values[0] + sum_rec(values[1:])
```

Platt lista som trädstruktur

- `[ 3]`
- **Problem**: Vad är summan av alla löv?
- Summan är värdet på detta löv, 3, + summan av övriga värden, vilket nu har beräknats till 0.

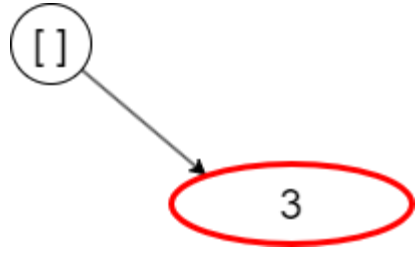



```
def sum_rec(values):
  if not values:
      return 0
  else:
      return values[0] + sum_rec(values[1:])
```

Platt lista som trädstruktur

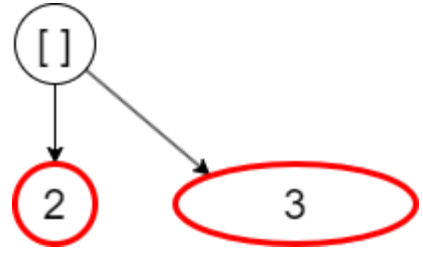
- `[ 3]`
- **Problem**: Vad är summan av alla löv?
- Returnera summan, dvs 3.

```
def sum_rec(values):
  if not values:
      return 0
  else:
      return values[0] + sum_rec(values[1:])
```



Platt lista som trädstruktur

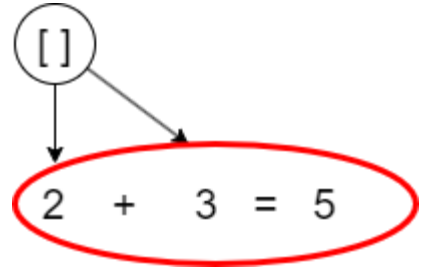
- `[ 2, 3]`
- **Problem**: Vad är summan av alla löv?
- Summan är värdet på detta löv, 2, + summan av övriga värden, vilket nu har beräknats till 3.




```
def sum_rec(values):
  if not values:
      return 0
  else:
      return values[0] + sum_rec(values[1:])
```

Platt lista som trädstruktur

- `[ 2, 3]`
- **Problem**: Vad är summan av alla löv?
- Summan är värdet på detta löv, 2, + summan av övriga värden, vilket nu har beräknats till 3.

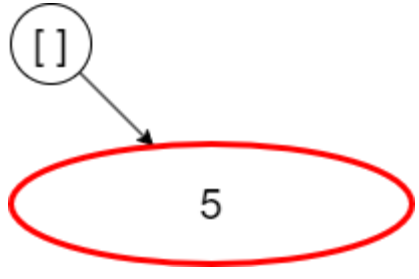



```
def sum_rec(values):
  if not values:
      return 0
  else:
      return values[0] + sum_rec(values[1:])
```

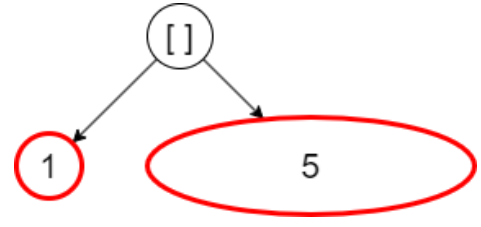
Platt lista som trädstruktur

- `[ 2, 3]`
- **Problem**: Vad är summan av alla löv?
- Returnera summan, dvs 5.

```
def sum_rec(values):
  if not values:
      return 0
  else:
      return values[0] + sum_rec(values[1:)
```



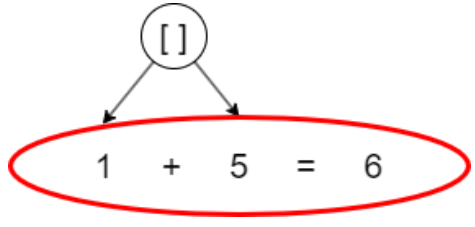
Platt lista som trädstruktur



- `[ 1, 2, 3]`
- **Problem**: Vad är summan av alla löv?
- Summan är värdet på detta löv, 1 + summan av övriga värden, vilket nu har beräknats till 5.

```
def sum_rec(values):
  if not values:
      return 0
  else:
      return values[0] + sum_rec(values[1:])
```

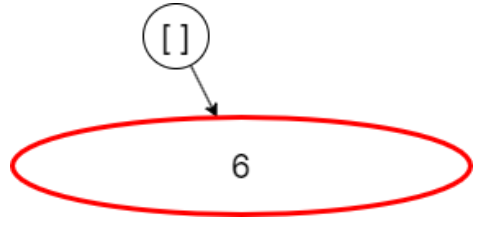
Platt lista som trädstruktur



- `[ 1, 2, 3]`
- **Problem**: Vad är summan av alla löv?
- Summan är värdet på detta löv, 1 + summan av övriga värden, vilket nu har beräknats till 5.

```
def sum_rec(values):
  if not values:
      return 0
  else:
      return values[0] + sum_rec(values[1:])
```

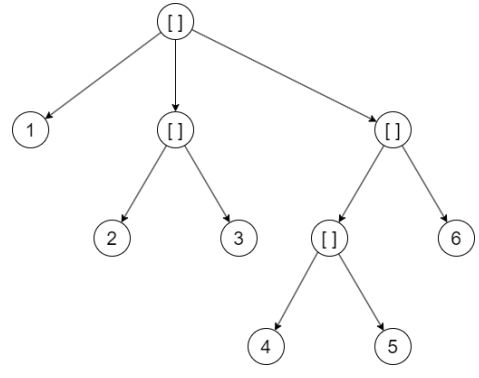
Platt lista som trädstruktur



- `[ 1, 2, 3]`
- **Problem**: Vad är summan av alla löv?
- Returnera summan, dvs 6.

```
def sum_rec(values):
  if not values:
      return 0
  else:
      return values[0] + sum_rec(values[1:])
```

Nästlade strukturer kan ses som trädstrukturer



- `[1, [2, 3], [ [4, 5], 6 ] ]`
- **Problem**: Vad är summan av alla löv?
- Om nod inte är ett löv:

  Noden är ett delträd, beräkna summan av delträdet.

  (Summan av ett delträd är beräknas på samma sätt som summan av ett träd.)

Nästlade strukturer kan ses som trädstrukturer

In [86]:
```
def sum_rec_nest(values):
    # om vi inte har några värden är summan 0
    if not values:
        return 0
     # om noden inte är ett värde, räkna ut delträdets värde och addera det ti
     # resten av värdena
    elif type(values[0]) == list:
        return sum_rec_nest(values[0]) + sum_rec_nest(values[1:])
     # noden är ett löv, addera dess värde till resten av värdena i trädet
    else:
        return values[0] + sum_rec_nest(values[1:])
```

In [87]:
```
sum_rec_nest([1, 2, 3])
```

Out[87]: 6

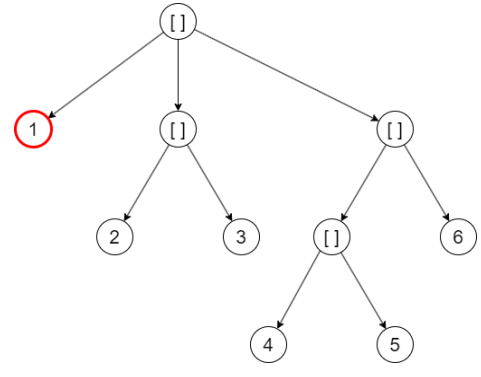
In [88]:
```
sum_rec_nest([1, [2, 3], [[4, 5], 6]])
```

Out[88]: 21

In [89]:
```
sum_rec_nest([1, [2, 3], [[4, 5, [6, 7, [8]]], 9]])
```

```
Out[89]: 45
```

Nästlade strukturer kan ses som trädstrukturer

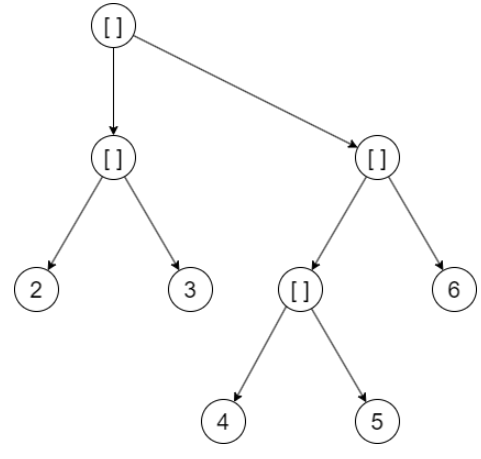


- `[1, [2, 3], [ [4, 5], 6 ] ]`
- **Problem**: Vad är summan av alla löv?
- Noden är ett löv: Beräkna 1 + summan av resten

  `sum_rec_nest` anropas på resten av listan:

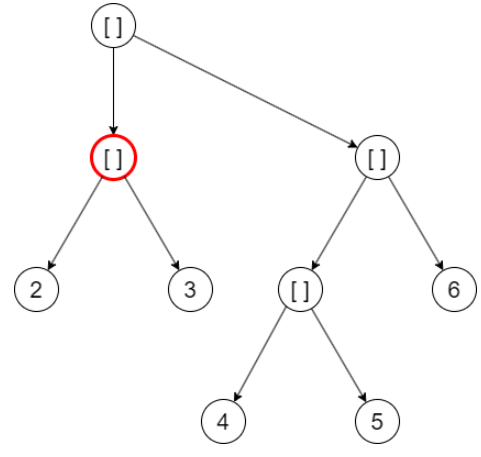
  `sum_rec_nest([ [2, 3], [ [4, 5], 6 ] ])`

Nästlade strukturer kan ses som trädstrukturer



- `[ [2, 3], [ [4, 5], 6 ] ]`
- **Problem**: Vad är summan av alla löv?

Nästlade strukturer kan ses som trädstrukturer



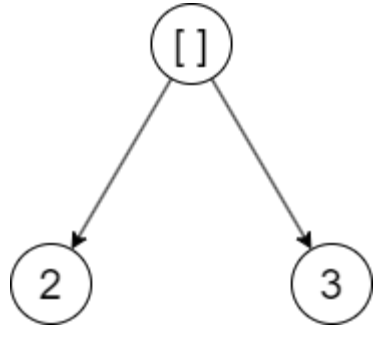
- `[ [2, 3], [ [4, 5], 6 ] ]`
- **Problem**: Vad är summan av alla löv?
- Noden är ett delträd: Beräkna summan av delträdet + summan av resten

  `sum_rec_nest` anropas på första noden:

  `sum_rec_nest([2, 3])`

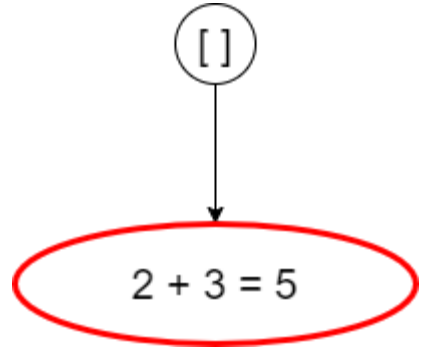
Nästlade strukturer kan ses som trädstrukturer

- `[2, 3]`
- **Problem**: Vad är summan av alla löv?



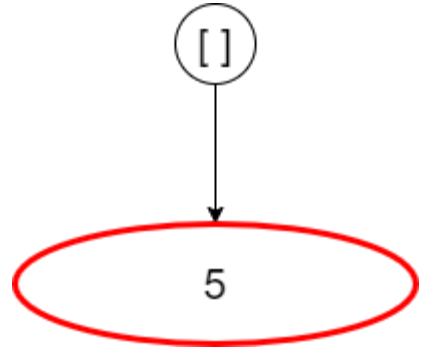
Nästlade strukturer kan ses som trädstrukturer

- `[ [2, 3] ]`
- **Problem**: Vad är summan av alla löv?
- Beräkna summan av delträdet.

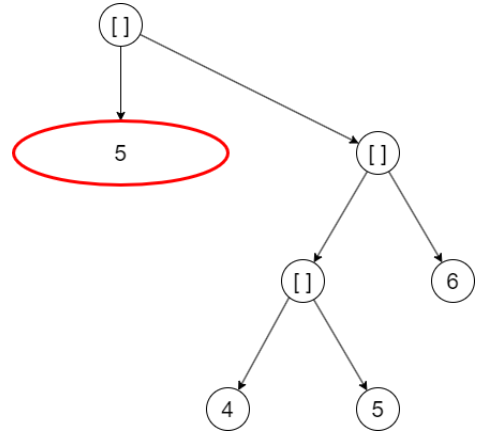


Nästlade strukturer kan ses som trädstrukturer

- `[ [2, 3] ]`
- **Problem**: Vad är summan av alla löv?
- Returnera summan, dvs 5.



Nästlade strukturer kan ses som trädstrukturer

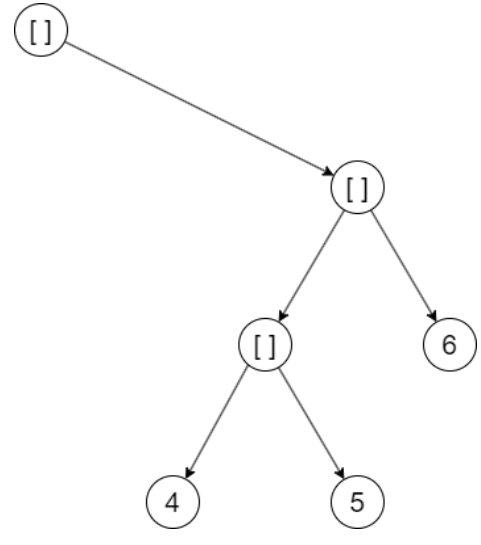


- `[ [2, 3], [ [4, 5], 6 ] ]`
- **Problem**: Vad är summan av alla löv?
- Summan av första noden är 5: Beräkna 5 + summan av resten

  `sum_rec_nest` anropas på resten av listan:

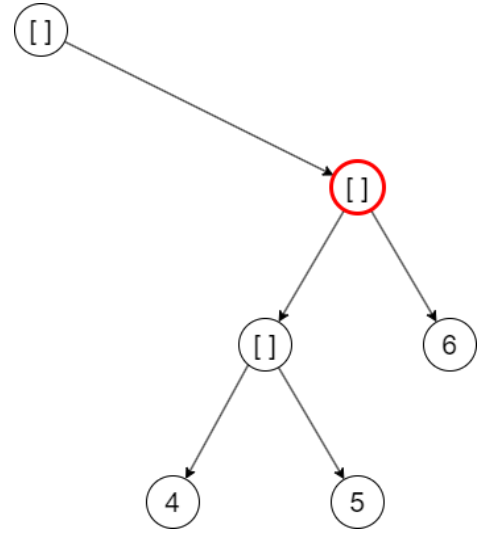
  `sum_rec_nest([ [ [4, 5], 6 ] ])`

Nästlade strukturer kan ses som trädstrukturer



- `[ [ [4, 5], 6 ] ]`
- **Problem**: Vad är summan av alla löv?

Nästlade strukturer kan ses som trädstrukturer

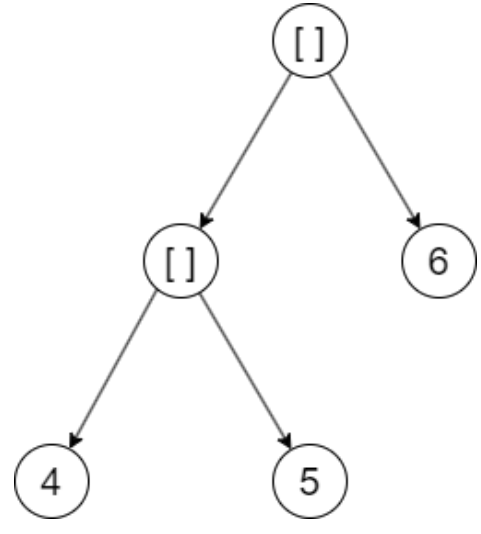


- `[ [ [4, 5], 6 ] ]`
- **Problem**: Vad är summan av alla löv?
- Noden är ett delträd: Beräkna summan av delträdet + summan av resten

  `sum_rec_nest` anropas på första noden:

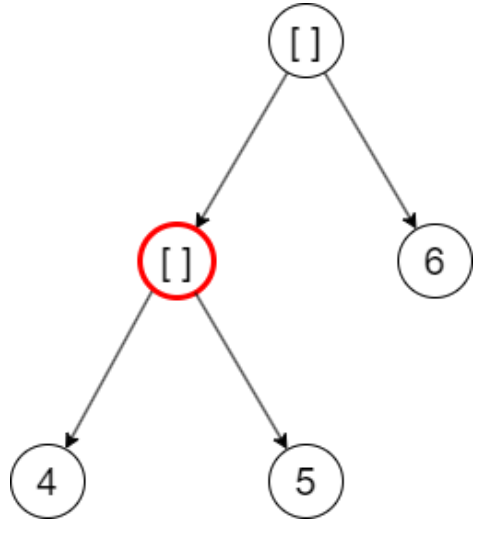
  `sum_rec_nest([ [4, 5], 6 ])`

Nästlade strukturer kan ses som trädstrukturer



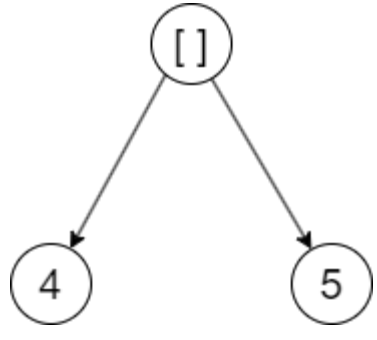
- `[ [4, 5], 6 ]`
- **Problem**: Vad är summan av alla löv?

Nästlade strukturer kan ses som trädstrukturer



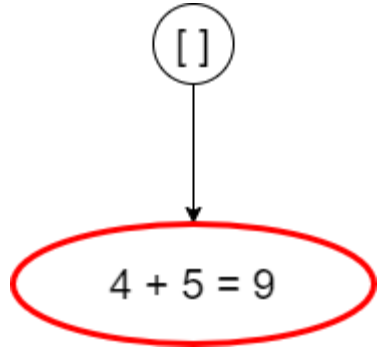
Nästlade strukturer kan ses som trädstrukturer

- `[4, 5]`
- **Problem**: Vad är summan av alla löv?

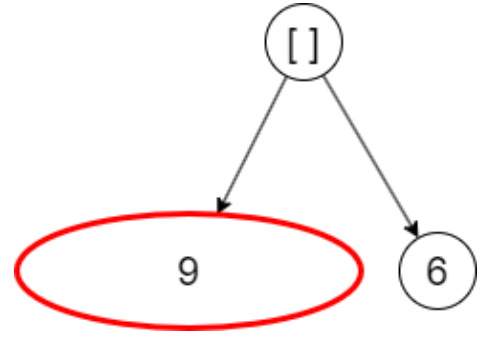


Nästlade strukturer kan ses som trädstrukturer

- `[4, 5]`
- **Problem**: Vad är summan av alla löv?
- Returnera summan, dvs 9



Nästlade strukturer kan ses som trädstrukturer



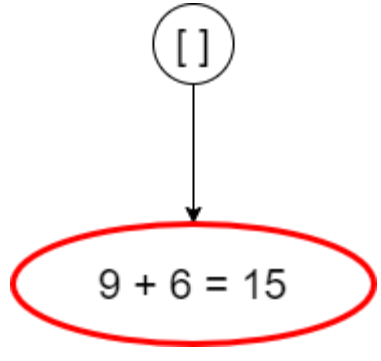
- `[ [4, 5], 6 ]`
- **Problem**: Vad är summan av alla löv?
- Summan av första noden är 9: Beräkna 9 + summan av resten

  `sum_rec_nest` anropas på resten av listan:

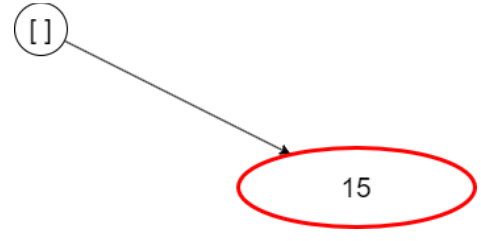
  `sum_rec_nest([ 6 ])` → 6

Nästlade strukturer kan ses som trädstrukturer

- `[ [4, 5], 6 ]`
- **Problem**: Vad är summan av alla löv?
- Returnera summan, dvs 15

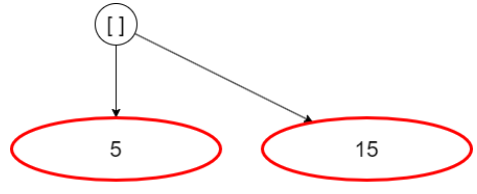


Nästlade strukturer kan ses som trädstrukturer



- `[ [ [4, 5], 6 ] ]`
- **Problem**: Vad är summan av alla löv?
- Summan har beräknats till 15
- Returnera summan, dvs 15

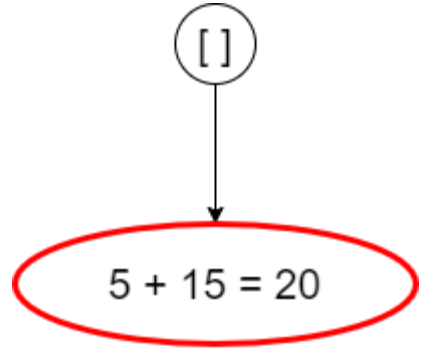
Nästlade strukturer kan ses som trädstrukturer



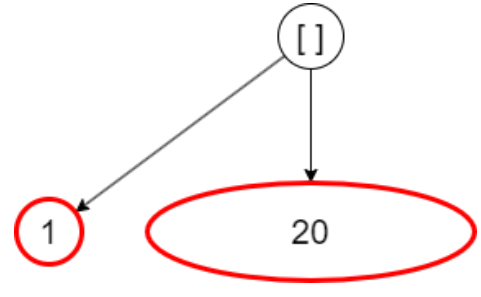
- `[ [2, 3], [ [4, 5], 6 ] ]`
- **Problem**: Vad är summan av alla löv?
- Summan av första noden är beräknad till 5
- Summan av resten är beräknad till 15

Nästlade strukturer kan ses som trädstrukturer

- `[ [2, 3], [ [4, 5], 6 ] ]`
- **Problem**: Vad är summan av alla löv?
- Returnera summan, dvs 5 + 15



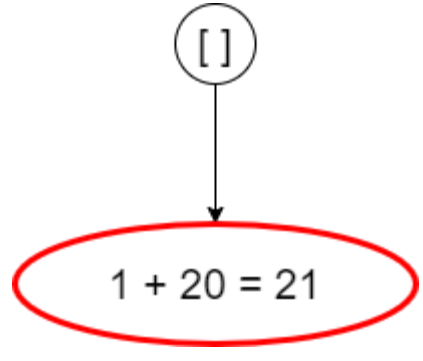
Nästlade strukturer kan ses som trädstrukturer



- `[1, [2, 3], [ [4, 5], 6 ] ]`
- **Problem**: Vad är summan av alla löv?
- Summan av resten har beräknats till 20

Nästlade strukturer kan ses som trädstrukturer

- `[1, [2, 3], [ [4, 5], 6 ] ]`
- **Problem**: Vad är summan av alla löv?
- Returnera summan, dvs 21



Leta efter ett värde i en nästlad lista

In [90]:
```
def look_for_value_rec_all(value, values):
    # Inget värde kan finnas i en tom lista
    if not values:
        return False

    # om första värdet inte är en lista, kolla om det är det värde
    # vi letar efter, returnera i så fall True
    elif values[0] == value:
        return True

    # om första värdet är en lista, returnera resultatet av att
    # leta i både den listan och resten av values
    elif type(values[0]) == list:
        return (look_for_value_rec_all(value, values[0]) or
                look_for_value_rec_all(value, values[1:]))

    # om inte första värdet varken var en lista eller det vi letade efter
    # returnera resultatet av att leta efter värdet i resten av listan
    else:
        return look_for_value_rec_all(value, values[1:])
```

Leta efter ett värde i en nästlad lista

In [91]:
```
numbers1 = [[1, 2, 75, 6, 7], [75, 6, 7], [75, 6, 7],
            [1, 2, 75, 6, 7], [75, 6, 7], [73]]
numbers2 = [[1, 2], 75, 6, 7, [75, 6, 7], 75, 6, 7, [1, 2, 75, 6, 7],
            75, 6, 7, 73]
numbers3 = [[1, 2], [75, [6, 7]], [75, [6, 7]], [75, [6, 7]],
            [1], 2, [75, [6, [7, [75], 6], 7]], 73, []]
```

In [92]:
```
look_for_value_rec_all(7, numbers1)
```

Out[92]: True

In [93]:
```
look_for_value_rec_all(73, numbers1)
```

Out[93]: True

In [94]:
```
look_for_value_rec_all([73], numbers1)
```

Out[94]: True

In [95]:
```
look_for_value_rec_all([73], numbers2)
```

Out[95]: False

Spara enbart strängar i en nästlad lista, men bevara strukturen på listan

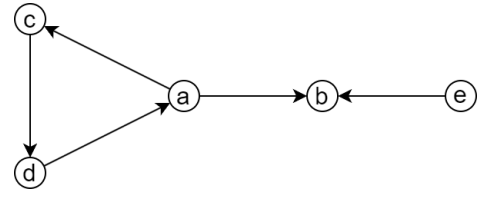
In [96]:
```python
def keep_strings_rec_all(values):
    # om listan är tom så resultatet en tom lista
    if not values:
        return []

    # om första värdet i listan är en sträng så är resultatet
    # en lista med den strängen + alla strängar i resten av values
    elif type(values[0]) == str:
        return [values[0]] + keep_strings_rec_all(values[1:])

    # om första värdet är en lista är resultatet en lista med listan
    # utan några andra datatyper än strängar + bearbetningen av resten
    # av listan
    elif type(values[0]) == list:
        return [keep_strings_rec_all(values[0])] + \
               keep_strings_rec_all(values[1:])
    # om första värdet i listan inte var en sträng så är resultatet
    # alla strängar i resten av listan
    else:
        return keep_strings_rec_all(values[1:])
```

In [97]:
```python
print(keep_strings_rec_all(["ett", [2, "sjuttiofem", 6], 7, [[75], [6, ["sju"]
```

```
['ett', ['sjuttiofem'], [[], [['sju']]]]
```

Tillämpning av rekursion: grafsökning

Rekursiva funktioner kan "utforska" flera olika lösningsvägar med trädrekursion



- **Exempel**: En karta med enkelriktade vägar representeras som en graf.
- **Problem**: Givet en startpunkt A kan vi ta oss till platsen B? Returnera vägen.
- **Representation**: Vi representerar grafen som ett dictionary där nycklarna är noder vars värde är en lista med de noder vi kan nå.

```
map1 = {"a": ["b", "c"],
"b": [],
"c": ["d"],
"d": ["a"],
"e": ["b"]}
```

Hitta vägen

```
def get_path(s_node, e_node, a_map, visited):
    """Returnera en lista med vägen från s_node till e_node om sådan
finns."""
    # är e_node direkt tillgänglig?
    if e_node in a_map[s_node]:
        return visited + [s_node, e_node]
    # kolla om vi kan ta oss till e_node från någon av grannarna till
s_node
    return get_path_hlp(a_map[s_node], e_node, a_map, visited +
[s_node])

def get_path_hlp(s_nodes, e_node, a_map, visited):
    """Returnera den första vägen från en nod i s_nodes till e_node om
sådan finns."""
    # om s_nodes är tom så kan vi inte ta oss till e_node
    if not s_nodes:
        return []
    # om vi inte redan besökt s_nodes[0]
    elif s_nodes[0] not in visited:
        # kolla om vi kan ta oss till e_node därifrån eller någon av de
övriga
        # nodern i s_nodes
        path = get_path(s_nodes[0], e_node, a_map, visited)
        if path:
            return path
        else:
```

```
                return get_path_hlp(s_nodes[1:], e_node, a_map, visited)
            # om vi redan besökt s_nodes[0]
            else:
                # kolla om vi kan ta oss till e_node från någon av de övriga
        noderna
                # i s_nodes
                return get_path_hlp(s_nodes[2:], e_node, a_map, visited)
```

In [104]:
```
print(get_path('c', 'b', map1, []))
```

```
['c', 'd', 'a', 'b']
```

Hitta vägen (med for-loop)

Ibland är den bästa lösningen att kombinera rekursion med vanliga loop-konstruktioner.

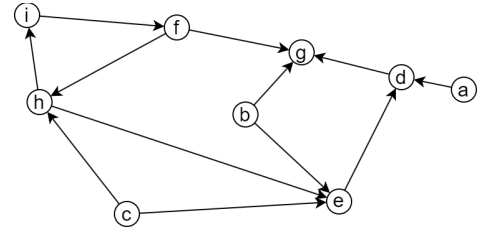
In [105]:
```
def get_path_with_for(s_node, e_node, a_map, visited):
    """Returnera en lista med vägen från s_node till e_node om sådan finns."""
    # är e_node direkt tillgänglig?
    if e_node in a_map[s_node]:
        return visited + [s_node, e_node]
    # kolla om vi kan ta oss till e_node från någon av grannarna till s_node
    for next_node in a_map[s_node]:
        if next_node not in visited:
            path = get_path_with_for(next_node, e_node, a_map, visited + [s_no
            if path:
                return path
    # om vi kommer hit kunde vi inte hitta någon väg
    return []
```

In [106]:
```
print(get_path_with_for('c', 'b', map1, []))
```

```
['c', 'd', 'a', 'b']
```

Större karta

```
map2 = {"a": ["d"],
        "b": ["e", "g"],
        "c": ["e", "h"],
        "d": ["g"],
        "e": ["d"],
        "f": ["g", "h"],
        "g": [],
        "h": ["e", "i"],
        "i": ["f"]}
```



In [107]:
```
print(get_path_with_for('h', 'g', map2, []))
```

```
['h', 'e', 'd', 'g']
```