# 729G46 Informationsteknologi och programmering

## Tema 2, Föreläsning 2.1-2.2

Johan Falkenjack, johan.falkenjack@liu.se

# Föreläsningsöversikt, FÖ 2.1-2.2

- Vad har vi lärt oss hittills?
- Tema 2: Informationsbearbetning
- Sekventiell information; sekvenser av värden
- Python: Vad behöver vi för att bearbeta sekventiell information?
    - sanningsvärden, villkor, loopar
- Funktioner och scope; lokala och globala variabler
- Felsökning och olika typer av fel
- Testning
- Programabstraktion
- Läsa från textfiler
- Några sträng- och listmetoder

# Vad har vi lärt oss under Tema 1?

# Tema 1

- Datorn och program
- Beståndsdelar i ett program:
    - värden, datatyper: int, float, str, list
    - variabler, operatorer, uttryck
    - satser, nyckelord
    - funktioner, argument, returvärden
    - block
- Diskret matematik
    - mängder

# Repetition: return eller print

- return
    - En sats som styr programflödet
    - Bara i funktioner
    - Skickar ett värde *tillbaka* till koden som anropat funktionen
    - Kan finnas på flera platser i en funktion men kan bara köras en gång och är alltid det sista som händer i funktionen
    - Inga parenteser
- print
    - En funktion som skriver ut ett värde
    - Kan göras hur många gånger som helst och nästan var som helst
    - Påverkar inte programflödet
    - Parenteser används för att anropa funktionen

# print

In [2]:
```
def add_1_and_print(value):
    print(value+1)

add_1_and_print(1)
```

```
2
```

# return

In [3]:
```
def add_1_and_return(value):
    return value+1
```

# return

In [3]:
```
def add_1_and_return(value):
    return value+1
```

In [4]:
```
print(add_1_and_return(1))
```

```
2
```

# return

In [3]:
```
def add_1_and_return(value):
    return value+1
```

In [4]:
```
print(add_1_and_return(1))
```

```
2
```

In [5]:
```
print(add_1_and_return(add_1_and_return(1)))
```

```
3
```

# print

In [6]:
```
def add_1_and_print(value):
    print(value+1)

add_1_and_print(1)
```

```
2
```

# print

In [6]:
```
def add_1_and_print(value):
    print(value+1)

add_1_and_print(1)
```

```
2
```

In [7]:
```
add_1_and_print(add_1_and_print(1))
```

```
2
```

```
---------------------------------------------------------------------
------
TypeError                                 Traceback (most recent call
last)
Cell In[7], line 1
----> 1 add_1_and_print(add_1_and_print(1))

Cell In[6], line 2, in add_1_and_print(value)
      1 def add_1_and_print(value):
----> 2     print(value+1)

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

# Tema 2
# Informationsbearbetning

# Hur representerar vi information i en dator?

- **Representation**
- **Modeller**
- **Datatyper**: heltal, flyttal, strängar, listor

# Hur representerar vi information i en dator?

- **Representation**
- **Modeller**
- **Datatyper**: heltal, flyttal, strängar, listor

- Ordböcker? Bilder? Filmer? Kontaktuppgifter? Ljud? Lukter? Släktingar? Ansiktsuttryck? Kunskap? Känslor? Känslan av att springa? Ordet vi nästan kan? De saker som finns på mitt skrivbord? Associationer?

# Information och informationsbearbetning

- Hur får vi "in" information?
- Hur ser information ut?
- Hur bearbetar vi information?
- Hur beskriver vi i ett program hur information ska bearbetas?
- Vad behöver vi kunna uttrycka med vårt programmerinsspråk?

# Information / data

- Enstaka värden: 0, 1
- I de flesta fall behöver vi representera mer "komplex" information. Hur?
- Ökar abstraktionsnivån genom att bygga med underliggande strukturer
    - { 0, 1 } -> tal -> tecken
    - Sekvenser av värden: strängar, listor
    - Sekvenser av sekvenser av värden: ord -> meningar -> dokument ...

# Tema 2

- Lagring, användning/bearbetning av värden i sekvens.
- Lagring
    - Python: datatyper
    - Begreppssem 2: filer, filformat, komprimering, filsystem
- Användning och bearbetning
    - styra flöde i program
    - repetition i program
- Diskret matematik
    - tupler, grafer, relationer

# Sekvenser av värden

När behövs en sekvens av värden?

Vad kan representeras som en sekvens av värden?

# Enkla värden

- Vad kan representeras som ett enkelt värde?
- T.ex. heltal, flyttal, strängar (om vi ser på en sträng som ett enstaka värde)

# Vad är en sekvens av värden

- Ordning
    - $A$ kommer före $B$
    - $B$ kommer före $C$
    - $C$ kommer före $D$
    - ...

# Vad är en sekvens av värden

- Ordning
    - $A$ kommer före $B$
    - $B$ kommer före $C$
    - $C$ kommer före $D$
    - ...

- Finns det scenarion där vi har
    - en ändlig sekvens av värden?
    - en oändlig sekvens av värden?

# Sekventiell information och datorn

- Minnesplatser i RAM
- Värden lagrade i filer
- Värden från olika sensorer
- Inlägg i ett flöde i sociala medier

# Från bits till tal, till tecken, till strängar

- Åtta bitar (binärt): `0b11111111`, `0b00001111`, `0b01010101`
- Hexadecimalt: `0xff`, `0x0f`, `0x55`
- Decimalt: `255`, `15`, `85`
- Teckentabeller: `65 = 'A'`, `97 = 'a'`
- Strängen `"Aa"` = `65 97 0` (decimalt), eller `0x41 0x61 0x00` (hexadecimalt)
    - Värdet 0 används för att visa var strängen slutar (t.ex. om man läser strängenfrån minnet)

# Sekvenser av värden

Lagring av data i en fil

# Filer och filformat

# Filer och filformat

- Fil
    - sekvens av värden (bytes) som hör ihop och lagras i ett filsystem.
    - en byte kan användas för att representera ett värde mellan 0-255 ( `0x00-0xFF` , `0b00000000-0b11111111` )

# Filer och filformat

- Fil
    - sekvens av värden (bytes) som hör ihop och lagras i ett filsystem.
    - en byte kan användas för att representera ett värde mellan 0-255 ( `0x00-0xFF` , `0b00000000-0b11111111` )

- Filformat
    - standard för hur information ska lagras (kodas) i en fil
    - specifikation
    - öppna kontra stängda filformat

# Filer och filformat

# Filer och filformat

- **Textbaserade filformat**
    - **Fördel**: vi kan läsa och redigera informationen med en texteditor
    - **Nackdel**: kan ta mycket mer plats än nödvändigt
    - **Exempel**: csv, tsv, xml, html, css, json

# Filer och filformat

- **Textbaserade filformat**
    - **Fördel**: vi kan läsa och redigera informationen med en texteditor
    - **Nackdel**: kan ta mycket mer plats än nödvändigt
    - **Exempel**: csv, tsv, xml, html, css, json
- **Binära filformat**
    - **Fördel**: tar mindre plats, snabbare/effektivare
    - **Nackdel**: vi behöver använda speciell mjukvara för att läsa/ändra
    - **Exempel**: jpg, png, gif, mp3, mov, m4v, pdf, doc

# Filer och filformat

- **Textbaserade filformat**
    - **Fördel**: vi kan läsa och redigera informationen med en texteditor
    - **Nackdel**: kan ta mycket mer plats än nödvändigt
    - **Exempel**: csv, tsv, xml, html, css, json
- **Binära filformat**
    - **Fördel**: tar mindre plats, snabbare/effektivare
    - **Nackdel**: vi behöver använda speciell mjukvara för att läsa/ändra
    - **Exempel**: jpg, png, gif, mp3, mov, m4v, pdf, doc
- **Mixade filformat**
    - **Exempel**: pdf, docx, xlsx

# Textbaserade filformat

# Textfiler

- Filer där varje enhet data/värde ska tolkas som ett tecken
- Olika teckentabeller och teckenkodningar
    - teckentabell: olika "skiffer", t.ex. "A" = 65 (i ASCII)
    - teckenkodning: olika sätt att lagra teckenvärdena

# ASCII

- American Standard Code for Information Interchange (1963)
- Standard på det tidiga internet och fortfarande den mest allmängiltiga teckenkodningen
- 7 bitar
    - (Lagras dock idag alltid i en byte, dvs 8 bitar, med 0 i första positionen)
- $2^7 = 128$ möjliga tecken, alltså väldigt begränsat
    - T.ex. saknas å, ä, och ö

# ASCII-tabellen

| | _0 | _1 | _2 | _3 | _4 | _5 | _6 | _7 | _8 | _9 | _A | _B | _C | _D | _E | _F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0_ 0 | NUL 0000 | SOH 0001 | STX 0002 | ETX 0003 | EOT 0004 | ENQ 0005 | ACK 0006 | BEL 0007 | BS 0008 | HT 0009 | LF 000A | VT 000B | FF 000C | CR 000D | SO 000E | SI 000F |
| 1_ 16 | DLE 0010 | DC1 0011 | DC2 0012 | DC3 0013 | DC4 0014 | NAK 0015 | SYN 0016 | ETB 0017 | CAN 0018 | EM 0019 | SUB 001A | ESC 001B | FS 001C | GS 001D | RS 001E | US 001F |
| 2_ 32 | SP 0020 | ! 0021 | " 0022 | # 0023 | $ 0024 | % 0025 | & 0026 | ' 0027 | ( 0028 | ) 0029 | * 002A | + 002B | , 002C | - 002D | . 002E | / 002F |
| 3_ 48 | 0 0030 | 1 0031 | 2 0032 | 3 0033 | 4 0034 | 5 0035 | 6 0036 | 7 0037 | 8 0038 | 9 0039 | : 003A | ; 003B | < 003C | = 003D | > 003E | ? 003F |
| 4_ 64 | @ 0040 | A 0041 | B 0042 | C 0043 | D 0044 | E 0045 | F 0046 | G 0047 | H 0048 | I 0049 | J 004A | K 004B | L 004C | M 004D | N 004E | O 004F |
| 5_ 80 | P 0050 | Q 0051 | R 0052 | S 0053 | T 0054 | U 0055 | V 0056 | W 0057 | X 0058 | Y 0059 | Z 005A | [ 005B | \ 005C | ] 005D | ^ 005E | _ 005F |
| 6_ 96 | ` 0060 | a 0061 | b 0062 | c 0063 | d 0064 | e 0065 | f 0066 | g 0067 | h 0068 | i 0069 | j 006A | k 006B | l 006C | m 006D | n 006E | o 006F |
| 7_ 112 | p 0070 | q 0071 | r 0072 | s 0073 | t 0074 | u 0075 | v 0076 | w 0077 | x 0078 | y 0079 | z 007A | { 007B | \| 007C | } 007D | ~ 007E | DEL 007F |

# ASCII och radbrytningar

- Radbrytningar i ASCII designade efter skrivmaskiner:
    - LF (`000 1010`): ”Line Feed”, gå till nästa rad
    - CR (`000 1101`): ”Carriage Return”, återvänd till början av raden
- Två tecken för en ”vanlig radbrytning”!
    - CR LF: Windows, MS-DOS
- Lagringsutrymme var dyrt och begränsat. Lösning: Välj ett. Men vilket?
    - LF: Unix (och Unix-lika system som macOS, GNU/Linux, Android), Amiga
    - CR: Commodore64, Apple II, Mac OS
- Orsakar fortfarande problem

# Arvet efter ASCII

- Olika länder skapade egna versioner av ASCII
    - ISO/IEC 646 beskriver olika nationella ASCII-varianter
- 8 bitar blev standard, ASCII slösade bort 1 hel bit, dvs $2^8 - 2^7 = 128$ möjliga tecken.

# Arvet efter ASCII

- Olika länder skapade egna versioner av ASCII
    - ISO/IEC 646 beskriver olika nationella ASCII-varianter
- 8 bitar blev standard, ASCII slösade bort 1 hel bit, dvs $2^8 - 2^7 = 128$ möjliga tecken.
- Snilleblixt: Använd alla 8 bitar och kalla det "utökad ASCII"! Tecken 0-127 samma som ASCII, men sen då?
    - Adobe: PostScript Standard Encoding
    - Microsoft: CP1252
    - Apple: MacRoman
    - Etc…
- Radbrytningskaoset all over again.

# Unicode

- **Teckentabell** med plats (tekniskt sett) för 1 114 112 olika tecken
    - Version 1.0.0 (1991): 7 163 tecken
    - Version 3.1 (2001): 94 140 tecken
    - Version 6.0 (2010): 110 116 tecken
    - Version 15.1 (2023): 149 813 tecken (publiceras idag, 12 september 2023)
- **Teckenkodningar**: UTF-8, UTF-16 m.fl.
- Diverse skriftsystem
    - latinska bokstäver, kyrilliska, punktskrift, kinesiska, arabiska, runskrift, kilskrift, hieroglyfer, m. fl.
- Diverse andra tecken/glyfer
    - emoji, pilar, matematiska symboler, valutor, noter

# Tabell-data, CSV

- "comma separated values"
    - ibland används något annat tecken än kommatecken, ofta kolon eller tab (ofta kallat tsv istället)
- Rader av värden
- Varje värde på raden skiljs åt av ett kommatecken (eller något annat tecken)
- Varje rad måste ha lika många värden

# CSV-exempel

```
"Sex","Weight (Sep)","Weight (Apr)","BMI (Sep)","BMI (Apr)"
"F",64,60,24.24,22.88
"F",56,53,21.23,20.23
"M",72,59,22.02,18.14
"M",97,86,19.7,17.44
"M",93,88,26.97,25.57
"F",68,64,21.51,20.1
"M",59,55,18.69,17.4
```

# XML - Extensible Markup Language

- Ett *uppmärkningsspråk* som används för att strukturera data.
- Textbaserat, dvs information lagras som text.
- DTD (Document Type Definition)/Schema specificerar strukturen separat från dokumentet som innehåller data
- Kan användas för många olika typer av information.
- Hierarkisk struktur
- **SVG**: XML-baserat grafik-format
- **HTML**: XML-liknande dokumentstrukturformat

# XML-exempel

```
<book>
    <author>Ludwig Wittgenstein</author>
    <title>Tractatus logico-philosophicus</title>
    <year>1921</year>
</book>
```

# XML-exempel

```
<book>
    <author>Ludwig Wittgenstein</author>
    <title>Tractatus logico-philosophicus</title>
    <year>1921</year>
    <contents>
        <chapter>
            <title>Inledning</title>
            <section>
                <title>1. Wittgenstein och Tractatus</title>
                <paragraph>
                ...
                </paragraph>
            </section>
        </chapter>
    </contents>
</book>
```

# Binära filformat

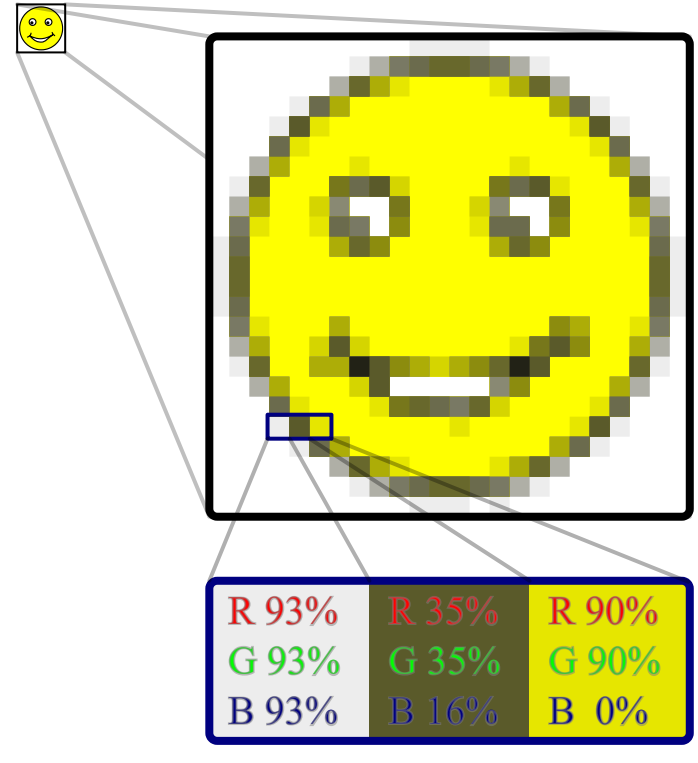
Värden behöver inte kunna tolkas som tecken

# Binära filformat

- Ofta förekommande struktur
    - *Header* med metadata (data om data, t.ex. beskrivningar av format) lagrat som text följt av data lagrad binärt
- Specifikt användningområde
    - Många format kan endast användas för att koda en specifik typ av information (t.ex. bild, ljud, presentation etc)
- Exempel på undantag (som kan lagra flera olika filer)
    - (komprimerade) arkiv-format som zip, rar, 7z, etc.

# Exempel: Bild

- Varje pixel kan t.ex. representeras som ett 24-bitars värde
    - Tre bytes behövs för varje pixel
    - 8 bitar för röd, 8 bitar för grön, 8 bitar för blå



# Sekvenser av värden

I Python

# Sekvenser i Python.

Exempel: strängar

- En sträng är en sekvens av tecken.

# Sekvenser i Python.

Exempel: strängar

- En sträng är en sekvens av tecken.

In [8]:
```
mitt_namn = "Pikachu"
mitt_namn[0]
```

Out[8]:
```
'P'
```

# Sekvenser i Python.

Exempel: strängar

- En sträng är en sekvens av tecken.

In [8]:
```
mitt_namn = "Pikachu"
mitt_namn[0]
```

Out[8]:
```
'P'
```

In [9]:
```
mitt_namn[1:3]
```

Out[9]:
```
'ik'
```

# Sekvenser i Python.

Exempel: strängar

- En sträng är en sekvens av tecken.

In [8]:
```
mitt_namn = "Pikachu"
mitt_namn[0]
```

Out[8]:
```
'P'
```

In [9]:
```
mitt_namn[1:3]
```

Out[9]:
```
'ik'
```

In [10]:
```
"Mr. " + mitt_namn
```

Out[10]:
```
'Mr. Pikachu'
```

# Sekvenser i Python.

Exempel: strängar

- En sträng är en sekvens av tecken.

In [8]:
```
mitt_namn = "Pikachu"
mitt_namn[0]
```

Out[8]:
```
'P'
```

In [9]:
```
mitt_namn[1:3]
```

Out[9]:
```
'ik'
```

In [10]:
```
"Mr. " + mitt_namn
```

Out[10]:
```
'Mr. Pikachu'
```

In [11]:
```
mitt_namn + "Mr. "
```

Out[11]:
```
'PikachuMr. '
```

# Sekvenser i Python.

Exempel: listor

- **En lista är sekvens av värden** (t.ex. heltal, flyttal, strängar).
- **En lista är också ett värde**, dvs en lista kan också innehålla andra listor.

# Sekvenser i Python.

Exempel: listor

- **En lista är sekvens av värden** (t.ex. heltal, flyttal, strängar).
- **En lista är också ett värde**, dvs en lista kan också innehålla andra listor.

In [12]:
```
godis = ["söt", "grön", "mjuk", "kletig"]
godis[0]
```

Out[12]:
```
'söt'
```

# Sekvenser i Python.

Exempel: listor

- **En lista är sekvens av värden** (t.ex. heltal, flyttal, strängar).
- **En lista är också ett värde**, dvs en lista kan också innehålla andra listor.

In [12]:
```
godis = ["söt", "grön", "mjuk", "kletig"]
godis[0]
```

Out[12]:
```
'söt'
```

In [13]:
```
godis[1:3]
```

Out[13]:
```
['grön', 'mjuk']
```

# Sekvenser i Python.

Exempel: listor

- **En lista är sekvens av värden** (t.ex. heltal, flyttal, strängar).
- **En lista är också ett värde**, dvs en lista kan också innehålla andra listor.

In [12]:
```
godis = ["söt", "grön", "mjuk", "kletig"]
godis[0]
```

Out[12]:
```
'söt'
```

In [13]:
```
godis[1:3]
```

Out[13]:
```
['grön', 'mjuk']
```

In [14]:
```
["god"] + godis
```

Out[14]:
```
['god', 'söt', 'grön', 'mjuk', 'kletig']
```

In [15]:
```
godis + ["god"]
```

Out[15]:
```
['söt', 'grön', 'mjuk', 'kletig', 'god']
```

# Varför börjar vi räkna från 0?

- "Så har det alltid varit!"
    - Men varför?
- Sekvenser lagrades i så kallade "fält" (eng: **array**) i minnet, dvs i direkt följd från en viss minnesadress
    - Minnesadressen pekade på början av första elementet i sekvensen
    - För att komma åt ett element på en viss position i sekvensen användes **index** som indikerade hur många steg från sekvensens start man behövde gå för att komma till elementet man ville ha
    - Första elementet låg alltså på sekvensens minnesadress + 0

# Varför börjar vi räkna från 0?

- "Så har det alltid varit!"
    - Men varför?
- Sekvenser lagrades i så kallade "fält" (eng: **array**) i minnet, dvs i direkt följd från en viss minnesadress
    - Minnesadressen pekade på början av första elementet i sekvensen
    - För att komma åt ett element på en viss position i sekvensen användes **index** som indikerade hur många steg från sekvensens start man behövde gå för att komma till elementet man ville ha
    - Första elementet låg alltså på sekvensens minnesadress + 0
- Enkelt trick: **Tänk på index som platserna mellan elementen**

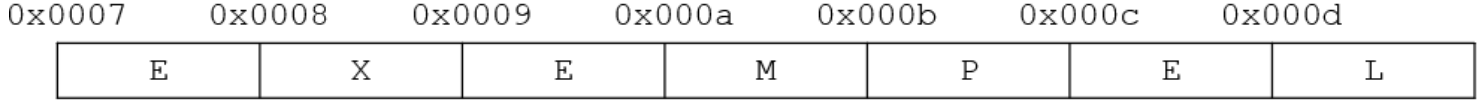
# Exempel

# Exempel

In [16]:
```
spam = "EXEMPEL"
```

# Exempel

In [16]:
```
spam = "EXEMPEL"
```




- Vi skapar en sträng `"EXEMPEL"` och sparar den i variabeln `spam`
- Vi antar att `spam` lagras på minnesplats `0x0007`
    - *(normalt sett behöver vi inte bry oss om faktiska minnesadresser i Python och på moderna datorer är minnesadresserna mycket längre)*

# Exempel

# Exempel

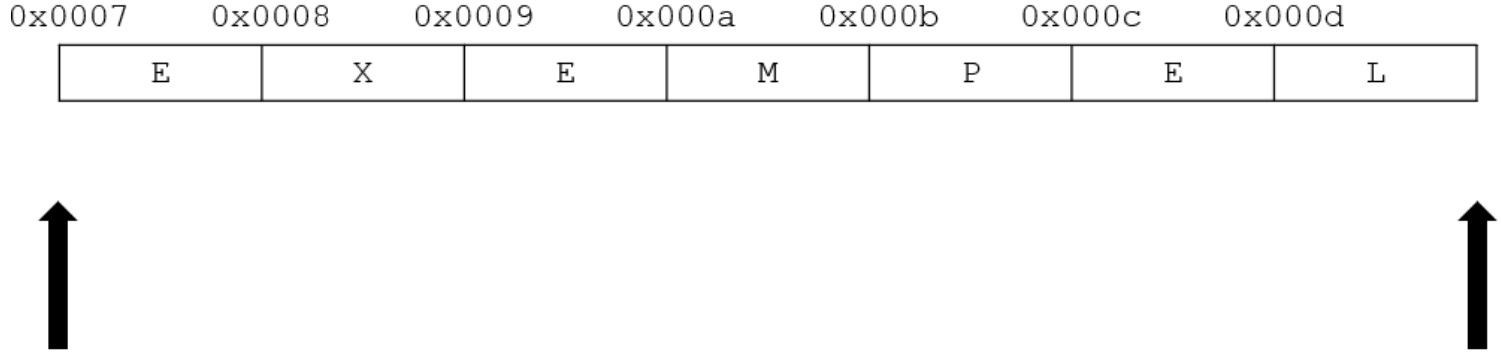
In [17]:
```
spam
```

Out[17]:
```
'EXEMPEL'
```

# Exempel

In [17]:
```
spam
```

Out[17]:
```
'EXEMPEL'
```

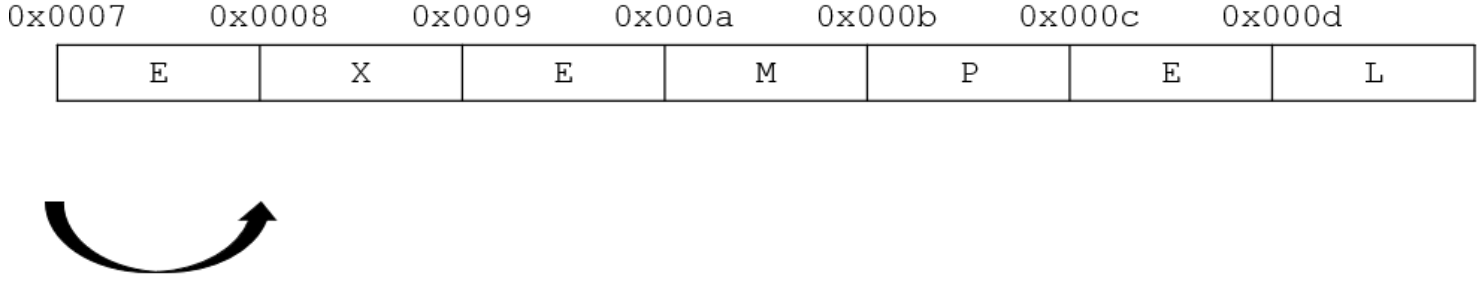



- Om vi *avrefererar* (använder) variabeln `spam` får vi ut sekvensen som börjar på minnesadressen `0x0007` och är 7 tecken lång

# Exempel

In [18]:
```
spam[0]
```

Out[18]:
```
'E'
```

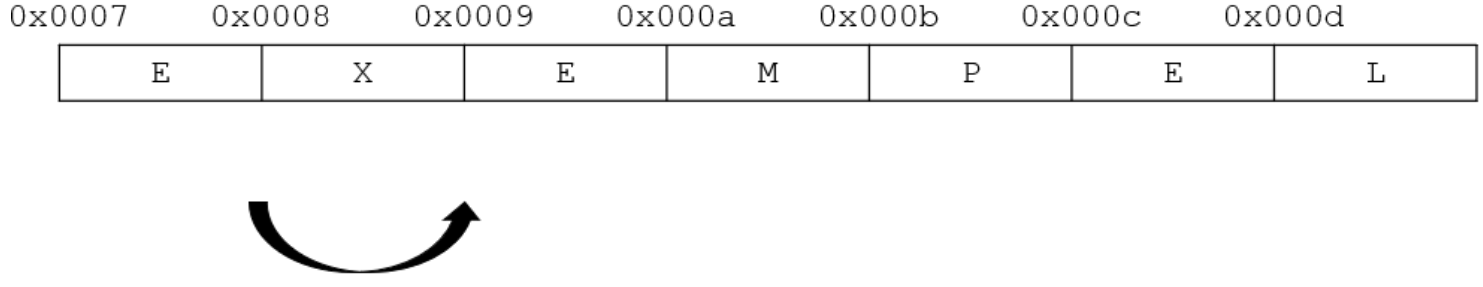



- Om vi använder **indexering** tar vi sekvensens minnesadress, adderar index-värdet, och läser ett element framåt från den beräknade minnesadressen

# Exempel

In [19]:
```
spam[1]
```

Out[19]:
```
'X'
```

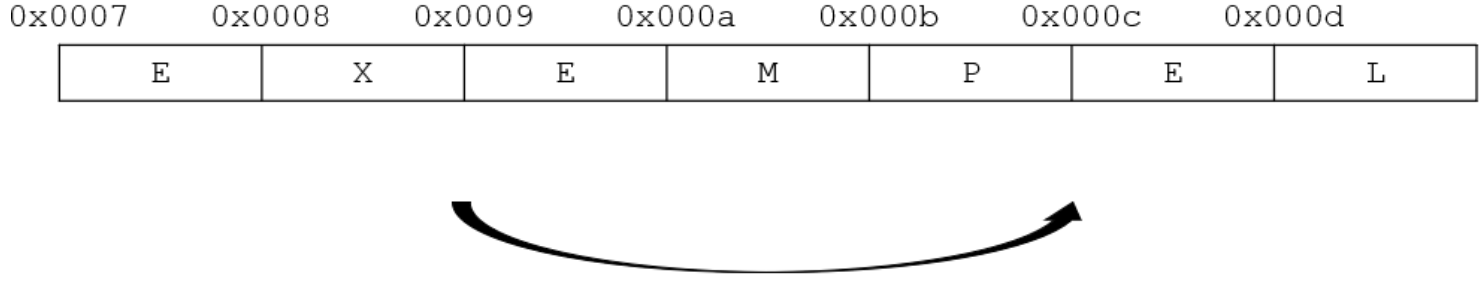



- Om vi använder **indexering** tar vi sekvensens minnesadress, adderar index-värdet, och läser ett element framåt från den beräknade minnesadressen

# Exempel

In [20]:
```
spam[2:5]
```

Out[20]:
```
'EMP'
```




- Om vi använder "utsnitt" (eng: **slice**) adderar start- och stopvärdena till sekvensens minnesadress och läser från den den första och fram till den andra beräknade minnesadressen.

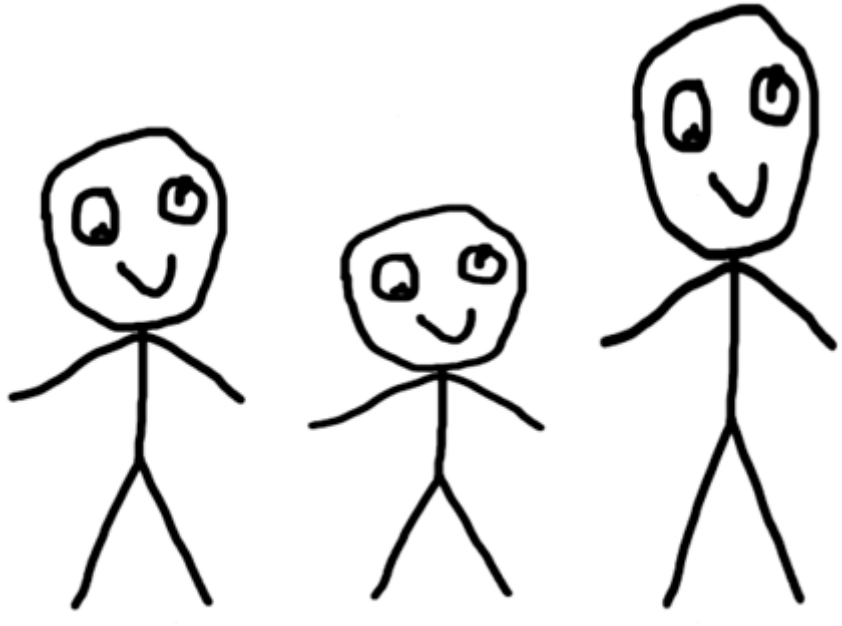
# Sekvenser av värden

Programmatisk bearbetning

# De flesta beräkningarna och processer i datorer är sekventiella

- Information läses sekventiellt från start till slut.
- Vad får detta för konsekvenser?

# Vem är längst?

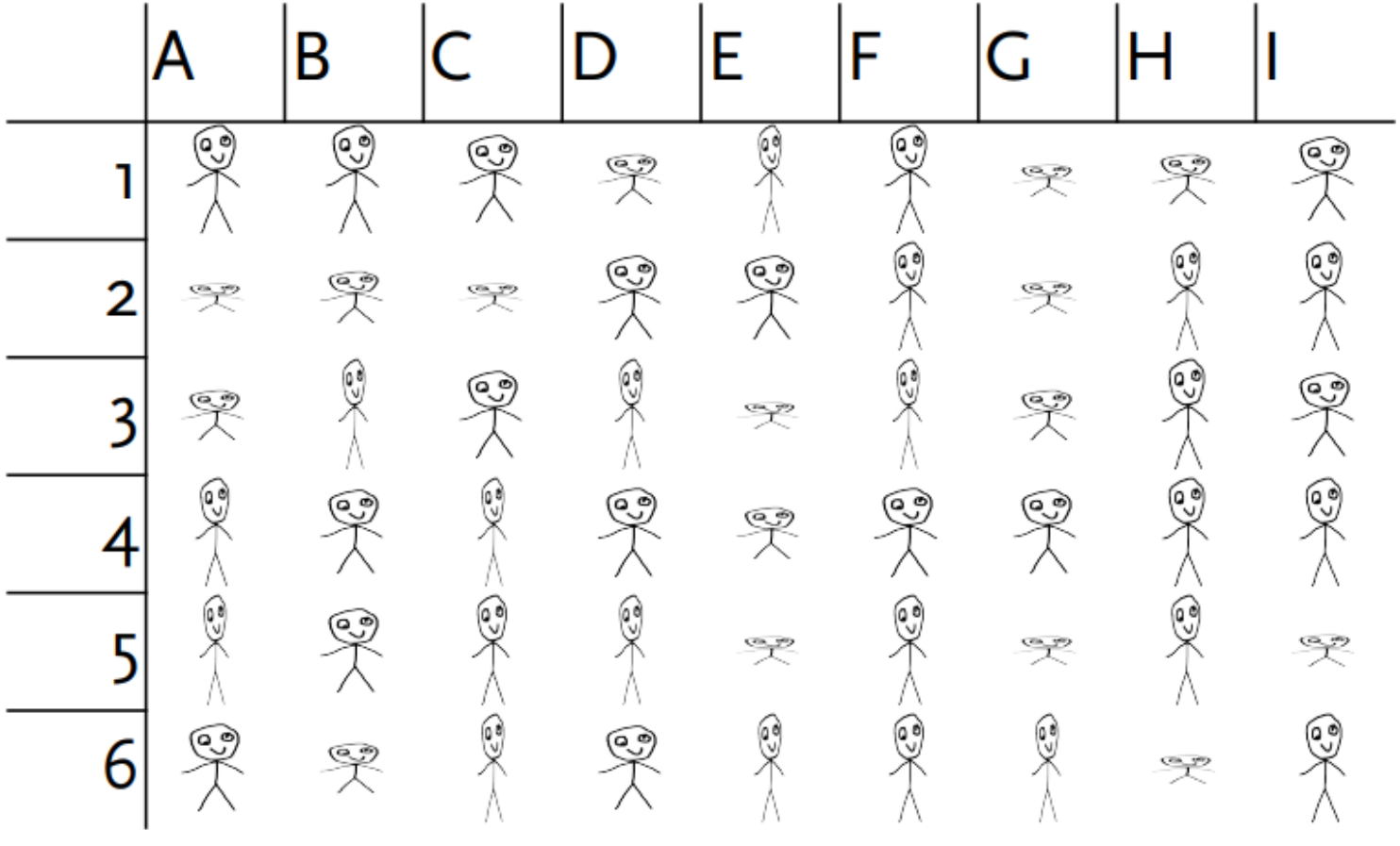


A B C

# Vem är längst?

| 120 | 100 | 160 |
| --- | --- | --- |
| A | B | C |

# Vem är längst?



# Vem är längst?

| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 71 | 6 | 56 | 5 | 57 | 55 | 84 |
| 2 | 17 | 89 | 13 | 31 | 22 | 30 | 34 | 65 | 61 |
| 3 | 69 | 3 | 2 | 62 | 68 | 86 | 8 | 23 | 67 |
| 4 | 14 | 7 | 2 | 32 | 7 | 11 | 63 | 60 | 16 |
| 5 | 32 | 2 | 70 | 89 | 3 | 72 | 68 | 59 | 15 |
| 6 | 53 | 7 | 24 | 8 | 26 | 28 | 67 | 54 | 64 |

# Från datorns perspektiv

- Har datorn en översiktsbild? Kan datorn titta på en hel sekvens på en gång?

# Från datorns perspektiv

- Har datorn en översiktsbild? Kan datorn titta på en hel sekvens på en gång?

- Datorn kan "titta" på ett element i taget, möjligtvis två precis vid en jämförelse
- Vi måste tillhandahålla arbetsminne
- Datorn kommer inte ihåg något "automatiskt"

# Vad behövs för att bearbeta en sekvens av data?

- något sätt att gå igenom sekvensen
- något sätt att beskriva regler för vad som göras beroende på vad för värden vi har i sekvensen

# Vad behövs för att bearbeta en sekvens av data?

- något sätt att gå igenom sekvensen
- något sätt att beskriva regler för vad som göras beroende på vad för värden vi har i sekvensen

- (låter detta bekant?)

# Nya komponenter

- villkor
- sanningsvärden (ny datatyp)
- jämförelser
- logiska operatorer
- loopar

# Nya komponenter

- villkor
- sanningsvärden (ny datatyp)
- jämförelser
- logiska operatorer
- loopar

- Vi kan använda dessa, inte bara för att bearbeta sekvenser, utan för att styra programflöde.

# Villkorssatser

- Används för att bestämma vilka delar av koden som ska köras
- Olika scenarion:
    - "Kör kodavsnitt $A$ om vissa villkor uppfylls"
    - "Kör det första kodavsnitt av $A_1, A_2, \ldots, A_n$ vars villkor stämmer"
    - "Kör det första kodavsnitt av $A_1, A_2, \ldots, A_n$ vars villkor stämmer, om inget av deras villkor stämmer, kör kodavsnitt $A_{n+1}$"

# Villkorssatser

- Bestämma villkor för när kod ska köras.

```
if <uttryck som beräknas till ett sanningsvärde>:
    sats1
    sats2
    sats3
    ...
```

# Sanningsvärden

- Datatyper vi stött på: `int` (heltal), `float` (flyttal)
- Ny datatyp: sanningsvärde; eng boolean, Python `bool`
    - Efter matematikern och logikern George Boole
- Finns två sanningsvärden: `True` och `False`.

# Sanningsvärden

- Datatyper vi stött på: `int` (heltal), `float` (flyttal)
- Ny datatyp: sanningsvärde; eng boolean, Python `bool`
    - Efter matematikern och logikern George Boole
- Finns två sanningsvärden: `True` och `False`.

- "Sannighet": Om andra datatyper än sanningsvärden används tolkar python dem enligt följande:
    - allt förutom värdena `None`, `[]`, `""` och `0` ses som sanna
    - (det finns fler, men vi har inte gått igenom de datatyperna än)

# Villkorssatser

- Bestämma villkor för när kod ska köras.

```
if <uttryck som beräknas till ett sanningsvärde>:
    sats1
    sats2
    sats3
    ...
```

- När sanningsvärdet är `True` körs satserna i **blocket** som hör till if-**satsen**.
- När sanningsvärdet är `False` hoppar pythontolken över blocket som hör till if-satsen.

# Exempel

In [21]:
```
if True:
    print("Det är sant.")
if False:
    print("Det är falskt.")
```

```
Det är sant.
```

In [22]:
```
if 30-30:
    print("Jag kan räkna med heltal.")
```

# Exempel

In [24]:
```
det_regnar = False
det_snöar = True

if det_regnar:
    print("Det regnar!")
if det_snöar:
    print("Det snöar!")
```

```
Det snöar!
```

# Exempel

In [25]:
```
def testfunktion1():
    return False

def testfunktion2():
    return "False"

def testfunktion3():
    print(True)

if testfunktion1():
    print("Test 1 OK.")
if testfunktion2():
    print("Test 2 OK.")
if testfunktion3():
    print("Test 3 OK.")
```

```
Test 2 OK.
True
```

# Jämförelser (jämförelseoperatorer)

- Jämförelseoperatorer är operatorer vars beräknade värde alltid är ett sanningsvärde, dvs. `True` eller `False`.
- Jämförelseoperatorer i Python
    - lika med: `==`
    - inte lika med: `!=`
    - större än: `>`
    - mindre än: `<`
    - större eller lika med: `>=`
    - mindre eller lika med: `<=`
- Exempel
    - `x == 5`
    - `mätvärde >= maxvärde`

# Exempel

```
# villkor med jämförelser
if mina_poäng > dina_poäng:
    print("Jag vann!")

if mina_poäng == dina_poäng:
    print("Oavgjort!")

if dina_poäng > mina_poäng:
    print("Ett fel har uppstått och programmet kommer avslutas.")
```

# Exempel

```
# returvärde från len() används som en operand vid jämförelse
if len(meddelande) > 140:
    print("Meddelandet är för långt.")

# returvärde från finns_i_ordlista() används som sanningsvärde
if finns_i_ordlista(ord):
    print("Draget är giltigt.")
```

# Logiska operatorer

- Logiska operatorer används för att *kombinera* eller *negera* sanningsvärden.
- Uttryck med logisk operator beräknas alltid till ett sanningsvärde, dvs antingen `True` eller `False`.
    - `and`: ger `True` om **båda** operander är sanna
    - `or`: ger `True` om **minst en** av operanderna är sanna
    - `not`: ger `True` om operanden är `False`, ger `False` om operanden är `True`
- Exempel
    - `mätvärde1 > 100 and mätvärde2 <= 10`

# Exempel

In [26]:
```
# villkor med logiska operatorer
temperatur = -6.2
nederbörd = True
```

# Exempel

In [26]:
```
# villkor med logiska operatorer
temperatur = -6.2
nederbörd = True
```

In [27]:
```
if temperatur < 0 and nederbörd:
    print("Jag tror att det snöar.")
```

```
Jag tror att det snöar.
```

# Exempel

In [26]:
```
# villkor med logiska operatorer
temperatur = -6.2
nederbörd = True
```

In [27]:
```
if temperatur < 0 and nederbörd:
    print("Jag tror att det snöar.")
```

```
Jag tror att det snöar.
```

In [28]:
```
if temperatur < 0 and not nederbörd:
    print("Det snöar i alla fall inte.")
```

# Exempel

In [26]:
```
# villkor med logiska operatorer
temperatur = -6.2
nederbörd = True
```

In [27]:
```
if temperatur < 0 and nederbörd:
    print("Jag tror att det snöar.")
```

```
Jag tror att det snöar.
```

In [28]:
```
if temperatur < 0 and not nederbörd:
    print("Det snöar i alla fall inte.")
```

In [29]:
```
if temperatur > -100 and temperatur < 100:
    print("Jag tror att vi är på en annan planet.")
```

```
Jag tror att vi är på en annan planet.
```

# Flera möjliga förgreningar

# Olika mönster av villkorssatser

- Blocket tillhörande `if` -satsen utförs om villkoret i `if` -satsen är sann.

```
if sanningsvärde:
  satser
```

- Exempel

```
if sensor_value < 150:
  output = 0
```

# Varje if-sats är fristående

- Alla `if`-satser på samma nivå kommer att kontrolleras.
- Flera `if`-satser kan vara sanna

In [30]:
```
sensor_value = 100
if sensor_value < 150:
    print("Mindre än 150!")
if sensor_value < 100:
    print("Mindre än 100!")
if sensor_value < 200:
    print("Mindre än 200!")
```

```
Mindre än 150!
Mindre än 200!
```

# Gör $A$ om villkoret är sant, annars, gör $B$

- Ibland vill vi utföra något om villkoret inte är sant, dvs om villkoret är falskt.
- Om vi t.ex. alltid vill utföra antingen $A$ eller $B$.

# Gör $A$ om villkoret är sant, annars, gör $B$

- Ibland vill vi utföra något om villkoret inte är sant, dvs om villkoret är falskt.
- Om vi t.ex. alltid vill utföra antingen $A$ eller $B$.

- Varje villkorssats kan ha en tillhörande `else`-gren som endast utförs om villkoret är falskt.

# if-sats tillsammans med else-sats

- en `else`-del kan (frivilligt) användas tillsammans med varje `if`-sats.
- `else`-satsens block utförs om villkorssatsen är falsk (dvs när `if`-grenen inte utförs)

```
if sanningsvärde:
  satser
else:
  satser
```

# if-sats tillsammans med else-sats

- Exempel

In [31]:

```
if sensor_value < 150:
    output = 0
else:
    output = 1
```

# Kontrollera flera villkor, utför ett av dem

- Vi kan vilja ställa upp ett antal olika villkor i prioritetsordning och endast utföra det första villkoret som är sant.
    - Om $A$ är sant gör $B$
    - Om $A$ är falskt, kolla om $C$ är sant, i så fall gör $D$
    - Om $A$ och $C$ är falska, kontrollera om $E$ är sant, i så fall gör $F$

## `if`-sats tillsammans med en eller flera `elif`-satser

- **Om sanningsvärde1 är sant**, utförs blocket tillhörande `if`-satsen.
    - Inga `elif`-satser i anslutning till `if`-satsen utvärderas.
- **Om sanningsvärde1 är falskt, utvärderas närmaste `elif`-sats** och dess block utförs om dess sanningsvärde är sant. Övriga `elif` / `else`-satser hoppas över.

```
if sanningsvärde1:
  satser
elif sanningsvärde2:
  satser
# fler elif-satser kan följa
```

# if-sats tillsammans med en eller flera elif-satser

- Exempel

```
if sensor_value < 100:
  output = 0
elif sensor_value >= 100 and sensor_value < 200:
  output = 1
elif sensor_value >= 200 and sensor_value < 300:
  output = 2
```

# if-sats med elif-satser och avslutande else-sats

- Om **varken if-satsen eller någon av dess elif-satser hade ett sanningsvärde som var sant**, utförs det block som hör ihop med else-satsen.

```
if sanningsvärde:
    satser
elif sanningsvärde:
    satser
# fler satser med elif kan följa
else:
    satser
```

# if-sats med elif-satser och avslutande else-sats

- Exempel

```
if sensor_value < 75:
  output = 0
elif sensor_value >= 75 and sensor_value < 150:
  output = 1
elif sensor_value >= 150 and sensor_value < 225:
  output = 2
else:
  output = 3
```

# Utföra samma sak flera gånger

loopar, iteration, repetition

# Loopar för att repetera/iterera

- Används för att köra samma kod flera gånger
- Exempel
    - skriva ut varje element i en lista
    - addera ett tal till en variabel tills det överstiger ett visst värde
    - flytta fram figuren tills det nuddar en vägg

# Två loopkonstruktioner i Python

- `while`-loopen
    - som en villkorssats men blocket upprepas så länge som villkoret är uppfyllt
- `for`-loop
    - används med en sekvens
    - utför blocket för varje element som finns i sekvensen

# while-loopen

... som en if-sats fast blocket utförs om och om igen så länge som dess villkor är sant.

# while-loopen

- Satserna i blocket som tillhör `while`-loopen upprepas tills `while`-satsens villkor blir falskt.
- Efter sista satsen i blocket börjar loopen om från blockets första sats
- Varje "varv" loopen gör kallas för en **iteration**

```
while <uttryck som beräknas till ett sanningsvärde>:
  sats1
  sats2
  ...
```

# Exempel

In [ ]:

```
# räkna upp till 10
count = 0
while count <= 10:
    print(count)
    count += 1
```

# Exempel

```
# evig loop
x = 0
while x <= 10:
    print("En dator blir aldrig uttråkad.")
    print(x)
```

# Avbryta en loop, eller gå till nästa iteration i en loop

- Nyckelordet `break` användas för att avbryta en loop (hela loopen).
- Nyckelordet `continue` används för att hoppa över resten av loop-blocket och börja om igen från början (nästa iteration)

In [32]:

```python
# kontrollvariabel för loopen
count = 0
while count < 10:
    # Gå vidare till nästa iteration om count är 3 eller 5, notera att vi *in
    if count == 3 or count == 5:
        count += 1
        continue

    if count > 7:
        print("Nu orkar jag inte mer." + "(count: " + str(count) + ")")
        break
    elif count > 3:
        print("Är vi framme snart?" + "(count: " + str(count) + ")")
    elif count > 4:
        print("Har vi kört vilse?" + "(count: " + str(count) + ")")
    else:
        print("Det här är roligt!" + "(count: " + str(count) + ")")
    count += 1
```

```
Det här är roligt!(count: 0)
Det här är roligt!(count: 1)
Det här är roligt!(count: 2)
Är vi framme snart?(count: 4)
Är vi framme snart?(count: 6)
Är vi framme snart?(count: 7)
Nu orkar jag inte mer.(count: 8)
```

# Bearbetning av strängar och listor med while-loop

# Skriva ut alla värden i en lista

In [33]:
```
namn = ["Ada", "Beata", "Cecilia", "Diana"]

index = 0
while index < len(namn):
    print(namn[index])
    index += 1
```

```
Ada
Beata
Cecilia
Diana
```

# Skriva ut alla värden i en lista som funktion

In [36]:
```
def print_all_values(values):
    index = 0
    while index < len(values):
        print(values[index])
        index += 1

namn = ["Ada", "Beata", "Cecilia", "Diana"]
print_all_values(namn)
```

```
Ada
Beata
Cecilia
Diana
```

# Skriva ut alla värden i en lista som funktion

In [36]:
```
def print_all_values(values):
    index = 0
    while index < len(values):
        print(values[index])
        index += 1

namn = ["Ada", "Beata", "Cecilia", "Diana"]
print_all_values(namn)
```

```
Ada
Beata
Cecilia
Diana
```

In [37]:
```
print_all_values(["Hej", "Hopp"])
```

```
Hej
Hopp
```

# for-loopen

# for-loopen

```
for element in a_sequence:
    statement1
    statement2
    statement3
    ...
```

- `for` -loopen loopar över en sekvens
- Vid varje iteration tilldelas variabeln `element` nästa värde i sekvensen `a_sequence`
- **OBS!** variablerna behöver inte heta `element` och `a_sequence`

# Gå igenom en lista med for

In [38]:
```
names = ["Adam", "Bethany", "Chris"]

# gå igenom alla element i names. det aktuella elementet
# refereras till via variabeln name i loopen.
for name in names:
    # skriv ut det aktuella elementet i listan names
    print(name)
```

```
Adam
Bethany
Chris
```

# Gå igenom en sträng med for

In [39]:
```
word = "fantastisk"

# gå igenom alla tecken i word. det aktuella tecknet
# refereras till via variabeln char i loopen.
for char in word:
    # skriv ut det aktuella tecknet i strängen word
    print(char)
```

```
f
a
n
t
a
s
t
i
s
k
```

# Byt ut bokstäver

In [40]:
```
word = "fantastisk"
new_word = ""

for char in word:
    # lägg till "i" istället för "a"
    if char == "a":
        new_word += "å"
    elif char == "s":
        new_word += "l"
    else:
        new_word += char
print(new_word)
```

```
fåntåltilk
```

# Vi kan inte ändra på en sträng

- I Python är strängar värden som är oföränderliga (eng. **immutable**)
- Vi kan alltså inte byta ut ett tecken på en visst index i en sträng.
- Vi kan däremot skapa en ny sträng genom att slå ihop en existerande sträng med en annan sträng (operatorn `+`)

# Funktionen range()

# range()

- Skapar en "virtuell lista med heltal" (en **generator**) som man kan använda som en vanlig lista med heltal.
- `range(stop)` : Börja från 0, öka med 1 och fortsätt generera heltal så länge som heltalet är mindre än `stop`
- `range(start, stop)` : Börja från `start` , öka med 1 och fortsätt generera heltal så länge som heltalet är mindre än `stop`
- `range(start, stop, step)` : Börja från `start` , öka med `step` och fortsätt att generera heltal så länge som heltalet är mindre än `stop`
- För att få en riktig lista kan vi skicka returvärdet från `range()` till funktionen `list()` , t.ex. `list(range(10))`

# range()

In [41]:
```
list(range(10))
```

Out[41]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# range()

In [41]:
```
list(range(10))
```

Out[41]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [42]:
```
list(range(4,6))
```

Out[42]:
```
[4, 5]
```

# range()

In [41]:
```
list(range(10))
```

Out[41]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [42]:
```
list(range(4,6))
```

Out[42]:
```
[4, 5]
```

In [43]:
```
list(range(10, 101, 10))
```

Out[43]:
```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

# range()

In [41]:
```
list(range(10))
```

Out[41]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [42]:
```
list(range(4,6))
```

Out[42]:
```
[4, 5]
```

In [43]:
```
list(range(10, 101, 10))
```

Out[43]:
```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

In [44]:
```
list(range(10, 110, 10))
```

Out[44]:
```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

# range()

In [41]:
```
list(range(10))
```

Out[41]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [42]:
```
list(range(4,6))
```

Out[42]:
```
[4, 5]
```

In [43]:
```
list(range(10, 101, 10))
```

Out[43]:
```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

In [44]:
```
list(range(10, 110, 10))
```

Out[44]:
```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

In [45]:
```
list(range(10, 0, -2))
```

Out[45]:
```
[10, 8, 6, 4, 2]
```

# range()

In [41]:
```
list(range(10))
```

Out[41]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [42]:
```
list(range(4,6))
```

Out[42]:
```
[4, 5]
```

In [43]:
```
list(range(10, 101, 10))
```

Out[43]:
```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

In [44]:
```
list(range(10, 110, 10))
```

Out[44]:
```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

In [45]:
```
list(range(10, 0, -2))
```

Out[45]:
```
[10, 8, 6, 4, 2]
```

In [46]:
```
list(range(10, 3, -2))
```

Out[46]:
```
[10, 8, 6, 4]
```

# for med range()

In [47]:
```
# Skriv ut siffrorna 0 till 9 (10 siffror)
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

# for med range() över längden på en sekvens

Ofta överflödigt:

In [48]:
```
names = ["Adam", "Bethany", "Chris"]
# Skriv ut elementen i listan names
for i in range(len(names)):
    print(names[i])
```

```
Adam
Bethany
Chris
```

# for med range() över längden på en sekvens

men ibland praktiskt:

In [49]:
```
names = ["Adam", "Bethany", "Chris"]
# Skriv ut elementen i listan names
for i in range(len(names)):
    print(names[i] + " likes " + names[(i+1) % len(names)])
```

```
Adam likes Bethany
Bethany likes Chris
Chris likes Adam
```

# Funktioner och scope

Tips: http://pythontutor.com/

# Anropsstacken (eng. call stack)

- En **stack** är en datatyp som fungerar lite som en lista men:
    - Nya element läggs alltid till på toppen av stacken (eng: **push**)
    - Bara det element som ligger på toppen av stacken kan tas bort (eng: **pop**)
- Liknelse: En lista där nya element bara kan läggas till i slutet, och bara det sista elementet kan tas bort.
- **Anropsstacken** är en stack som håller reda på var vi befinner oss i exekveringen av ett Python-skript.
    - På anropsstacken lagras omgivningar (eng: frames) som innehåller de variabler och funktioner som Python-tolken har tillgång till.

# Programflöde - funktionsanrop

- Ett pythonprogram tolkas sekventiellt av pythontolken.
- Vi tänker oss att vi har en körmarkör som pekar på vad det som ska utföras näst
- När pythontolken ser ett funktionanrop görs följande
    1. eventuella **argument** i funktionsanropet **beräknas**
    2. eventuella **argument skickas till den anropade funktionen** och en **frame för anropet skapas**
    3. den anropade funktionens **funktionskropp utförs sekventiellt** - körmarkören flyttas till funktionskroppen
    4. **när funktionen är klar tas dess frame bort** och
    5. **körmarkören flyttas tillbaka dit anropet påträffades** och funktionens **returvärde "ersätter" funktionsanropet**

# Programflöde - funktionsanrop

In [50]:
```
print("Första raden")

def calc_double(value):
    new_value = value * 2
    print("värdet på value:", value)
    print("värdet på new_value:", new_value)
    return new_value

def main():
    print("Nu körs main()")
    value = 5
    double = calc_double(value)
    print("Dubbla värdet är", double)

main()
print("Nu är programmet slut!")
```

```
Första raden
Nu körs main()
värdet på value: 5
värdet på new_value: 10
Dubbla värdet är 10
Nu är programmet slut!
```

Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 1 of 17

Print output (drag lower right corner to resize)

Frames Objects

Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 2 of 17

Print output (drag lower right corner to resize)

```
Första raden
```

Frames Objects

Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

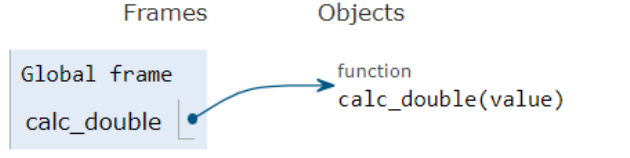
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 3 of 17

Print output (drag lower right corner to resize)

```
Första raden
```



Python 3.6
known limitations

```
 1  print("Första raden")
 2
 3  def calc_double(value):
 4      new_value = value * 2
 5      print("värdet på value:", value)
 6      print("värdet på new_value:", new_value)
 7      return new_value
 8  def main():
 9      print("Nu körs main()")
10      value = 5
11      double = calc_double(value)
12      print("Dubbla värdet är", double)
13
14  main()
15  print("Nu är programmet slut!")
```

Edit this code

line that just executed
next line to execute

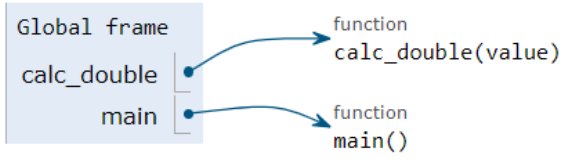
<< First | < Prev | Next > | Last >>

Step 4 of 17

Print output (drag lower right corner to resize)

```
Första raden
```

Frames Objects



Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

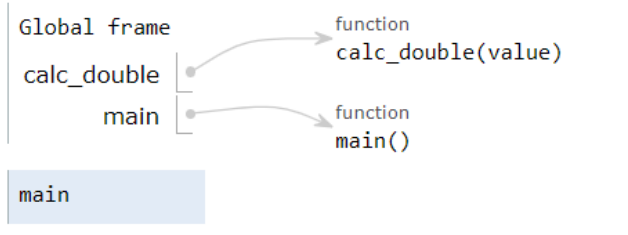
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 5 of 17

Print output (drag lower right corner to resize)

```
Första raden
```




```
main
```

Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

line that just executed
next line to execute

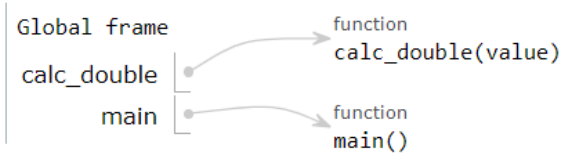
<< First | < Prev | Next > | Last >>

Step 6 of 17

Print output (drag lower right corner to resize)

```
Första raden
```

Frames Objects




main

Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

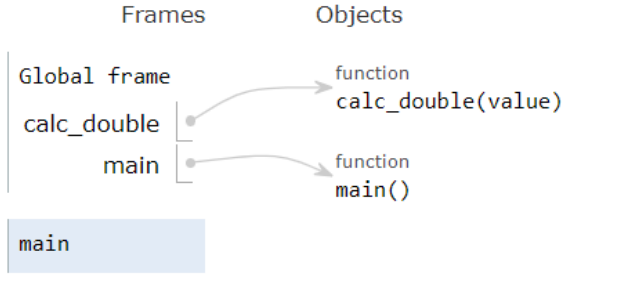
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 7 of 17

Print output (drag lower right corner to resize)

```
Första raden
Nu körs main()
```




```
main
```

Python 3.6
known limitations

```
 1  print("Första raden")
 2
 3  def calc_double(value):
 4      new_value = value * 2
 5      print("värdet på value:", value)
 6      print("värdet på new_value:", new_value)
 7      return new_value
 8  def main():
 9      print("Nu körs main()")
10      value = 5
11      double = calc_double(value)
12      print("Dubbla värdet är", double)
13
14  main()
15  print("Nu är programmet slut!")
```

Edit this code

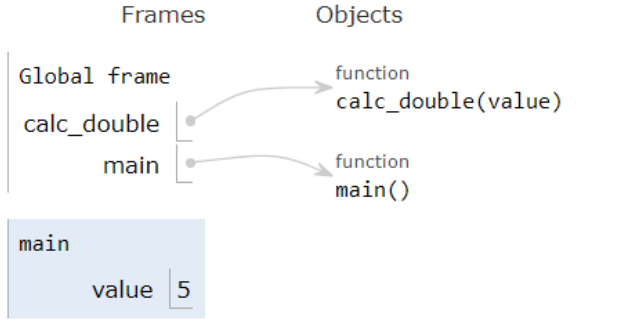
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 8 of 17

Print output (drag lower right corner to resize)

```
Första raden
Nu körs main()
```



Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

line that just executed
next line to execute

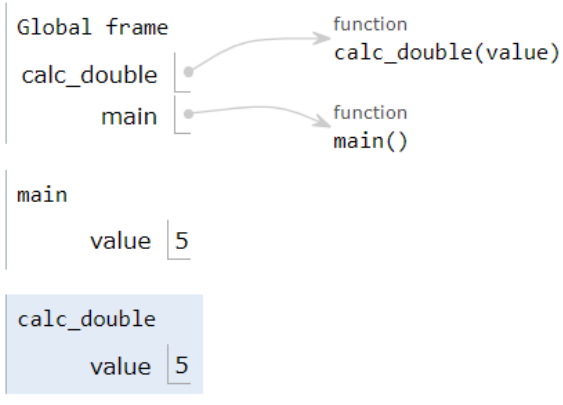
<< First | < Prev | Next > | Last >>

Step 9 of 17

Print output (drag lower right corner to resize)

```
Första raden
Nu körs main()
```

Frames Objects



Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

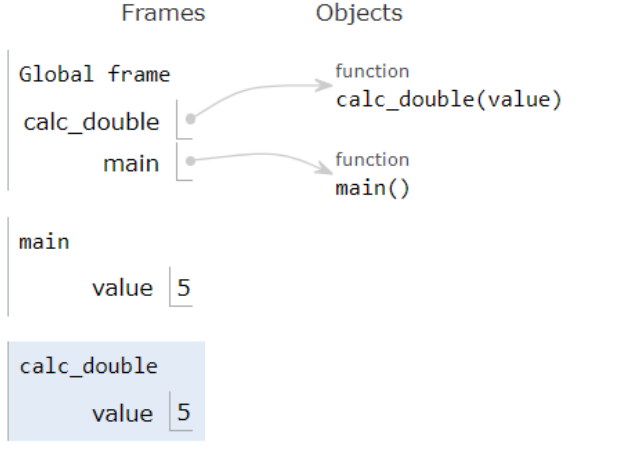
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 10 of 17

Print output (drag lower right corner to resize)

```
Första raden
Nu körs main()
```



Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

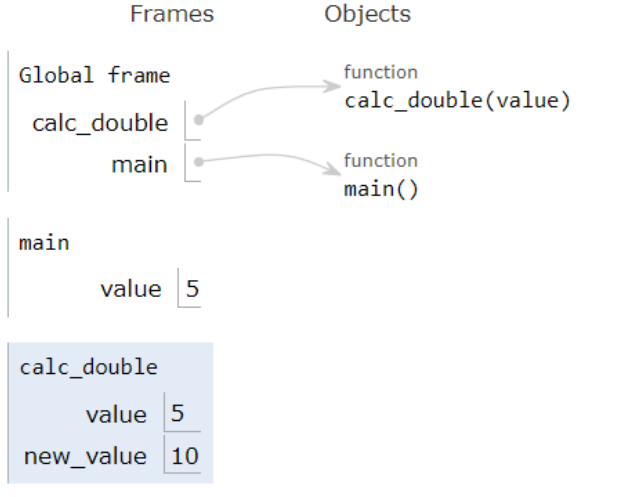
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 11 of 17

Print output (drag lower right corner to resize)

```
Första raden
Nu körs main()
```



Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

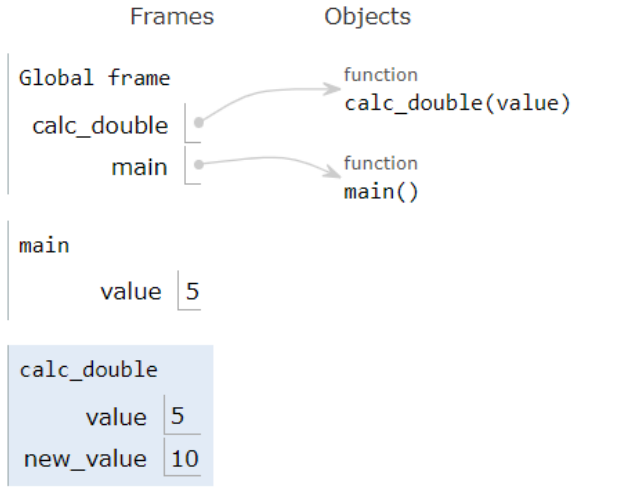
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 12 of 17

Print output (drag lower right corner to resize)

```
Första raden
Nu körs main()
värdet på value: 5
```



Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

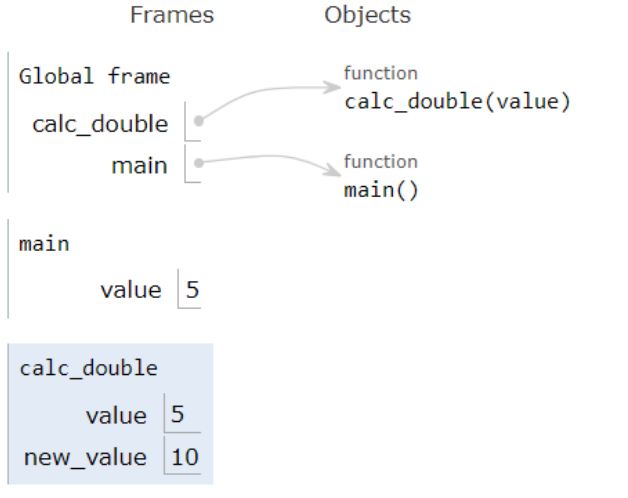
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 13 of 17

Print output (drag lower right corner to resize)

```
Första raden
Nu körs main()
värdet på value: 5
värdet på new_value: 10
```



Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

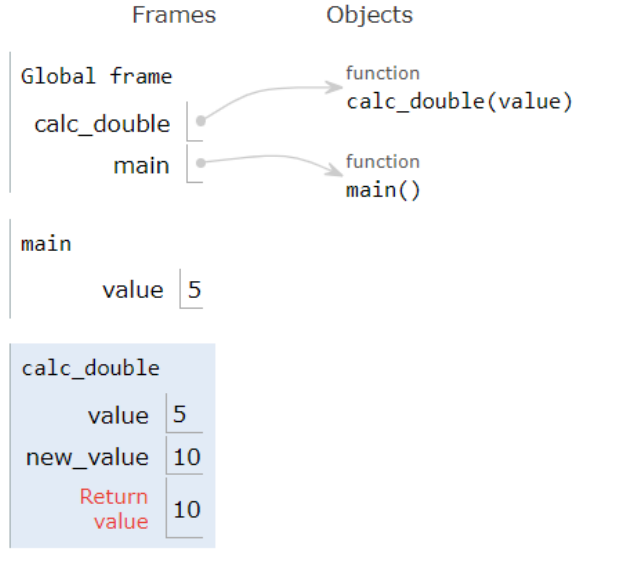
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 14 of 17

Print output (drag lower right corner to resize)

```
Första raden
Nu körs main()
värdet på value: 5
värdet på new_value: 10
```



Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

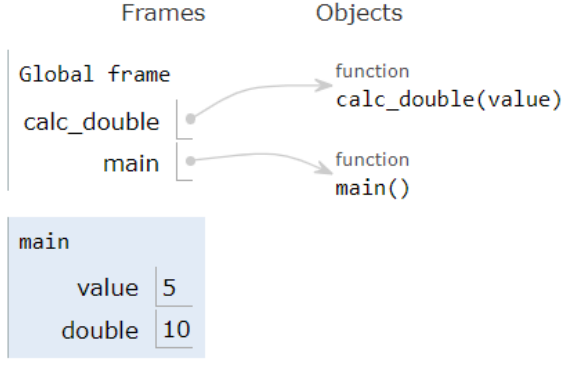
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 15 of 17

Print output (drag lower right corner to resize)

```
Första raden
Nu körs main()
värdet på value: 5
värdet på new_value: 10
```



Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

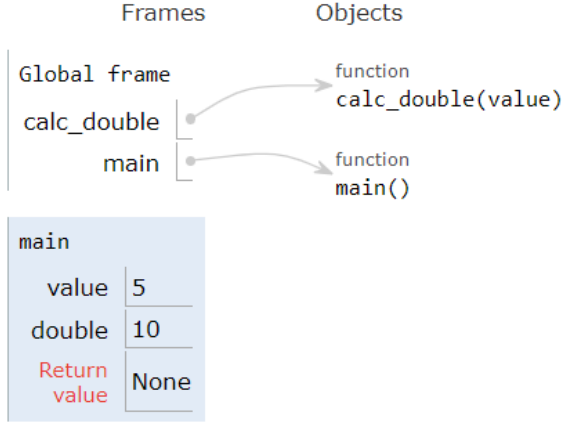
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 16 of 17

Print output (drag lower right corner to resize)

```
Första raden
Nu körs main()
värdet på value: 5
värdet på new_value: 10
Dubbla värdet är 10
```



Python 3.6
known limitations

```
1  print("Första raden")
2
3  def calc_double(value):
4      new_value = value * 2
5      print("värdet på value:", value)
6      print("värdet på new_value:", new_value)
7      return new_value
8  def main():
9      print("Nu körs main()")
10     value = 5
11     double = calc_double(value)
12     print("Dubbla värdet är", double)
13
14 main()
15 print("Nu är programmet slut!")
```

Edit this code

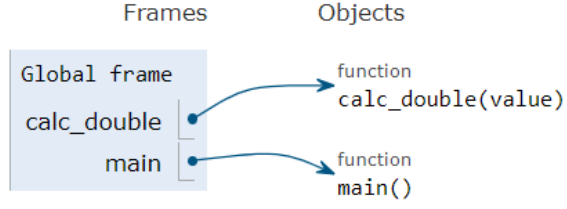
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 17 of 17

Print output (drag lower right corner to resize)

```
Första raden
Nu körs main()
värdet på value: 5
värdet på new_value: 10
Dubbla värdet är 10
```



# Funktionsanrop - lokala variabler

- När en funktion anropas skapas en ny frame för just det funktionsanropet (eng. *function call*).
- Alla variabler som definieras när funktionen körs tillhör funktionsaropets frame.
- Dessa kallas för **lokala variabler** - variabler som tillhör en ickeglobal frame.
- När körmarkören hoppar från en körande funktion till dess anrop (return), förstörs den frame som hörde ihop med funktionsanropet.

# Scope

- Termen "scope" i programmering har att göra med i vilka kontexter saker är giltiga.
- Scopet för en lokal variabel är endast anropets frame.
- Dvs **den lokala variabeln är endast giltig och tillgänglig i funktionsanropets frame**.

# Lokal variabel

In [ ]:

```
print("Första raden")

def calc_double(value):
    new_value = value * 2
    print("värdet på value:", value)
    print("värdet på new_value:", new_value)
    return new_value

def main():
    print("Nu körs main()")
    value = 5
    double = calc_double(value)
    print("Dubbla värdet är", double)

main()
# vad händer om försöker skriva ut value här?
print("value:", value)
print("Nu är programmet slut!")
```

In [ ]:

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

# Global och lokal variabel

In [ ]:
```
value = "hejsan"
main()
```

In [ ]:
```
print("value:", value)
```

In [ ]:
```
main()
```

In [ ]:
```
# vad händer om försöker skriva ut value här?
print("value:", value)
print("Nu är programmet slut!")
```

# Testning och felsökning

# Olika typer av fel

- **Syntaxfel**: Koden bryter mot programspråkets grammatiska regler, exempelvis
    - använda reserverade ord på fel sätt, t.ex. att döpa en variabel till `return` eller en funktion till `if`
    - saknat kolon i slutet av if-sats
- **Runtime-fel**: Fel som uppstår vid körning och resulterar i en krasch, exempelvis
    - Division med 0
    - Försöka läsa en fil som inte existerar
- Logiska fel i programmet, exempelvis
    - Felaktig operatorprioritet, t.ex. råka skriva `x + y * 5` när man menade `(x + y) * 5`
    - *Off-by-one*-fel, t.ex. glömma bort att indexering börjar på 0, inte 1

# Spårutskrifter

- Utskrift av variabelvärden
- Utskrift av text för att visa var i koden pythontolken befinner sig

# Spårutskrifter

- Utskrift av variabelvärden
- Utskrift av text för att visa var i koden pythontolken befinner sig

- Hur vi skrev ut variabeln `value` på olika ställen för att se vilket värde den hade i en viss kontext

# Felsökning av olika typer av fel

- Syntaxfel: kontrollera syntax, antal blanksteg, parenteser, kolon, etc.
- runtime-fel: tolka felmeddelandet, lägg till spårutskrifter
- Logiska fel i programmet: försök att isolera felet, på vilka ställen i koden används variabeln/funktionen som har med felet att göra? Lägg till spårutskrifter.

# Felsökning av olika typer av fel

- Syntaxfel: kontrollera syntax, antal blanksteg, parenteser, kolon, etc.
- runtime-fel: tolka felmeddelandet, lägg till spårutskrifter
- Logiska fel i programmet: försök att isolera felet, på vilka ställen i koden används variabeln/funktionen som har med felet att göra? Lägg till spårutskrifter.

- När det är svårt att hitta felet: det är bättre att vara noggrann och dubbelkolla med spårutskrifter än att göra antaganden.

# Testning

- Att veta ***hur*** man testar sitt program är en grundläggande och relativt enkel färdighet för en programmerare
- Att veta ***vad*** man ska testa är inte alltid lätt
    - vi vill testa så mycket som möjligt med så få tester som möjligt
    - för att veta vad man ska testa, behöver man alltså ha en förståelse för vad som kan gå fel
    - detta blir lättare med erfarenhet (eftersom man gjort flera typer av fel)

# Exempel på hur man kan testa

- Interaktiva anrop
- Fördefinierade funktionsanrop
    - med utskrifter
    - med villkor som skriver ut OK/inte OK
- `assert`-satser
- Ramverk för enhetstester (eng. *unit testing*)
- Läs vidare på https://www.ida.liu.se/~729G46/tips/testa_kod/

# Testning med interaktiva anrop

- Kör pythontolken interaktivt
    - `$ python3 -i filnamn.py`
    - `$ ipython3 -i filnamn.py`
- Utför olika anrop till de funktioner du vill testa
- **Fördelar**: kan testa på ett utforskande sätt
- **Nackdelar**: omständigt att upprepa testerna när kodfilen ändrats.

# Fördefinierade anrop + utskrifter

- Lägg till testanrop och utskrifter av resultat i kodfilen.
- Kör kodfilen
    - `$ python3 kodfil.py`
- **Fördelar**: Samma test upprepas varje gång som filen körs.
- **Nackdelar**: Behöver studera utskrifterna och avgöra om resultatet är korrekt eller inte.

# Fördefinierade anrop + villkorade utskrifter

- Lägg till testanrop och villkor som skriver ut
    - ett meddelande om resultatet är korrekt
    - ett annat meddelande om resultatet är felaktigt
- Kör kodfilen
    - `$ python3 kodfil.py`
- **Fördelar**: Samma test upprepas varje gång som filen körs och utskrifterna berättar om resultatet är korrekt eller inte.
- **Nackdelar**: Lite mer kod

# Fördefinierade anrop + villkorade utskrifter

- Lägg till assert-satser
    - `assert funktionsanrop() == korrekt_värde`
- Kör kodfilen
    - `$ python3 kodfil.py`
- **Fördelar**: Samma test upprepas varje gång som filen körs och programmet kraschar om resultatet är fel
- **Nackdelar**: Bör inte användas i produktionskod

# Programabstraktion

dela upp ett problem/program i mindre delar genom att skapa funktioner som löser delproblem

# Dela upp ett problem i delproblem

- Samma grundidé som i många andra delar av ingenjörskonsten
- Lättare att lösa ett mindre och avgränsat problem än ett stort, komplext problem
- Exempel
    - all konstruktion...
    - hus
    - bilar
    - telefoner
    - datorer

# Program- och dataabstraktion

- Programabstraktion
    - nedbrytning av uppgifter och processer
- Dataabstraktion
    - nedbrytning av hur information är strukturerad

# Bygga upp ett program med hjälp av funktioner

- Istället för att ha en funktion som gör allt
    - identifiera mönster
    - deluppgifter som förekommer på flera olika ställen i ett program
- Skriv en funktion för varje deluppgift
- Uppdelningen kan vara hierarkisk:
    - Problem $A$ består av delproblemen $B$ och $C$
    - Problem $B$ består av delproblemen $E$, $F$ och $G$
    - Problem $C$ består av delproblemen $H$, $E$ och $I$

# Välj beskrivande variabelnamn...

In [ ]:

```
i = ""
while i != 'q':
    i = input("Skriv något: ")
    p = None
    h = False
    for c in i:
        if p and p == c:
            h = True
        p = c
    if h:
        print("Det du skrev har upprepade bokstäver i sig!")
    else:
        print("Det finns inga upprepade bokstäver där.")
print("Hej då.")
```

# Välj beskrivande variabelnamn...

In [ ]:

```
user_input = ""
while user_input != 'q':
    user_input = input("Skriv något: ")
    prev_char = None
    has_repeated = False
    for curr_char in user_input:
        if prev_char and prev_char == curr_char:
            has_repeated = True
        prev_char = curr_char
    if has_repeated:
        print("Det du skrev har upprepade bokstäver i sig!")
    else:
        print("Det finns inga upprepade bokstäver där.")
print("Hej då.")
```

# Vi kan skapa egna abstraktioner med t.ex. funktioner

In [ ]:

```
def has_repeated_char(s):
    prev_char = None
    for curr_char in s:
        if prev_char and prev_char == curr_char:
            return True
        prev_char = curr_char
    return False

user_input = ""
while user_input != 'q':
    user_input = input("Skriv något: ")
    if has_repeated_char(user_input):
        print("Det du skrev har upprepade bokstäver i sig!")
    else:
        print("Det finns inga upprepade bokstäver där.")
print("Hej då.")
```

# Kommentarer kan göra det lättare att förstå koden, men bara om de stämmer!

In [ ]:

```
def has_repeated_char(s):
    """Returnera True om upprepade tecken finns i s, annars False."""
    prev_char = None
    for curr_char in s:
        # Kolla om föregående tecken är samma som aktuellt tecken.
        if prev_char and prev_char == curr_char:
            return True
        prev_char = curr_char
    return False

# Fråga användaren om ett ord och berätta om det innehåller upprepade tecken
# eller inte.
user_input = ""
while user_input != 'q':
    user_input = input("Skriv något: ")
    if has_repeated_char(user_input):
        print("Det du skrev har upprepade bokstäver i sig!")
    else:
        print("Det finns inga upprepade bokstäver där.")
print("Hej då.")
```

# Exempel (skiss)

In [ ]:

```
def ta_emot_kommando():
    satser
    return kommando

def kommando1():
    satser

def kommando2():
    satser

def utför_kommando(kommando):
    satser

def main():
    while True:
        kommando = ta_emot_kommando()
        utför_kommando(kommando)
```

# Exempel (skiss)

In [ ]:

```
def ladda_ordlistor():
    satser
def ladda_ordlista(ordlista):
    satser
def autocomplete_strategi1():
    satser
def autocomplete_strategi2():
    satser
def autocomplete(strategi, ord, ordlistor)
    satser
def välj_strategi():
    satser
def main():
    ordlistor = ladda_ordlistor()
    strategi = välj_strategi()
    while True:
        ord = input("Skriv ett ord: ")
        if ord != "q":
            kompletterat_ord = autocomplete(strategi, ord, ordlistor)
            print(kompletterat_ord)
```

# Läsa från textfil

# Läsa data från textfil

- Läsning sker sekventiellt
- Olika strategier
    - hela filen som en textsträng
    - en rad i taget, en rad → en sträng
    - hela filen som en lista, varje rad blir ett element i en lista

# Förberedelser innan läsning

- Filen måste öppnas för läsning
- Liknelse: undersöka innehåll i en låda från en lagerlokal
    - *ange vilken låda* - ange sökväg till filen
    - *öppna låda för att plocka ut saker* - öppna filen i "läsläge"
    - *plocka fram innehåll* - läs från fil

# Läsa in information från en fil

- Funktionen `open` tar in en sträng (namn på fil), och öppnar filen och **returnerar** den öppna filen (speciell datatyp)

In [ ]:
```
file = open(filnamn)
file = open(filnamn, 'r') # read - endast läsning
file = open(filnamn, 'w') # write - skriva över
file = open(filnamn, 'a') # append - lägga till
```

# Läsa in information från en fil

- Metoden `.readlines` läser innehållet i filen och **returnerar** det som en **lista av strängar**.
    - (metod: funktion som sitter ihop med ett värde)

In [ ]:
```
file = open("hemligt.txt")
contents = file.readlines()
```

# Stänga fil

- Vi stänger en fil när vi är klara med den (praxis, för att undvika diverse problem)

In [ ]:
```
file = open("data.csv")
contents = file.readlines()
file.close()
```

# Funktionen repr

- `repr` returnerar sträng inklusive citattecken för att indikera att det är en sträng

In [ ]:
```
print(True)
print("True")
print(repr(True))
print(repr("True"))
```

# Sträng- och listmetoder

# ”Metod”

- En metod är en funktion som hör ihop med ett objekt (mer om detta längre fram i kursen).
- Exempel (vi skriver `datatyp.metod()` för att illustrera, men i normala fall skriver man `variabelnamn.metod()`):
    - `list.append(objekt)`
    - `string.join(list)`
    - `string.index(substring)`

# Praktiska strängmetoder

- `str.upper()`
- `str.lower()`
- `str.strip()`
- `str.rstrip()`
- `str.lstrip()`
- `str.split()`
- `str.startswith()`
- `str.endswith()`
- https://docs.python.org/3/library/stdtypes.html#string-methods

# Listmetoder

- `list.append()`
- `list.extend()`
- `list.remove()`
- `list.pop()`
- `list.insert()`
- `list.index()`
- `list.sort()`
- `list.count()`
- https://docs.python.org/3/tutorial/datastructures.html#more-on-lists