

- **Funktioner med enkla parametrar**
 - Mer om funktioner med andra parametrar i kommande föreläsning(ar)...
- **Strukturera kod**
- **Scope, variablers räckvidd**

Strukturerar kod

```
82 self.color = colors
83
84 def on_key_press(self, symbol, modifiers):
85     #self.token_type, self.token_string, _, _, _ = next(token)
86     if self.context_index == -1:
87         if symbol == key.UP and not self.active_index == 0:
88             self.menu_labels[self.active_index].color = [255, 255, 255, 255]
89             self.active_index -= 1
90             self.mags_dt = self.get_act_color_mags()
91         elif symbol == key.DOWN and not self.active_index == 3:
92             self.menu_labels[self.active_index].color = [255, 255, 255, 255]
93             self.active_index += 1
94             self.mags_dt = self.get_act_color_mags()
95         elif symbol == key.ENTER:
96             if self.active_index == 1:
97                 pygame.app.quit()
98             else:
99                 self.context_index = 1
100         elif symbol == key.ESCAPE:
101             if self.context_index == 1:
102                 self.context_index = -1
103             elif self.context_index == 1:
104                 self.context_index = -1
105         else:
106             self.cur_game.on_key_press(symbol, modifiers)
107     else:
108         if symbol == key.ESCAPE:
109             self.context_index = -1
110
111
```

```
import tokenize #java.util.*;
import io, math #import java.io.*;
import myParser

class Calculator: #klassen ska börja med stor bokstav
    def __init__(self, token, v):
        self.token = token
        self.variables = v #{"PI": math.pi, "E": math.e, "ans": 0.0}
        self.parser = parser.Parser(self.token, self.variables)

    def statement():
        self.token_type, self.token_string, _, self.end, _ = next(token)
        print('i statement.', self.token_string)
        while self.end == tokenizer.NEWLINE:
            self.token_type, self.token_string, _, self.end, _ = next(token)
            print('inte newline')
        command = self.token_string
        print(command)
```

```
print("Numerical calculator version 2013-04-10")
s=input("> ")
code_buffer = io.StringIO(s)
token = tokenize.generate_tokens(code_buffer.readline)
variables = {"PI": math.pi, "E": math.e, "ans": 0.0}
p = myParser.My_parser(token, variables)
print('h')
calc = Calculator(token, variables)
while (true):
    calc.statement()
```

```
def term(self):
    prod = self.factor()
    #self.token_type, self.token_string, _, _, _ = next(token)
    while (self.token_string == '*' or self.token_string == '/'):
        if self.token_string == '*':
            self.token_type, self.token_string, _, _, _ = next(token)
            prod = prod * self.factor()
            print('i term prod=', prod)
        if self.token_string == '/':
            self.token_type, self.token_string, _, _, _ = next(token)
            namnare = self.factor() #int(self.token_string)
            if namnare == 0.: # EXCEPTION HÄR!
                print('Division by 0')
                return -1
            prod = prod/namnare
    return prod

def factor(self):
    return self.primary()

def primary(self):
    result = 99999
    print('primary self.token_type', self.token_type)
    if self.token_string == '(':
        self.token_type, self.token_string, _, _, _ = next(token)
        result = self.assignment()
        if self.token_string == ')':
            self.token_type, self.token_string, _, _, _ = next(token)
            return result
    else:
        print("Expected ')") # SYNTAX-EXCEPTION HÄR!

    elif self.token_type == 2: # tokenize.NUMBER:
        print('i primary, elif type==2', self.token_string)
        result = float(self.token_string)
        self.token_type, self.token_string, _, _, _ = next(token)
        return result

    elif self.token_string == '-': #negering token_type == 54
        self.token_type, self.token_string, _, _, _ = next(token)
        result = -self.primary()
        return result
```

Programvara är ofta stor och komplex. 100-tals programmerare kan arbeta med olika delar av koden. Det behövs en struktur.

```

82     self.color = colors
83
84     def on_key_press(self, symbol, modifiers):
85         # Delegate key press to
86         if self.context_index == -1:
87             if symbol == key.UP and not self.active_index == 0:
88                 self.menu_labels[self.active_index].color = [255, 255, 255, 255]
89                 self.active_index -= 1
90                 self.mags.dt = self.get_act_color_mags()
91             elif symbol == key.DOWN and not self.active_index == 3:
92                 self.menu_labels[self.active_index].color = [255, 255, 255, 255]
93                 self.active_index += 1
94                 self.mags.dt = self.get_act_color_mags()

```

```

94 import tokenize #java.util.*;
95 import io, math #import java.io.*;
96 import myParser
97
98 class Calculator: #klassen ska börja med stor bokstav
99     def __init__(self, token, v):
100         self.token = token
101         self.variables = v #{"PI": math.pi, "E": math.e, "ans": 0.0}
102         self.parser = parser.Parser(self.token, self.variables)
103
104     def statement():
105         self.token_type, self.token_string, _, self.end, _ = next(token)
106         print('i statement.', self.token_string)
107         while self.end == tokenizer.NEWLINE:
108             self.token_type, self.token_string, _, self.end, _ = next(token)
109             print('inte newline')
110             command = self.token_string
111             print(command)

```

```

print("Numerical calculator version 2013-04-10")
s=input("> ")
code_buffer = io.StringIO(s)
token = tokenize.generate_tokens(code_buffer.readline)
variables = {"PI": math.pi, "E": math.e, "ans": 0.0}
p = myParser.My_parser(token, variables)
print('h')
calc = Calculator(token, variables)
while (true):
    calc.statement()

```

```

def term(self):
    prod = self.factor()
    #self.token_type, self.token_string, _, _, _ = next(token)
    while (self.token_string == '*' or self.token_string == '/'):
        if self.token_string == '*':
            self.token_type, self.token_string, _, _, _ = next(token)
            prod = prod * self.factor()
            print('i term prod=', prod)
        if self.token_string == '/':
            self.token_type, self.token_string, _, _, _ = next(token)
            namnare = self.factor() #int(self.token_string)
            if namnare == 0.: # EXCEPTION HÄR!
                print('Division by 0')
                return -1
            prod = prod/namnare
    return prod

def factor(self):
    return self.primary()

def primary(self):
    result = 99999
    print('primary self.token_type', self.token_type)
    if self.token_string == '(':
        self.token_type, self.token_string, _, _, _ = next(token)
        result = self.assignment()
        if self.token_string == ')':
            self.token_type, self.token_string, _, _, _ = next(token)
            return result
    else:
        print("Expected '('") # SYNTAX-EXCEPTION HÄR!

    elif self.token_type == 2: # tokenize.NUMBER:
        print('i primary, elif type==2', self.token_string)
        result = float(self.token_string)
        self.token_type, self.token_string, _, _, _ = next(token)
        return result

    elif self.token_string == '-': #negering token_type == 54
        self.token_type, self.token_string, _, _, _ = next(token)
        result = -self.primary()
        -----

```

En grundläggande strategi är att dela upp koden i **moduler**, filer, som delas in i **funktioner** som gör väl avgränsade uppgifter. Moduler kan läggas i **paket**.

```
81 self.color = colors[self.mags.dt[i]]
82
83 def on_key_press(self, symbol, modifiers):
84     # Handle key presses
85     if self.context_index == 1:
86         if symbol == key.UP and not self.active_index == 0:
87             self.menu_labels[self.active_index].color = [255, 255, 255]
88             self.active_index -= 1
89             self.mags.dt = self.get_act_color_mags()
90         elif symbol == key.DOWN and not self.active_index == 3:
91             self.menu_labels[self.active_index].color = [255, 255, 255]
92             self.active_index += 1
93             self.mags.dt = self.get_act_color_mags()
94
95 import tokenize #java.util.*;
96 import io, math #import java.io.*;
97 import myParser
98
99 class Calculator: #klassen ska börja med stor bokstav
100     def __init__(self, token, v):
101         self.token = token
102         self.variables = v {"PI": math.pi, "E": math.e, "ans": 0.0}
103         self.parser = parser.Parser(self.token, self.variables)
104
105     def statement():
106         self.token_type, self.token_string, _, self.end, _ = next(token)
107         print('i statement.', self.token_string)
108         while self.end == tokenizer.NEWLINE:
109             self.token_type, self.token_string, _, self.end, _ = next(token)
110             print('inte newline')
111         command = self.token_string
112         print(command)
```

```
print("Numerical calculator version 2013-04-10")
s=input("> ")
code_buffer = io.StringIO(s)
token = tokenize.generate_tokens(code_buffer.readline)
variables = {"PI": math.pi, "E": math.e, "ans": 0.0}
p = myParser.My_parser(token, variables)
print('h')
calc = Calculator(token, variables)
while (true):
    calc.statement()
```

```
def term(self):
    prod = self.factor()
    #self.token_type, self.token_string, _, _, _ = next(token)
    while (self.token_string == '*' or self.token_string == '/'):
        if self.token_string == '*':
            self.token_type, self.token_string, _, _, _ = next(token)
            prod = prod * self.factor()
            print('i term prod=', prod)
        if self.token_string == '/':
            self.token_type, self.token_string, _, _, _ = next(token)
            namnare = self.factor() #int(self.token_string)
            if namnare == 0.: # EXCEPTION HÄR!
                print('Division by 0')
                return -1
            prod = prod/namnare
    return prod

def factor(self):
    return self.primary()

def primary(self):
    result = 99999
    print('primary self.token_type', self.token_type)
    if self.token_string == '(':
        self.token_type, self.token_string, _, _, _ = next(token)
        result = self.assignment()
        if self.token_string == ')':
            self.token_type, self.token_string, _, _, _ = next(token)
            return result
    else:
        print("Expected '(')" # SYNTAX-EXCEPTION HÄR!

    elif self.token_type == 2: # tokenize.NUMBER:
        print('i primary, elif type==2', self.token_string)
        result = float(self.token_string)
        self.token_type, self.token_string, _, _, _ = next(token)
        return result

    elif self.token_string == '-': #negering token_type == 54
        self.token_type, self.token_string, _, _, _ = next(token)
        result = -self.primary()
        -----
```

Många färdiga moduler som ofta används.
math, random, matplotlib (ou4), turtle, ...

Funktioner och metoder i Python

Strukturerar kod mha funktioner

- Hittills: skrivit kod som exekveras (körs) sekventiellt.
- Lång kod, upprepningar?

Funktioner:

- Dela in koden i "delar" med specifika uppgifter
- Lättare att läsa!
- Lättare att felsöka
- Lättare att återanvända (ex sin-funktionen finns redan)
- Undvik upprepning av kod
- Funktionerna gör typiskt **en** (del)uppgift och är lättare att återanvända än ett helt program

Exempel hittills: Funktioner känns (ofta) igen på parenteser:

```
namn = input('Vad heter du?')
```

```
hyp = math.sqrt(a*a + b*b)
```

**Funktioner
som följer
med python**

Inbyggda

print()

input()

int()

...

importeras

math.sin()

turtle.forward()

csv.reader()

...

**Vanliga
funktioner,
behöver ofta
laddas ned (ou4)**

matplotlib.pyplot()

matplotlib.xscale()

...

**Funktioner
du/ni skriver**

Lektion 4 --

[Python's built-in functions: https://docs.python.org/3/library/functions.html](https://docs.python.org/3/library/functions.html)

Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

Built-in Functions				
<code>abs()</code>	<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>
<code>all()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>any()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>ascii()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>bin()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bool()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>breakpoint()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	

abs(x)

Return the absolute value of a number. The argument may be an integer, a floating point number, or an object implementing `__abs__()`. If the argument is a complex number, its magnitude is returned.

all(iterable)

Funktioner med enkla parameterar

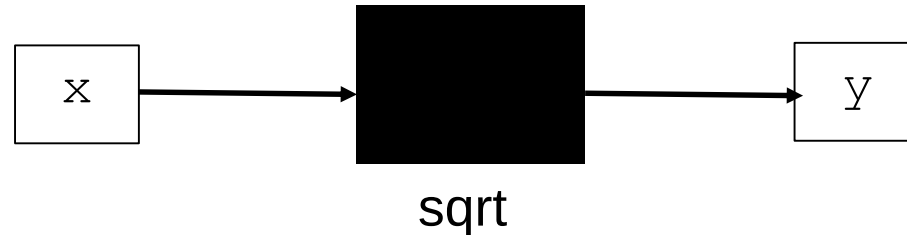
Exempel på en funktion som finns i modulen *math*:

```
y = math.sqrt(x)      # Förutsätter satsen: import math
```

En funktion är som en svart låda. Man stoppar in ett eller flera värden och ut kommer ett resultat som beror på värdena man stoppar in.

I ovan exempel **anropas** funktionen *sqrt* och vid anropet skickas *x* till funktionen. Funktionen ger resultatet, **returnerar**, värdet av $\sqrt{x}$.

Det värde man skickar till funktionen kallas **argument**, i detta fall är *x* argument.



```
y = math.pow(x, 4) # y tilldelas returvärdet av math.pow
```

Funktionen *pow* har två argument och returnerar vid detta anrop x^4

Funktionerna *sqrt* och *pow* är exempel på funktioner i modulen *math*.

Nu skall vi se hur man kan **konstruera egna funktioner**. Detta kallas för att **definiera** funktioner. Funktionens namn hittar vi på själva, men namnet bör vara beskrivande.

Ex. en funktion *average* som beräknar medelvärdet av två tal. Anropet kan ske så här:

```
c = average(a, b)
```



Funktionen har två argument, som vid anropet skickas till funktionen som beräknar och returnerar ett resultat till den kod som gör anropet.

Funktionen kan även anropas så här:

```
q = average(x, y) # argumenten är x och y
```

```
w = 5.0 + 7*average(5, z) # argumenten är 5 och z
```

```
print(average(math.sqrt(8), 9)) # argumenten är math.sqrt(8) och 9
```

I ovan exempel tas för givet att variablerna *x*, *y* och *z* existerar.

Vi måste nu *definiera* funktionen, dvs skriva de pythonsatser som gör att funktionen existerar och att den gör det den skall. Här är ***funktionsdefinitionen***:

```
def average(x, y):          # Funktionshuvud
    return (x + y) / 2      # Funktionskropp
```

Funktionshuvudet består av tre delar, det reserverade ordet *def*, funktionens namn och en parentes med funktionens *parametrar* *x* och *y* samt tecknet kolon.

Funktionskroppen är resterande del som är indragen (*indenterat*) relativt huvudet. Det reserverade ordet *return* är en sats som innebär att metoden returnerar ett resultat och avslutas. I detta fall beräknas $(x+y) / 2$ som returnas till den kod som anropar funktionen.

Funktionskroppen hade alternativt kunna se ut så här:

```
r = (x + y) / 2
return r
```

Funktionskroppen är innehålllet av den “svarta lådan”, ty den beskriver hur funktionen löser problemet.

Antag att all kod i programmet ser ut så här med funktionens definition överst och ett exempel på kod som anropar funktionen

```
def average(x, y):  
    r = (x + y) / 2  
    return r  
  
# anropande kod  
b = 5  
c = average(b, 4)
```

Notera: Koden för definitionen måste ligga ovanför anropande kod, annars fattar ej python vad `average` är

Det som händer när man kör koden är följande:

Python tolkar koden från början till slutet.

1. Aha tänker python, en funktionsdefinition som heter `average`. Den har två parametrar och returnerar ett värde. Den skall jag "komma ihåg".
2. Tilldela variabeln `b` värdet 5
3. Beräkna högerledet, dvs anropa funktionen `average`.
4. Python kommer då att "hoppa" till koden i funktionen och göra dess kod.
5. Värdet av högerledet (det som funktionen returnerar) tilldelas variabeln `c`

Detta händer vid anropet, mer detaljer

Notera: Koden för definitionen skall ligga ovanför anropande kod

```
b = 5 # Sats 1
c = average(b, 4) # Sats 2
```

```
def average(x, y):
    r = (x + y) / 2
    return r
```

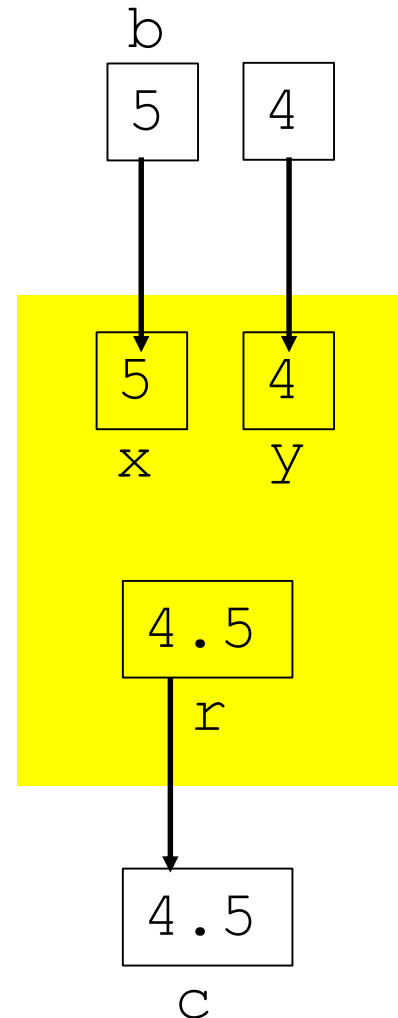
←Värdet av r

Vid sats 1 tilldelas b värdet 5.
Vid sats 2 anropas `average` och värdena av argumenten b och 4 **kopieras** till funktionens parametrar x resp. y .

Funktionen har tre st lokala variabler för att lösa sitt problem: x och y som den får vid anropet samt r för beräkningen.

När funktionen returnerar r och avslutas, kopieras värdet av r till högerledet i sats 2 och variabeln c tilldelas det värdet.

De lokala variablerna x , y , r i funktionen existerar enbart i funktionen.



Funktioner behöver ej ha en return-sats, ett exempel.

```
def printAvalue(x):  
    print('Value =', x)
```

Anrop:

```
printAvalue(8) # Följande skrivs ut: Value = 8
```

Funktioner som ej returnerar explicit med returnsats, returnerar dock alltid `None`.

`None` är ett reserverat ord i Python.

```
print( printAvalue(8) ) # Sådant anrop av print är Nonsens!  
# Skriver ut två rader:  
Value = 8  
None
```

Ett annat exempel:

```
x = printAvalue(8) ) # Sådant anrop av print är Nonsens!  
print('x=', x)  
# Skriver ut två rader:  
Value = 8  
x = None
```

Kommer detta att fungera?

```
def swap(a, b):  
    temp = a    # spara a i temp  
    a = b       # kopiera b till a  
    b = temp    # kopiera temp till b  
  
# Anropande kod  
x = 7  
y = 5  
swap(x, y) # anropafunktionen swap med argumenten x och y  
print(x, y)
```

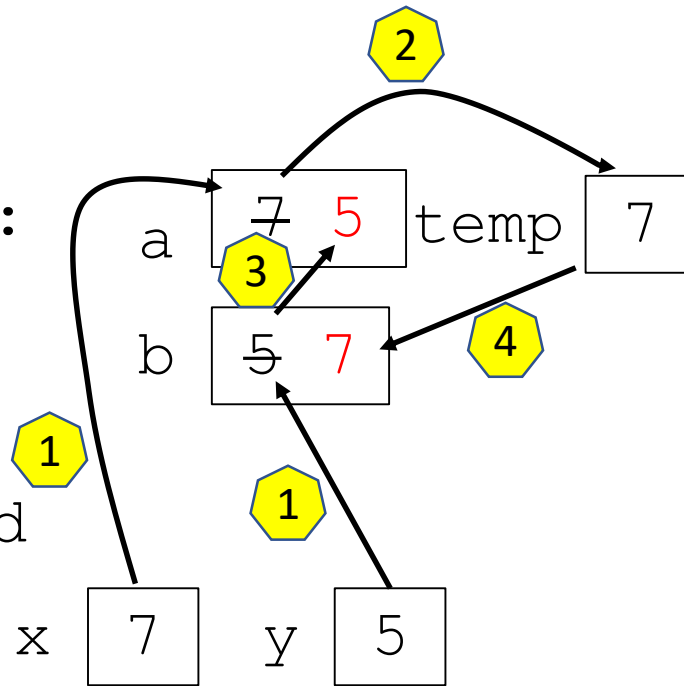
?

Vi skissar på
tavlan vad som
händer ...

Eller se nästa sida...

Detta händer...

```
def swap(a, b):  
    temp = a  
    a = b  
    b = temp  
# Anropande kod  
x = 7  
y = 5  
swap(x, y)  
print(x, y)
```



Vid anropet får `a` värdet 7 och `b` 5.
Sedan:

- `temp` får värdet 7
- `a` får värdet 5
- `b` får värdet 7

Skriver ut: 7 5, dvs `x` och `y` är oförändrade. Variablerna `a` och `b` som är lokala i funktionen `swap` byter värden. Men detta får ingen åverkan på variablerna `x` och `y` i den anropande koden.

Beräkna siffersumman av ett heltal.

ex: $123 = 1 + 2 + 3 = 6$

Ide': Dividera talet upprepat med 10. I varje iteration beräkna entalssiffran och summera. $123 \% 10 = \mathbf{3}$, $12 \% 10 = \mathbf{2}$, $1 \% 10 = \mathbf{1}$, summera $3+2+1$

```
def digitSum(n):      # n är parameter och lokal variabel
    s = 0              # s lokal variabel
    while n > 0:
        r = n % 10     # r lokal variabel
        s = s + r
        n = n // 10    # Heltalsdivison, dividera bort entalet
    return s
```

```
print(digitSum(12345)) # -> 15
```

Annan ide': Plocka fram siffrorna från vänster?



$123 // 100 = 1$
$23 // 10 = 2$
$3 // 1 = 3$

Funktionshuvud kommenteras exvis så här enligt s.k. docstring-standard:

```
def average(x, y):  
    '''Takes in two numbers, returns their sum'''  
    r = (x + y) / 2  
    return r
```

Defaultvärden i parameterlistan

Exempel, beräkna saldot med ett givet kapital, räntesats och antal år.

$\text{Saldot} = \text{kapital} * (1 + 0.01 * \text{räntesats})^{\text{år}}$

Parametrarna `kapital` och `år` har defaultvärden:

```
def saldo(räntesats, kapital=100, år=1):  
    return kapital*(1+0.01*räntesats)**år
```

```
print(saldo(1.0))           # kapital=100, år=1  
print(saldo(1.0, 10, 5))  
print(saldo(1.0, 20))      # år=1
```

Ger:

101.0

10.51

20.2

Funktioner kan returnera flera värden

Exempel, funktion som returnerar en s.k. tuple. Mer om tuple, kommande Fö.

```
def roots(a, b, c):  
    '''Calc the roots of  $ax^2 + bx + c = 0$ '''  
    d = (b * b) - (4 * a * c)  
    if(d > 0): # reella rötter  
        r1 = (-b + math.sqrt(d) / (2 * a))  
        r2 = (-b - math.sqrt(d) / (2 * a))  
        return (r1,r2) # tuple med två flyttal  
    else:  
        return None # None, ett reserverat ord i Python
```

Ex. anrop:

```
r = roots(1,6,-7) # r blir: (-2.0, -10.0)  
(x1,x2) = roots(1,6,-7) # x1 blir -2.0, x2=-10.0  
r = roots(1,1,1) # r blir None
```

Scope, variablers räckvidd i koden

```
def first_func(x): # Parametern x är en lokal variabel
    y = 1 # y är en lokal variabel som bara existerar i funktionen
    x = y+3 # Den lokala variabeln x ändras
    return x

def second_func():
    global x # Globala variabeln räcker utanför funktionen. Usch!!!
    x = 3
    text = 'hejsan, '
    return text*x

x = 2
print('first_func: ', first_func(x)) # Värdet 2 skickas till funk
print('x =', x)
# print(y) # Ger FEL. y bara definierad i sitt scope, dvs i funk first_func
print('second_func: ', second_func())
print('x =', x)
```

Snygg, lättläst kod som följer konventionen.

```
import ...

def average (...
    ...

def digitSum (...
    ...

def roots (...
    ...

#kod här:
Variabler...
funktionsanrop...
```

1. import-satser
2. Funktionsdefinitionerna
3. Resten av koden.

En tom rad mellan funktionsdefinitionerna mm.

Liten bokstav i namn, _ för tydlighet. "Beskrivande" namn på variabler, funktionsnamn

OBS! Lägg inte variabler eller funktionsanrop innan eller mellan funktionsdefinitionerna.