Flik 3: Grunderna

Operatorer

- Standardoperatorer för aritmetik:

  `+, -, *, /, //, %, **`

  (// för heltalsdivision)

```
>>> 2 + 2
4
>>> 2 + 3 * 4
14
>>> 17 / 7
2.4285714285714284
>>> 17 // 7
2
>>> 17 % 7
3
>>> 2 ** 5
32
>>>
```


© Init AB

Jämförelseoperatorer

- Jämförelseoperatorer och booleska operatorer:

```
<, <=, >, >=, ==, !=, and, or, not

>>> 2 < 4 and 2 == 4
False
>>> 2 > 4 or 2 < 4
True
>>> not 6.2 <= 6
True
>>> 3 < 4 < 5
True
>>> "Windows" if "Bill" > "Linus" else "Linux"
'Linux'
```


© Init AB

Variabler och tilldelning

- Variabler börjar med en bokstav eller _ följt av bokstäver, _, 0-9
- Även icke-ASCII-tecken är tillåtna
  - fast det kräver att man håller reda på teckenkodningen
- Stora och små bokstäver har betydelse i variabelnamn!

```
>>> a = 1
>>> print(a)
1
>>> print(A)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'A' is not defined
```


© Init AB

Dynamisk typning

- Dynamiskt typat — typen sätts vid tilldelningen

```
>>> a = 2
>>> b = 1.5
>>> c = a * b
>>> d = 1
>>> print(c)
3.0
>>> print(d/a)
0.5
>>> b = 'text'
>>> c = b + b
>>> print(c)
texttext
>>> print(2 * b)
texttext
>>> print(2 + b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```


© Init AB

Variabler och tilldelning

- Utökad tilldelning ("augmented")

```
>>> x = x + 10
```

kan skrivas som

```
>>> x += 10
```

```
+=, -=, *=, /=, //=, %=, **=
<<=, >>=, &=, ^=, |=
```

OBS! "a++", "++b" etc finns ej

- Multipel tilldelning

```
>>> x = y = z = 1
>>> x, y, z = 1, 2, 'a string'
```

- In-line tilldelning stödjs ej:

```
if a=my_func():
    print('Ok')
```


© Init AB

Variabler - tal, strängar, listor, tupler

- Tal

```
>>> x = 1      # Heltal
>>> x = 1.0       # Flyttal
```

- Strängar

```
>>> hej_en = 'hello' # Även " fungerar
>>> hej_en[0]
'h'
```

- Listor

```
>>> my_list1 = [1, 2, 3]
>>> my_list2 = [1, 2, 'a', 3, 'hej', 1.0]
>>> my_list2[4]
'hej'
```

- Tupler

```
>>> my_tuple1 = (1, 2, 3)
>>> my_tuple2 = (1, 2, 'a', 3, 'hej', 1.0)
>>> my_tuple2[4]
'hej'
>>>
```


© Init AB

Tupler är som listor, men ”read-only”

- Listor

```
>>> L = [2, 3, 4]
>>> L[2] = 5
>>> L.append(7)
>>> L
[2, 3, 5, 7]
```

- Tupler

```
>>> T = (2, 3, 4)
>>> T[2] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> T.append(7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> T
(2, 3, 4)
```


© Init AB

Några grundtyper i python

- Typer - exempel

```
>>> a = 1.0
>>> type(a)
<type 'float'>

>>> type('hej')
<type 'str'>

>>> type(0)
<type 'int'>

>>> type(False)
<type 'bool'>

>>> type(())
<type 'tuple'>

>>> type([])
<type 'list'>
```


© Init AB

Typkonvertering

```
>>> a = 1.0
>>> int(a)
1
>>> int("42")
42
>>> s = "42"
>>> isinstance(s, int)
False
>>> isinstance(s, str)
True
>>> s + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> int(s)
42
>>> int(s) + 3
45
>>>
```


© Init AB

Jämföra olika typer

```
>>> n = 2.2
>>> s = "2.2"
>>> n == s
False
>>> n == float(s)
True
>>>
```


© Init AB

None

- None står ett tomt värde (dvs. värde saknas)

```
>>> s = None
>>> type(s)
<type 'NoneType'>
>>> s is None
True
>>> ulla is None        # s finns med tomt värde; ulla finns ej
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ulla' is not defined
>>> t = "Göran"
>>> t is None
False
>>> t is not None
True
>>> s == t
False
>>> s == None
True
>>>
```


© Init AB

Indentering har betydelse...

- Indentering för att markera block (...=sekundär prompt)

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print('Be careful not to fall off!')
...
Be careful not to fall off!
```

- Indentering i en fil för att markera block:

```
if uttryck:
    a = 1
    b = 2
else:
    a = 2
    b = 1
print(a, b)
```


© Init AB

- Om *uttryck* är olika med noll eller *True* - exekvera *if_block*:

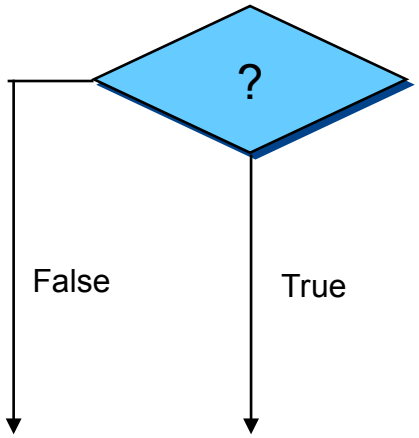
```
if uttryck:
    if_block
```

- Om *uttryck* är lika med noll eller *False* - exekvera *else_block*:

```
if uttryck:
    if_block
else:
    else_block
```

- Flera uttryck:

```
if uttryck1:
    if_block1
elif uttryck2:
    elif_block1
elif uttryck3:
    elif_block2
else:
    else_block
```



© Init AB

Iteration - While- och For-loopen

- while - när antal iterationer inte är känt i förväg

```
while uttryck:
    while_block
```

  - Iterera tills *uttryck* blir noll eller *False*

- for - när antal iterationer är känt i förväg

```
lista = [1, 2, 3]
for item in lista:
    for_block
```

  - Iterera över alla listelement

- *range()*-funktionen

```
for item in range(3):        # motsvarar listan [0,1,2]
    for_block
```

  - Utökad: *range(start, end, step=1)*

© Init AB

Exempel

- **Skriv tal**

```
for x in range(1, 20, 3):
    print(x, end=" ")
print()

$ python3 skriv_tal.py
1 4 7 10 13 16 19
$
```


© Init AB

Iteration - break, continue

- *break* - för att hoppa ur en *for*- eller *while*-loop i förtid
- *continue* - för att hoppa över resten av raderna i loopen och påbörja nästa iteration förutsatt att det finns värden kvar att loopa över (*for*) eller att villkoret fortfarande är uppfyllt (*while*)

© Init AB

Iteration - else

- *else:* i en *for-* eller *while*-loop - else-blocket körs förutsatt att loopen *inte* har avbrutits i förtid av en *break*

```
>>> for i in range(3):
...     print('varv: ', i)
... else:
        print('hela loopen löptes igenom')
...
varv: 0
varv: 1
varv: 2
hela loopen löptes igenom
```


© Init AB

Inmatning

- *input()*

```
>>> mat = input('Favoritmat: ')
Favoritmat: Räksmörgås
>>> print(mat)
Räksmörgås
>>>
```


© Init AB

Exempel

- **Gissa tal**

```
svar=4
while True:
    gissning=input("What is 2+2: ")
    if int(gissning) == svar:
        break
    print("Wrong!", end=" ")
print("Correct!")
```

```
$ python3 gissa.py
What is 2+2: 5
Wrong! What is 2+2: 4
Correct!
$
```


© Init AB

Modulen random

```
>>> import random
>>> random.randint(1, 10)
4
>>> random.randint(1, 10)
10
>>> L = [ "Bill", "Steve", "Linus", "Ken" ]
>>> random.choice(L)
'Ken'
>>> random.choice(L)
'Steve'
>>> random.sample(L, 2)
['Steve', 'Linus']
>>> random.shuffle(L)
>>> L
['Ken', 'Bill', 'Linus', 'Steve']
>>>
```


© Init AB

Övning

- **ovn0301/matte.py**
  - Källkoden till övningarna ska vara utcheckad.
  - Skapa en egen gren (branch).
  - Gå till katalogen `ovn0301`. Koden från exemplet på förra sidan ska finnas i filen `matte.py`. Kör programmet `test_matte.py` eller

```
$ python3 -m pytest verify.py -vx --tb=line
```

  Gör `git push` av din branch till servern. Finns din branch i Jenkins?

- **ovn0302/skrivtal.py** Gör ett program (i filen `ovn0302/skrivtal.py`) som skriver ut talen 0,1,...,20 på varsin rad. Därefter ska det skriva ut summan av alla talen på en ny rad. Kör testprogrammet. Checka in (commit) och pusha till din gren på servern. Kolla Jenkins.
  - Kod under `if __name__ == '__main__':` testas *inte*.
  - Testprogrammen förutsätter att koden läser från stdin och skriver till stdout (t.ex. via de vanliga funktionerna input och print).
  - Testprogrammen kan ibland vara petiga med hur utskrifterna ser ut.
  - Vissa tester förutsätter att `raise SystemExit` *inte* anropas.

© Init AB

Övning

- **ovn0303/gissatal.py** Skriv ett program som låter dig gissa på ett tal mellan 1 och 1000 tills du hittar rätt. Det skall för varje gissning indikera om gissningen var för hög eller för låg. Skriv också ut antalet gånger det krävdes för att hitta rätt.

```
Gissa tal (1-1000): 500
För litet! Gissa tal (1-1000): 750
För litet! Gissa tal (1-1000): 875
För stort! Gissa tal (1-1000): 812
För litet! Gissa tal (1-1000): 843
För stort! Gissa tal (1-1000): 827
För litet! Gissa tal (1-1000): 835
För litet! Gissa tal (1-1000): 839
För litet! Gissa tal (1-1000): 841
OK efter 9 gissningar.
```


© Init AB

Övning

- **ovn0304/sekvens.py*** Skriv ett program som låter användaren mata in tre värden: start, stop, increment. Det ska fungera enligt följande:

```
$ ./sekv.py
Start: 3
Stop: 10
Increment: 2
3 5 7 9
Start: 15
Stop: 35
Increment: 5
15 20 25 30
…
```

Avsluta ifall increment <= 0 eller stop < start.

© Init AB

Strängfunktionerna split och join

- Ett smidigt sätt att skapa en lista av strängar är split:

```
>>> "Bill Steve Linus Ken".split()
['Bill', 'Steve', 'Linus', 'Ken']
>>> L = "Bill Steve Linus Ken".split()
>>> L
['Bill', 'Steve', 'Linus', 'Ken']
>>> len(L)
4
>>> " och ".join(L)
'Bill och Steve och Linus och Ken'
>>> "12 + 19 + 33 + 27".split(" + ")
['12', '19', '33', '27']
```


© Init AB

Strängfunktionen splitlines

- Dela upp sträng i enskilda rader:

```
>>> NAMN = """Bill Gates
... Steve Jobs
... Linus Torvalds
... Ken Thompson"""
>>> NAMN
'Bill Gates\nSteve Jobs\nLinus Torvalds\nKen Thompson'
>>> NAMN.splitlines()
['Bill Gates', 'Steve Jobs', 'Linus Torvalds', 'Ken Thompson']
```


© Init AB

List comprehensions

- List Comprehensions
  - [*expr* for *iter_var* in *iterable*]
  - [*expr* for *iter_var* in *iterable* if *cond_expr*]

```
>>> [ x*x for x in range(5) ]
[0, 1, 4, 9, 16]

>>> a = [ x*x for x in range(5) ]
>>> [ x*x for x in a ]
[0, 1, 16, 81, 256]

>>> [ x*x for x in a if not x % 2 ]
[0, 16, 256]

>>> [ x*y for x in (1, 2, 3) for y in (2, 3) ]
[2, 3, 4, 6, 6, 9]
```


© Init AB

Sortera listor

```
>>> scores = [ 12, 19, 33, 27, 31 ]
>>> sorted(scores)
[12, 19, 27, 31, 33]
>>> scores
[12, 19, 33, 27, 31]
>>> scores.sort()
>>> scores
[12, 19, 27, 31, 33]
>>>
```


© Init AB

Listor och iteration

```
>>> dudes = [ 'Bill', 'Steve', 'Linus', 'Ken' ]
>>> scores = [ 12, 19, 33, 27 ]
```

- *reversed()*

```
>>> for s in reversed(scores): print(s, end=' ')
27 33 19 12
```

- *enumerate()*

```
>>> for i, dude in enumerate(dudes): print(i, dude)
0 Bill
1 Steve
2 Linus
3 Ken
```

- *zip()*

```
>>> for s, dude in zip(scores, dudes): print(s, dude)
12 Bill
19 Steve
33 Linus
27 Ken
```


© Init AB

Övning

- **ovn0305/tallista.py** Skriv ett program som läser in heltal tills man matar in talet 0. Då ska programmet skriva ut alla talen (utom det sista, 0) på varsin rad, sorterade i växande ordning. Uppdatera sedan programmet så att tal som förekommer flera gånger endast skrivs ut en gång.
- **ovn0306/frekvent.py*** Skriv ett program som läser in text från terminalen tills en tom rad matas in. Sedan ska programmet skriva ut det ord som förekom flest gånger. (Om det finns flera ord som förekommer lika många gånger, så skriv ut det av dem som kommer först i bokstavsordning.)

```
$ python3 prog.py
ett och två
samt tre och fyra

Mest frekvent ord: och
$
```


© Init AB

Lazy evaluation

- Python väntar ofta med att göra beräkningar tills de faktiskt behövs

```
>>> a = [1, 2, 3]
>>> reversed(a)
<list_reverseiterator object at 0x7ff682ed23d0>
>>> list(reversed((a))
[3, 2, 1]
>>> r=reversed(a)
>>> for i in r:
...     print(i)
...
3
2
1
>>> r[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list_reverseiterator' object is not subscriptable
>>>
```


© Init AB

Lazy evaluation: range

- Funktionen range bygger på lazy evaluation

```
>>> L = range(1, 1000000000000)
>>> L[55]
56
>>> for x in L:
...     if x>10: print(); break
...     else: print(x, end=" ")
...
1 2 3 4 5 6 7 8 9 10
>>> L
range(1, 1000000000000)
>>> list(L)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
>>>
```


© Init AB

Diskutera

- Vilka av följande funktioner använder lazy evaluation:
  - range
  - sorted
  - reversed
  - enumerate
  - zip

- Vad är poängen med lazy evaluation?

© Init AB

Funktioner

- Funktioner

```
def times2(x):
    return 2*x

def print_upper(x):
    print(x.upper())
```

- Anropa funktioner

```
>>> times2(5)
10
>>> times2('kalle')
'kallekalle'
>>> print_upper('hej')
HEJ
```


© Init AB

Funktioner – definition och anrop

- Funktioner måste *definieras* innan de *anropas.*
- Detta går bra:

```
def f(x):
    return double(x+1)

def double(x):
    return x+x

print(f(5))
```


© Init AB

Funktioner – definition och anrop

- Detta går inte bra:

```
def f(x):
    return double(x+1)

print(f(5))

def double(x):
    return x+x
```


© Init AB

Doc strings

- Dokumentation

```
def foo():
    "This is a doc string for the foo function"
    return True

>>> foo()
True
>>> help(foo)
Help on function foo in module __main__:

foo()
    This is a doc string for the foo function
```


© Init AB

Inbyggd dokumentation

- Dokumentation av inbyggda funktioner

```
>>> help(sorted)
Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable
  in ascending order.
```


© Init AB

Lokala och globala variabler

- **locvar.py**

```
x = 3
y = 4

def f():
    y = 6
    print("In function: x=", x, ", y=", y, sep='')

f()
print("Outside: x=", x, ", y=", y, sep='')

$ python3 locvar.py
In function: x=3, y=6
Outside: x=3, y=4
$
```


© Init AB

Ändra globala variabler i funktion

- **globvar.py**

```
x = 3
y = 4

def f():
    global y
    x = 5
    y = 6

print("Before: x=", x, ", y=", y, sep='')
f()
print("After: x=", x, ", y=", y, sep='')

$ python3 globvar.py
Before: x=3, y=4
After: x=3, y=6
$
```


© Init AB

”Duck typing”

- Normalt låter man pythons operationer anpassa sig efter operandernas typ (“duck typing”). Om nödvändigt kan man kontrollera t.ex. funktionsargumentens typ explicit enligt nedan:

```
def times2(x):
  if isinstance(x,(int, float, complex)):
      return 2*x
  else:
      return '?'

>>> times2(25)
50
>>> times2('a')
?
```


© Init AB

Funktioner - argument

- Default-argument

```
def addera_moms(pris, faktor = 1.25):
    return pris*faktor

>>> addera_moms(200)
250.0
>>> addera_moms(200, 1.06)
212.0
>>> addera_moms(faktor=1.06, pris=200)
212.0
>>>
```


© Init AB

Funktioner - argument

- Godtyckligt antal argument

```
def bar2(*args, **kw):
    print(args, kw)

>>> bar2(1, 2, 4, x=5, y=6)
(1, 2, 4) {'x': 5, 'y': 6}
```


© Init AB

Funktioner - returvärden

- Flera returvärden - returnera tupel eller lista

```
>>> def func():
...     return (val1, val2)    # Alt: return val1, val2

>>> def func():
...     return [val1, val2]
```

- Inga returvärden - *None*
  - Om inte return anropas returneras None när funktionen avslutas

© Init AB

Övning

- **ovn0307/funktion.py** Skriv en funktion med namnet `maxtal` som tar en lista av tal som parameter och returnerar det största talet i listan. T.ex. ska `maxtal([13, 5, 23, 8])` returnera 23.

- **ovn0308/funktion2.py*** Som ovan, men
  - Om listan är tom ska funktionen returnera None.
  - Om listan innehåller något som inte är `int` eller `float`, så ska funktionen returnera None.
  - Det ska gå att anropa med en tupel eller range, inte enbart en lista.
  - Det ska gå att anropa med en godtycklig sekvens (”lazy evaluation”), t.ex. `maxtal(reversed([5, 55, -5]))`.

- **ovn0309/funktion3.py*** Som ovan, men i stället för att funktionen ska ta ett enda argument som är en lista, ska den anropas med noll eller fler tal som argument. T.ex. ska `maxtal(13, 5, 23, 8)` returnera 23.

© Init AB

Pythonobjekt

- Ett objekt har: *typ*, *värde* och *identitet*

```
>>> a = [1, 2, 3]
>>> b = a
>>> id(a)                 # id() ger identiteten,
136046976
>>> id(b)
136046976
>>> a is b                # is ger sant om det är samma objekt
True
>>> c = [1, 2, 3]
>>> c is a
False
>>> vars()
{'__name__': '__main__', '__doc__': None,
  '__package__': None, 'a': [1, 2, 3], 'b': [1, 2, 3],
  'c': [1, 2, 3]}
```


© Init AB

Ta bort variabler

- Borttagning av variabler görs med *del*
  - automatiska minneshanteringen (skräpsamling) kan ta vid

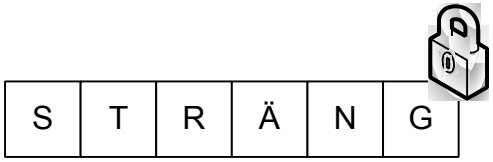
```
>>> a = 1.2
>>> b = a
>>> del a
>>> b
1.2
>>> 'b' in vars()
True
>>> 'a' in vars()
False
>>>
```


© Init AB

Immutable, Mutable

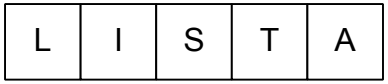
- Immutable - objekt vars värden inte kan förändras
  - Exempel: Strängar

```
>>> s = 'STRÄNG'
>>> print(s[1])
'T'
>>> s[1] = 'P'
Traceback...
...
TypeError: 'str' object does not support item assignment
```



- Mutable - objekt vars värden kan förändras
  - Exempel: Listor

```
>>> li = list('LISTA')
>>> li[0] = 'P'
>>> print(li)
['P', 'I', 'S', 'T', 'A']
```

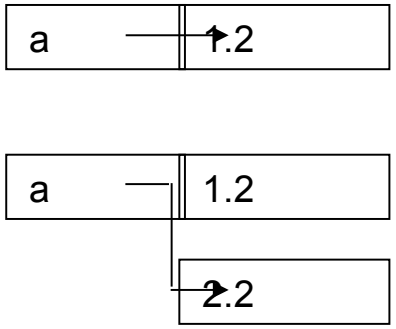


© Init AB

Immutable

- Immutable - Tal, strängar, tupler, frozenset

```
>>> a = 1.2
>>> id(a)
136427476
>>> a += 1
>>> id(a)
136427444
```




```
>>> b = a
>>> id(b)
136427444
```

  - Just nu råkar a och b peka på samma minnesutrymme men så fort b sätts till ett nytt värde kommer b att peka på ett nytt minnesutrymme medan a behåller sitt gamla

© Init AB

Mutable

- Mutable - objekt vars värden kan förändras
  - Listor, dictionaries, set

```
>>> a = [1, 2, 3]
>>> id(a)
136011360
>>> b = a
>>> id(b)
136011360
>>> a += [4, 5]
>>> print(a)
[1, 2, 3, 4, 5]
>>> print(b)
[1, 2, 3, 4, 5]
>>> id(a)
136011360
```

  - *a* och *b* refererar nu till samma minnesutrymme. Om *a* förändras kommer även *b* att förändras
  - För att göra en ny (ytlig) kopia av *a* - skriv: `b = list(a)`

© Init AB

"Funktionella" funktioner

- *lambda* - anonyma funktioner

```
>>> a = lambda x: x*x*x
>>> a(3)
27
```

- *map* - applicera funktion på varje element i lista.
  - Returnerar ett iteratorobjekt

```
>>> a = lambda x: x*x*x     # Eller: def a(x): return x*x*x
>>> list(map(a, [1, 2, 3, 4]))
[1, 8, 27, 64]
```

  - map kan även ta flera listor som argument. Då appliceras en funktion f(x,y) på lista1[0],lista2[0] osv.
- *filter* - filtrera bort element (funktionen ska returnera *bool*)

```
>>> filter(lambda x: isinstance(x, int), [1, 2, 3, '23', 'a'])
[1, 2, 3]
```


© Init AB

Sorteringskriterium

- Sorteringsordning bestäms av en ”poängsättningsfunktion”

```
>>> L=[2, 3, 5, 7, 11, 13, 17, 19, 23]
>>> def siffersumma(x):
...     return sum([int(d) for d in str(x)])
...
>>> siffersumma(342)
9
>>> sorted(L, key=siffersumma)
[2, 11, 3, 13, 5, 23, 7, 17, 19]
>>> sorted(L, key=lambda x: x%10)
[11, 2, 3, 13, 23, 5, 7, 17, 19]
>>> sorted(L, key=lambda x: str(x))
[11, 13, 17, 19, 2, 23, 3, 5, 7]
>>> L.sort(key=siffersumma, reverse=True)
>>> L
[19, 17, 7, 5, 23, 13, 3, 2, 11]
>>>
```


© Init AB

Funktioner - dekoratorer (överkurs)

- Modifierar en befintlig funktions egenskaper:

```
>>> def debug(f):
...     def wrapped_func(x):
...         print('Funktionen', f.__name__, 'anropades')
...         return f(x)
...     return wrapped_func
...
>>> def func1(x):
...     x += 1
...     return x
...
>>> func1 = debug(func1)
>>> func1(25)
Funktionen func1 anropades
26
```

Funktioner - dekoratorer (överkurs)

- Numera skriver man på följande sätt

```
>>> def debug(f):
...     def wrapped_func(x):
...         print('Funktionen', f.__name__, 'anropades')
...         return f(x)
...     return wrapped_func
...
>>> @debug
... def func1(x):
...     x += 1
...     return x
...
>>> func1(25)
Funktionen func1 anropades
26
```

Funktioner - att tänka på

- Defaultargument sätts bara vid definitionen => en gång

```
>>> def add_element(a, L=[]):
...     L.append(a)
...     return L
...
>>> add_element(11, [2, 3, 5, 7])
[2, 3, 5, 7, 11]
>>> add_element(4)
[4]
>>> add_element(21, [0, 1, 1, 2, 3, 5, 8, 13])
[0, 1, 1, 2, 3, 5, 8, 13, 21]
>>> add_element(100)
[4, 100]
>>>
```


© Init AB

Stil-guide

- Indentering med fyra mellanslag per indenteringsnivå
- Editorer med python-mod omvandlar ofta automatiskt <tab> till fyra mellanslag. **Använd en editor med pythonstöd!**
- Blanda inte <tab> och mellanslag
- Max 79 tecken per rad - "\" markerar fortsättning på en rad
- Mellanslag för ökad läsbarhet

```
x = 2
if x == 2:
print(x, y)
```

- Modulnamn med små bokstäver
- Klassnamn enligt *CapWords*. Objekt enligt my*obj* eller *myObj*
- Funktionsnamn, globala variabelnamn, metodnamn, instansvariabler: *sma_bokstaver* med "_" mellan orden
- För detaljer se: http://www.python.org/dev/peps/pep-0008