Programmeringsteknik I

Föreläsning 2: Funktioner och moduler

Johan Öfverstedt

Förra gången

- Uttryck
- Variabler
- Grundläggande operatorer
- Villkorssatser (if)
- Slingor/loopar (while och for)

Dagens föreläsning

- Funktioner och procedurer
  - Definition av funktioner
  - Anrop av funktioner
  - Terminologi
  - Aktuella parametrar / argument
  - Parameteröverföring
  - Returvärden, multipla returvärden med tupler
  - Funktioner med funktioner som argument för flexibla och kraftfulla lösningar
- Moduler
  - Importering av/från moduler
  - Definition av moduler
  - Moduler som en abstraktion
- Generatorer

Funktioner/procedurer i Python

En funktion defineras:

```
def funktionsnamn(parameter1, parameter2, …):
    funktionens kropp

def fn(a, b):
    return a * b
```

Funktioner med en parameter

```
def square(x):
    return x * x

y = square(2)
print(y)
>> 4
y = square(2.5)
print(y)
>> 6.25
```

Funktioner med en parameter

```
def factorial(n):
    assert(type(n) is int)
    res = 1
    for i in range(1, n+1):
        res = res * i
    return res

y = fac(5)
print(y)
>> 120
```

Funktioner med en parameter

```
def factorial(n):
    assert(type(n) is int)
    res = 1
    for i in range(1, n+1):
        res = res * i
    return res

y = fac(5)
print(y)
>> 120
```

```
y = square(5.0)
print(y)
>> Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fac
AssertionError
```

Funktioner med två parametrar

```
a = 2.0
b = 3.0

def mystery(a, b):
    if b < a:
        return b
    else:
        return a

print(mystery(b, a))
>> ???
```

Vad händer när koden körs?

1) '3.0'
2) '2.0’
3) 'None'
4) Det blir fel

Funktioner med två parametrar

```
a = 2.0
b = 3.0

def minimum(a, b):
    if b < a:
        return b
    else:
        return a

print(minimum(b, a))
>> 2.0
```

Vad händer när koden körs?

1) '3.0'
2) '2.0’
3) 'None'
4) Det blir fel

Parameteröverföring - exempel 1

```
y = 1

def f(x):
    x = x + 1
    return x

z = f(y)

print(f'{y}, {z}')
>> ???
```

Vad händer när koden körs?

1) '1, 2'
2) '2, 2'
3) '2, 1'
4) Det blir fel

Parameteröverföring - exempel 1

```
y = 1

def f(x):
    x = x + 1
    return x

z = f(y)

print(f'{y}, {z}')
>> 1, 2
```

Vad händer när koden körs?

1) '1, 2'
2) '2, 2'
3) '2, 1'
4) Det blir fel

Parameteröverföring - exempel 2

```
x = 1

def f(x):
    x = x + 1
    return x

y = f(x)

print(f'{x}, {y}')
>> ???
```

Vad händer när koden körs?

1) '1, 2'
2) '2, 2'
3) '2, 1'
4) Det blir fel

Parameteröverföring - exempel 2

```
x = 1

def f(x):
    x = x + 1
    return x

y = f(x)

print(f'{x}, {y}')
>> 1, 2
```

Vad händer när koden körs?

1) '1, 2'
2) '2, 2'
3) '2, 1'
4) Det blir fel

Parameteröverföring - exempel 3

```
def swap_first_two(list):
    list[0], list[1] = list[1], list[0]
    return list

x = [1, 2, 3, 4, 5]

print(x)
>> [1, 2, 3, 4, 5]
y = swap_first_two(x)
print(x)
>> ??
```

Vad händer när koden körs?

1) '[1, 2, 3, 4, 5]'
2) '[2, 1, 3, 4, 5]'
3) 'None'
4) Det blir fel

Parameteröverföring - exempel 3

```
def swap_first_two(list):
    list[0], list[1] = list[1], list[0]
    return list

x = [1, 2, 3, 4, 5]

print(x)
>> [1, 2, 3, 4, 5]
y = swap_first_two(x)
print(x)
>> ??
```

Vad händer när koden körs?

1) '[1, 2, 3, 4, 5]'
2) '[2, 1, 3, 4, 5]'
3) 'None'
4) Det blir fel

Parameteröverföring med namn i anropet

Ta funktionen:

```
def divide(denominator, numerator):
    return numerator / denominator
```

Man kan nu anropa `divide`:

```
print(divide(denominator=2.0, numerator=1.0))
>> 0.5
```

Parameteröverföring med standardvärden

Ibland kan det finnas standardvärden som vi oftast vill använda när vi anropar en funktion, men vill ändå att det ska finnas möjlighet att ange ett annat värde när det behövs. Då kan vi definera en funktion:

```
def divide(denominator, numerator = 1.0):
    return numerator / denominator

print(divide(4.0))
>> 0.25
print(divide(4.0, 2.0))
>> 0.5
```

Ducktyping! If it walks like a duck...

Parametertyper i Python

Precis som variabler i Python har parametrar inte någon specificerad datatyp.

Funktioner kan skrivas en gång, men anropas med parametrar av olika datatyper. (Vi såg i det första exemplet square, som enbart kräver att multiplikation är definerat för argumentet.)

```
def square(x):
    return x * x
```

Detta gör att man kan skriva väldigt generella funktioner utan att göra mer specifika antaganden än nödvändigt.

Returvärden

I Python, med sitt dynamiska typsystem, är det fullt möjligt att skriva kod som:

```
def f(x):
    if x > 0.5:
        return 2 * x
    else:
        return f'{x}'
```

men det är dålig stil. Den som anropar y = f(x), måste då först börja med att testa vilken datatyp y har, innan man vet vad man kan göra med returvärdet. Om returvärdet är relaterat till parametrarnas typ kan det vara okej att returnerna olika datatyper.

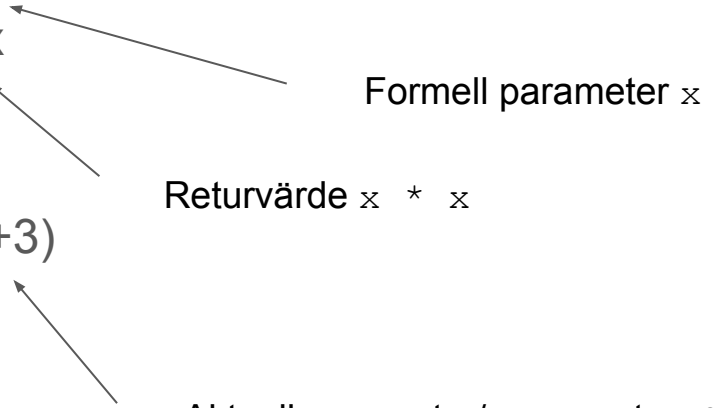
Funktioner som inte returnerar värden (procedurer)

```
def print_balance(name, balance):
    print(f'{name} has {balance} in their bank account.')

print_balance('Johan', 512)
>> Johan has 512 in their bank account.
r = print_balance('Johan', 512)
>> Johan has 512 in their bank account.
print(type(r))
>> <class 'NoneType'>
print(r)
>> None
```

Alla funktioner returnerar alltså ett värde: **None** ifall ingen explicit return-sats nåtts innan funktionen når sitt slut.

Terminologi

```
def square(x):
    return x * x

y = 1
z = square(y+3)
```



Formell parameter - en variabler som får det inskickade argumentet.
Aktuell parameter/argument - värdet som skickas in vid ett konkret anrop.

Funktioner med funktioner som parametrar

```
def bubble_sort(lst, cmp = less):
    n = len(lst)-1
    for n in range(n, 0, -1):
        for i in range(n):
            if cmp(lst[i+1], lst[i]):
                lst[i+1], lst[i] = lst[i], lst[i+1]


xs = [19, 1, 4, 8, 2, 9, 5, 13, 7]
```

Funktioner med funktioner som parametrar

```
def less(x, y):
    return x < y

def greater(x, y):
    return y < x

print(type(less))
>> <class 'function'>
```

Funktioner med funktioner som parametrar

```
def bubble_sort(lst, cmp = less):
    n = len(lst)-1
    for n in range(n, 0, -1):
        for i in range(n):
            if cmp(lst[i+1], lst[i]):
                lst[i+1], lst[i] = lst[i], lst[i+1]
```

```
xs = [19, 1, 4, 8, 2, 9, 5, 13, 7]
print(xs)
>> [19, 1, 4, 8, 2, 9, 5, 13, 7]
bubble_sort(xs, less)
print(xs)
>> ??
```

Bubble sort debugging

```
def bubble_sort_debug(lst, cmp = less):
    n = len(lst)-1
    for n in range(n, 0, -1):
        for i in range(n):
            if cmp(lst[i+1], lst[i]):
                lst[i+1], lst[i] = lst[i], lst[i+1]
        print(lst)
```

[3, 2, 8, 1, 5, 9, 0]
After 1 iterations:
[2, 3, 1, 5, 8, 0, 9]
After 2 iterations:
[2, 1, 3, 5, 0, 8, 9]
After 3 iterations:
[1, 2, 3, 0, 5, 8, 9]
[1, 2, 0, 3, 5, 8, 9]
[1, 0, 2, 3, 5, 8, 9]
[0, 1, 2, 3, 5, 8, 9]

Funktioner med funktioner som parametrar

```
def bubble_sort(lst, cmp = less):
    n = len(lst)-1
    for n in range(n, 0, -1):
        for i in range(n):
            if cmp(lst[i+1], lst[i]):
                lst[i+1], lst[i] = lst[i], lst[i+1]
```

```
xs = [19, 1, 4, 8, 2, 9, 5, 13, 7]
print(xs)
>> [19, 1, 4, 8, 2, 9, 5, 13, 7]
bubble_sort(xs, less)
print(xs)
>> [1, 2, 4, 5, 7, 8, 9, 13, 19]
```

Funktioner med funktioner som parametrar

```
def bubble_sort(lst, cmp = less):
    n = len(lst)-1
    for n in range(n, 0, -1):
        for i in range(n):
            if cmp(lst[i+1], lst[i]):
                lst[i+1], lst[i] = lst[i], lst[i+1]
```

```
xs = [19, 1, 4, 8, 2, 9, 5, 13, 7]
print(xs)
>> [19, 1, 4, 8, 2, 9, 5, 13, 7]
bubble_sort(xs, greater)
print(xs)
>> [19, 13, 9, 8, 7, 5, 4, 2, 1]
```

Funktioner med flera returvärden

I Python kan funktioner lätt returnera flera värden genom att returnera en tupel:

```
def quad_equation(p, q):
    discriminant = p*p - 4*q
    if discriminant >= 0:
        d = math.sqrt(discriminant)
        x1 = (-p + d)/2.0
        x2 = (-p - d)/2.0
        return x1, x2
    else:
        return None  #
```

Funktioner med flera returvärden

I Python kan funktioner lätt returnera flera värden genom att returnera en tupel:

```
roots = quad_equation(-3, 2)
print(roots)
>> (1.0, 2.0)
r1, r2 = quad_equation(-3, 2)
print(r1, r2)
>> 1.0 2.0
```

Return avslutar funktionen

`return` avslutar funktionen (och tilldelar returvärdet).

```
def abs_value_with_print(x):
    if(x >= 0.0):
        return x
    print(f'{x}')
    return -x
```

```
print(abs_value_with_print(2.0))
>> 2.0
```

```
print(abs_value_with_print(-2.0))
>> -2.0
2.0
```

Fel (som inte fångas) avslutar funktionen

```
def divide(denominator, numerator):
    if denominator == 0:
        raise ValueError('Zero denominator in division.. Booo')
    return numerator / denominator

divide(0, 2.0)
>> Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Zero denominator, boo
```

Funktioner inuti funktioner

```
def smooth(list, alpha):
    def comp_y(y, x):
        return y * alpha + (1.0 - alpha) * x
    y_val = 0.0
    for i in range(len(list)):
        y_val = comp_y(y_val, list[i])
        list[i] = y_val
```

Interna funktioner kan vara användbara om man vill definera och namnge en operation som används inuti den yttre funktionen.

Metoder - funktioner associerade med objekt

Metoder är funktioner som hör ihop med objekt (till skillnad från fristående funktioner). Ni har sett dessa:

`t1.forward(30)`
forward flyttar sköldpaddan som variabeln `t1` refererar till 30 bildpunkter framåt.

Objektet före punkten är det objekt som metoden anropas för.
`t2.forward(30)`
flyttar sköldpaddan som variabeln `t2` refererar till (vilket kan antingen vara samma padda som `t1` eller en annan.) Om metoden inte är definerad (för datatypen som variabeln före punkten är av) får vi ett felmeddelande.

Moduler

En modul är en Python-fil (.py) som innehåller en samling definitioner och variabler. En modul kan köras som ett skript, eller importeras till en annan modul/skript.

Genom att anropa t.ex.
python my_program.py

så kommer my_program.py att köras som ett skript.

Moduler med funktioner

Som exempel, ta en modul (my_math.py) med en enda funktion

```
def factorial(n):
    res = 1
    for i in range(1, n+1):
        res = res * i
    return res
```

Importering av definitioner

I en annan modul (my_script.py), kan vi nu komma åt ‘factorial’ från ‘my_math.py’ på flera sätt:

1)

```
from my_math import factorial
print(factorial(7))
>> 5040
```

2)

```
import my_math as mm
print(mm.factorial(7))
>> 5040
```

Att importera en modul är att köra ett skript

Om man lägger in kod som körs (utanför funktioner) i en modul och importerar denna modul, kommer koden att köras. Exempel för en modul test_module.py:

```
x = 5
print(x)
```

Sedan importerar vi test_module.py från en annan modul:

```
import test_module as tm
>> 5
```

Att importera en modul är att köra ett skript

Exempel körs!

Man vill alltså sällan inkludera kod utanför funktioner som körs i moduler, ifall någon vill/behöver importera modulen i en annan modul/skript utan att den koden körs. En lösning om man vill att en modul ska vara både modul och skript:

```
def main():
    print(5)

if __name__ == '__main__':
    main()
```

Moduler som abstraktioner

Ett vanligt sätt att skapa återanvändbara kodpaket är att samla funktioner (och klasser, som Turtle) i moduler.

Genom att samla all kod som hör till ett delproblem i en modul kan man strukturera sitt program så att det är lätt att hitta i koden.

Eftersom funktioner i en modul måste importera funktioner från andra moduler för att de ska synas, och för att man inte vill ha för många moduler, är det bra att samla relaterade begrepp.

Moduler exempel på en indelning - ex: py_alpha_amd

För ett riktigt exempelprojekt med bildregistrering (alignment) av digitala bilder:

Paket:
distances (alpha_amd.py, symmetric_amd_distance.py, ...)
filters (filt.py)
generators (mask_generators.py, noise_generators.py)
optimizers (gd_optimizer.py, adam_optimizer.py)
transforms (affine_transform.py, rigid_transform.py, rotate_2d_transform.py, ...)

Moduler exempel på en indelning - ex: py_alpha_amd

transforms (affine_transform.py, rigid_transform.py, rotate_2d_transform.py, ...) innehåller en uppsättning olika geometriska transformationer. De grupperas naturligt sett i ett paket bestående av ett antal moduler.

Transformationerna är väldigt lika varandra och beror i ett flertal fall på varandra vilket gör att dem hör ihop.

Allting relaterat till affina transformationer samlas logiskt nog i affine_transform.py. Om man importerar denna modul finner man allting som är relevant för sådana transformationer. Man behöver inte importera 10 olika moduler.

Generatorer - ett exempel med primtalsgenerering

```
def primality_test(x):
    for i in range(2, x):
        # if x is divisible by i, x is not prime
        if x % i == 0:
            return False
    return True
def prime_list(n):
    list = []
    for x in range(2, n+1):
        if primality_test(x):
            list.append(x)
    return list
```

Generatorer - ett exempel med primtalsgenerering

```
def primality_test(x):
    for i in range(2, x):
        # if x is divisible by i, x is not prime
        if x % i == 0:
            return False
    return True

def prime_generator(n):
    for x in range(2, n+1):
        if primality_test(x):
            yield x
```

yield fryser funktionens exekvering och returnerar x. När nästa värde efterfrågas fortsätter funktionen på nästa rad tills ytterligare ett värde kan returneras eller tills funktionen returnerar.

Generatorer - ett exempel med primtalsgenerering

```
for p in prime_generator(1000):
    print(p, end=' ')
```

```
>> 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193
197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307
311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421
431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547
557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659
661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797
809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929
937 941 947 953 967 971 977 983 991 997
```

Generatorer - ett exempel med primtalsgenerering

```
# skapa en generator
pgen = prime_generator(100)
print(next(pgen))
print(next(pgen))
print(next(pgen))
print('Hello')
print(next(pgen))
print(next(pgen))
```

```
>> 2
3
5
Hello
7
11
```

Oändlig lista med generatorer

```
def prime_generator_infinite():
    x = 2
    while True:
        if primality_test(x):
            yield x
        x = x + 1
```

Oändlig loop. Om det här var en funktion skulle den aldrig returnera.

Eftersom generatorn fryser sin exekvering vid yield kan vi här definera en oändlig sekvens med primtal, utan att räkna upp dem alla. Varje primtal beräknas bara vid efterfrågan.

Generatorer av oändliga listor

```
pgen_inf = prime_generator_infinite()
for i in range(1000):
    print(next(pgen_inf), end=' ')
```

```
>> 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109
113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239
241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379
383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521
523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661
673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827
829 839 ......
```