Kort om programmering i PYTHON

1 Inledning

Redan i första laborationen gjorde ni ett litet program. Ni skrev en script eller skriptfil som beräknade summan

$$\sum_{i=1}^{5} i^2 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2$$

med en programkod som kanske såg ut så här

```
from pylab import *

s=0
for i in arange(1,6): s=s+i**2
print('Summan är', s)
```

Nu skall vi lära oss lite mer om programmering. Vi börjar med att se på skript- och funktionsfiler som vi använder för att ge struktur åt program, därefter ser vi hur vi kan styra ut- och inmatning av data för att avslutningsvis se på kontrollstrukturer som `if`-, `for`- och `while`-satser med vilka vi styr flödet genom programkoden.

2 Egendefinierade funktioner

Vi har sett lite funktioner. Nu måste vi bli lite grundligare.

2.1 Skriptfil

En script eller skriptfil är en textfil som innehåller det man skulle kunna skriva direkt i Console. Om man använde SPYDER kör man sedan innehållet i scriptet genom att trycka på den gröna pilen högt upp i fönstret. Namnet på ett pythonscript har tillägget `.py` för att skilja det från andra filer.

Programsatserna i en skriptfil opererar *globalt* på variablerna i arbetsarean (de som finns i Variable explorer).

Alla utskrifter från programmet skrivs som standard i Console, liksom alla felmeddelanden. Man kan också styra utskrifter av beräkningsresultat till en fil, se hjälptexterna vid behov.

Editorn i SPYDER markerar koden med olika färger för att visa vad som är kommentarer, nyckelord, textsträngar, etc., och har flera funktioner för att underlätta vårt arbete.

2.2 Funktioner

En funktion i programmering är ett block av satser som sätts ihop och som sedan kan anropas genom att man anropar funktionen. I PYTHON finns många inbyggda funktioner. T.ex. funktionen `print()` som ju skriver ut ett värde i Console. Vi har också använt flera funktioner från paketen `NumPy` och `Matplotlib`, t.ex. `sin()` för att beräkna sinus och `plot()` för att rita en figur.

I PYTHON finns också möjlighet att skapa egna funktioner. Det finns ett par olika sätt att göra detta på men vi kommer att koncentrera oss på ett sätt.

När man skriver så pass små program som vi gör definerar man ofta de egendefinerade funktionerna högst upp i ett script och fyller på med programkod längre ner i scriptet där man anropar funktionerna. I laboration 1 avsnitt 6 finns föjande exempel

```
from pylab import *
from scipy.optimize import newton

def min_fun(x):
    y=x**3-cos(x)
    return y

# Rita en figur
x=linspace(-1.5,1.5)
y=min_fun(x)
plot(x,y)
grid()
```

Efter raderna som importerar de olika paketen defineras en funktion som heter `min_fun`. (Funktionen anropas sedan på andra raden efter kommentaren `# Rita en figur`).

När man definerar en funktion i PYTHON skriver man först ordet `def`, sedan namnet på funktionen (`min_fun` i exemplet). Efter namnet och inom parentes räknar man upp namnen på ev. inparametrar (`x` i exemplet), sedan följer ett kolon (`:`). Funktionskroppen, de satser som ska utföras när funktionen anropas skrivs ett stycke in från vänstermarginalen. I exemplet ovan har man skrivit de två raderna som tillhör funktionskroppen 4 blanksteg in. Alla de andra raderna i programmet har skrivits med start i vänstermarginalen, de tillhör inte funktionskroppen.

När man kör koden i scriptet ovan kommer exekveringen att börja med raden

```
x=linspace(-1.5,1.5)
```

I anropet `y=min_fun(x)`, som står på nästa rad, utförs de två raderna som står efter `def min_fun(x):`. Sedan utförs anropet till `plot` och till sist anropet `grid()`.

En viktig poäng med att definera en egen funktion är att så fort man vill ha de två raderna i funktionen utförda anropar man funktionen. När man har funktioner som innehåller flera rader blir programkoden mycket tydligare om man lyfter ut kodavsnitt som man anropar, istället för att hela tiden skriva alla raderna funktionen innehåller så fort man behöver utföra dem. I exemplet ovan är funktionen såpass kort, så vi hade lika gärna kunnat ersätta raden `y=min_fun(x)` med `y=x**3-cos(x)`. Programmet hade blivit lika tydligt.

Ett annat viktigt användningsområde för egendefinerade funktioner är att man kan använda dem som argument till andra funktioner. I laboration 1 avsnitt 6 använder man den egendefinerade funktionen som argument i anropet till `newton` när man söker efter nollstället.

```
z=newton(min_fun,1)
print(z)
```

I anropet till `newton` ovan kan man *inte* skriva `y=x**3-cos(x)` istället för `min_fun`.

Det finns en viktig detalj att känna till om egendefinierade funktioner. Variabeln `y` som skapas på första raden i funktionen `min_fun` *finns bara inne i funktionen*. Man säger att den är *lokal* i funktionen. Det `y` som skapas med raden `y=min_fun(x)` i anropet till funktionen är ett annat `y`. Raden `return y` som står på andra raden i `min_fun` betyder att innehållet i funktionens `y` ska skickas ut från funktionen. Värdena skickas till anroparen. Det innebär att man t.ex. kan skriva

```
f=min_fun(x)
```

i anropet av funktionen. Innehållet från `y` i funktionen placeras då i variablen `f`. (Om man anropar funktionen med `f=min_fun(x)` måste man också ändra anropet till `plot` på nästa rad till `plot(x,f)`. Det finns ju i såfall inte längre något `y` utanför funktionen).

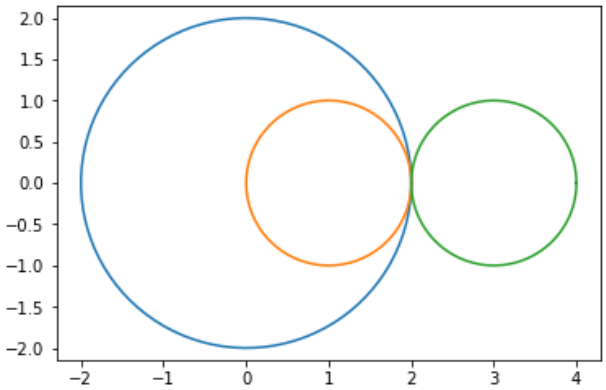
Exempel 1. Vi vill rita cirklar med olika medelpunkt (a, b) och radie r. Vi gör en funktion med namnet `cirkel.m` med a, b och r som indata och två vektorer x och y som utdata enligt

```
from pylab import *

def cirkel(a,b,r):
   t=linspace(0,2*pi,100)
   x=a+r*cos(t)
   y=b+r*sin(t)
   return (x,y)
```

I de två vektorerna har vi koordinater för punkter jämnt fördelade på cirkeln. Vi ritar några cirklar med

```
(x,y)=cirkel(0,0,2)
plot(x,y)
axis('equal')
(x,y)=cirkel(1,0,1)
plot(x,y)
(x,y)=cirkel(3,0,1)
plot(x,y)
```



Uppgift 1(a). Definera funktionen `cirkel` som ovan. Pröva funktionen, rita någon cirkel. Hur långa är vektorerna som funktionen ger som resultat? Ändra er funktion så att ni får 30 punkter på cirkeln, eller varför inte 5 punkter. Rita de "nya" cirklarna.
(b). Om vi lägger till kommentarer direkt ovanför funktionsdefinitionen enligt

```
# Hjälptext, det man skriver direkt före
# funktionsdefinitonen, kommer att skrivas
# ut som hjälptext om man använder help i Console.
def cirkel(a,b,r):
    t=linspace(0,2*pi,100)
    x=a+r*cos(t)
    y=b+r*sin(t)
    return (x,y)
```

och skriver `help(cirkel)` i Console så kommer hjälptexten skrivas ut. Vilken hjälptext tycker ni är bra till vår funktion? Skriv in er hjälptext i funktionsfilen och pröva med `help`.

3 In- och utmatning

I samband med programmering behöver vi olika former av in- och utmatning. Ibland kan man t.ex. vilja ha inmatning till ett script medan det utförs (körs) eller att beräkningsresultat skall skrivas ut med ett visst format, t.ex. ett visst antal decimaler.

Med `input` kan vi mata in ett värde enligt

```
antal=input('Ange antal kast: ');
```

Vi skriver in raden i Editorn och kör programmet i Console. Det kan se ut så här:

```
Ange antal kast 5
```

När Python kommer till `input`-kommandot så skrivs texten `'Ange antal kast:'` ut och programmet väntar på raden tills vi skriver ett svar, i vårt fall 5, variabeln `antal` ges detta värde och programmet fortsätter sedan med nästa kommando.

Efter raden har körts finns det värde som angavs sparat i variabeln `antal`.

```
>>> antal
5
```

Man behöver inte bara mata in tal utan kan mata in t.ex. ord och texter:

```
Ange antal kast fem
```

Här har man matat in ordet `fem`. Variabeln `antal` kommer nu att ha värdet `'fem'` istället.

```
>>> antal
'fem'
```

Vill vi bara skriva ut variabelvärden kan vi använda `print`. Vi skriver följande i Editorn

```
antal=input('Ange antal kast: ')
print('Du skrev',antal)
```

När vi kör i Console kan det se ut så här:

```
Ange antal kast: 8
Du skrev 8
```

4 Kontrollstrukturer

Nu skall vi lära oss lite mer om kontrollstrukturer som `if`-, `for`- och `while`-satser med vilka vi styr flödet genom programkoden.

4.1 Logiska uttryck och operationer

Vi kommer behöva använda logiska villkor av typen $x > 5$. Detta uttryck är sant om ett tal vi betecknar med x är strängt större än talet 5, annars falskt. Vi skriver detta uttryck i PYTHON som `x>5` helt enkelt. Det logiska värdet sant beskrivs i PYTHON av värdet `True` (talet 1) och falskt beskrivs av `False` (talet 0).

Relationsoperatorerna $<$, $\leq$, $>$, $\geq$, $=$ och $\neq$ skrivs i PYTHON med `<`, `<=`, `>`, `>=`, `==` respektive `!=`. Observera dubbla likhetstecken i PYTHON för att beteckna (logisk) likhet, enkelt likhetstecken används ju för tilldelning, dvs. att ge en variabel ett värde. Vidare har vi ibland nytta av funktionerna `any` och `all` från `NumPy` som arbetar på vektorer (se gärna hjälptexten i PYTHON).

De logiska operatorerna "och", "eller" samt "negation" skrivs i PYTHON `and`, `or` respektive `not`.

4.2 Villkorssatser

Det allmänna utseendet på en `if`-sats är något av följande alternativ

```
if uttryck:
    sats
```

```
if uttryck:
    sats
else:
    sats
```

```
if uttryck:
    sats
elif uttryck:
    sats
else:
    sats
```

där vi kan ha godtyckligt många `elif` i det sista alternativet. Med `uttryck` avser vi ett logiskt uttryck av den typen vi nämnde ovan och `sats` består av en eller flera rader som ska utföras om uttrycket i `uttryck` är sant. Observera att raderna måste skrivas en bit in från vänstermarginalen.

Exempel 3. Vi har två värden a och b och vill att c skall ges det största av dessa värden. Detta kan göras med följade kod

```
if a<b:
    c=b
else:
    c=a
```

Om $a < b$ är sant, så är b störst och c ges värdet av b, annars är a störst och c ges värdet av a.

Exempel 4. Vi skall göra en funktion som beräknar medianen av värdena i en vektor. Medianen är det mittersta värdet i storleksordning om vektorn har udda antal element, och medelvärdet av de två mittersta (i storleksordning) om antal element är jämnt.

Längden av en vektor (antal element) ges av `size` och elementen i en vektor sorteras i storleksordning av `sort`.

Operatorn `%` ger resten vid heltalsdivision. T.ex. får `n%2` värdet 0 om `n` är att jämnt heltal och värdet 1 om `n` är ett udda heltal.

När man indexerar i ett fält (`array`) får man bara använda heltal. Operatorn `/` som används för division svarar med ett reellt tal. För att få heltalsdivision använder man istället operatorn `//`.

Nu skriver vi vår funktion

```
from pylab import *

# m = min_median(v) beräknar medianen av elementen i vektorn v
def min_median(v):
    s=sort(v)               # s sorterad version av v
    n=size(v)               # n antal element i v
    if n%2==0:              # n jämnt
        m=(s[(n-2)//2]+s[n//2])/2
    else:                   # n udda
        m=s([(n-1)//2])
    return m
```

som vi skriver in i Editorn. Vi laddar in funktionen i Console genom att köra filen med den gröna pilen högst upp på desktopen. I Console kan vi se på hjälptexten med

```
>>> help(min_median)
min_median(v)
    # m = min_median(v) beräknar medianen av elementen i vektorn v
```

och tar medianen av en slumptalsvektor (`rand` ger slumptal mellan 0 och 1)

```
>>> v=rand(6)
>>> v
array([0.10489515, 0.66827307, 0.54949596, 0.46574508, 0.72426309,
       0.32999641])
>>> m=min_median(v)
>>> m
0.5076205222066289
```

Nu finns det en funktion `median` i `NumPy` för att bilda medianen som vi givetvis använder istället när vi bestämmer medianvärdet av element i en vektor. Pröva `help median` och beräkna medianen med ett anrop till `median`.

4.3 Repetitionssatser

För att upprepa en grupp av satser flera gånger används `for`-satser eller `while`-satser. Vet vi på förhand hur många gånger upprepningen skall ske, så är normalt en `for`-sats att föredra i annat fall är en `while`-sats lämpligare.

4.3.1 `for`-satser

Det allmänna utseendet på en `for`-sats är

```
for variabel in uttryck:
    satser
```

Här är `uttryck` en vektor av tal eller ett uttryck som bygger upp en sådan vektor. Successivt kommer `variabel` tilldelas värdena i `uttryck` i tur och ordning och samtidigt kommer alla `satser` att utföras. En gång för varje värde som `elem` ges.

Vi har tidigare använt föjande variant:

```
for variabel in arange(start,slut,steg):
    satser
```

där `arange(start,slut,steg)` bildar listan med tal som startar med `start`, stegar `steg` och slutar så nära `slut` som det går. Om man utlämnar `steg` får man steget 1 − så när man vill ha steget 1 brukar man istället skriva

```
for variabel in arange(start,slut):
    satser
```

Vi har redan sett på några summor som vi beräknat med `for`-sats och här kommer några ytterligare exempel.

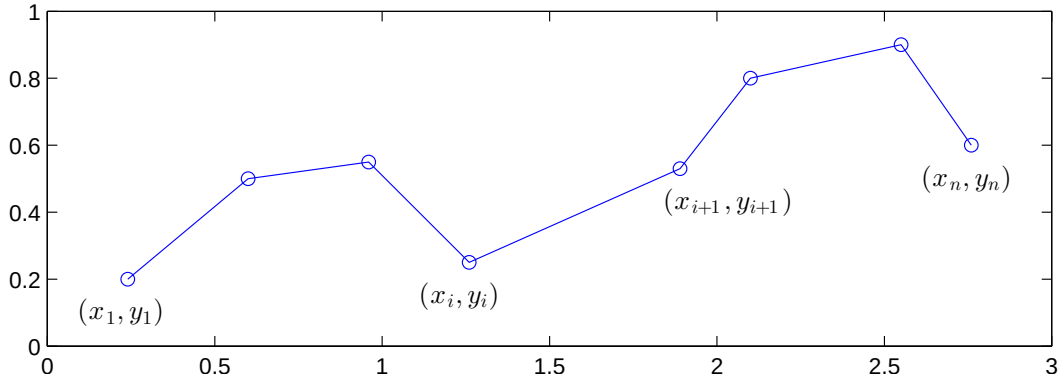
Exempel 5. Vi kan beräkna summan $s = 1 + 3 + 5 + 7 + 9 + 11 + 13$ med en programkod som kanske ser ut så här

```
s=0
for i in arange(1,14,2):
    s=s+i
print(s)
```

Vi kan se det som att vi har en låda `s` som vi samlar värden (termer) i. Först ser vi till att lådan från början är tom med `s=0`.

När `for`-satsen utförs kommer `i` successivt få värdena $1, 3, 5, \cdots, 13$. För varje värde som `i` får kommer `s=s+i` utföras, dvs. vi kommer lägga aktuellt värde på `i` till det vi redan har i lådan `s`. När `for`-satsen är klar skrivs värdet på `s` ut med `print`.

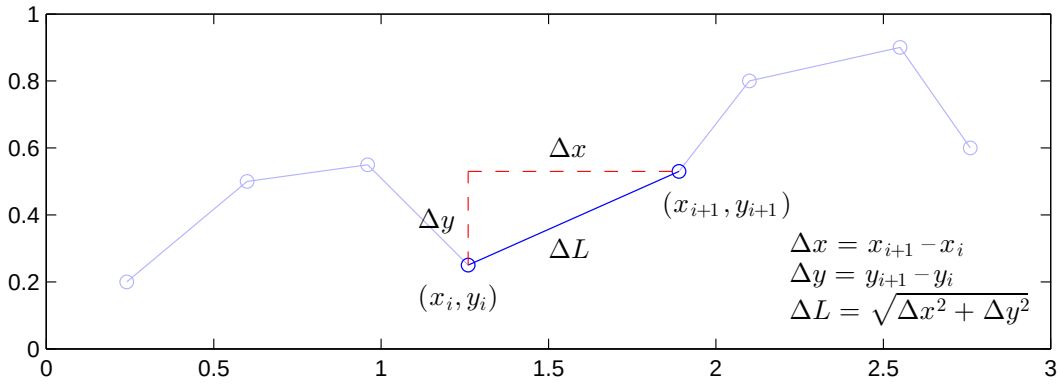
Exempel 6. Vi tänker oss att vi har ett polygontåg $(x_1, y_1), (x_2, y_2) \cdots, (x_n, y_n)$ som vi ritat en figur av



Vill vi beräkna polygontågets längd kan vi göra det med

$$L = \sum_{i=1}^{n-1} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

Denna formel fås genom att använda Pytagoras sats på varje segment i polygontåget.



Antag att koordinaterna samlade i två vektorer $\mathbf{x} = (x_1, x_2, \cdots, x_n)$ och $\mathbf{y} = (y_1, y_2, \cdots, y_n)$, då beräknar vi längden enligt

```
n=size(x)
L=0
for i in arange(0,n-1):
    L=L+sqrt((x[i+1]-x[i])**2+(y[i+1]-y[i])**2)
print(L)
```

Om polygontåget är slutet, dvs. att $x_n = x_1$ och $y_n = y_1$, så omsluts ett område med arean

$$A = \left| \frac{1}{2} \sum_{i=1}^{n-1} (x_{i+1} + x_i)(y_{i+1} - y_i) \right|$$

Denna formel är lite svårare att ta fram. Så här beräknar vi arean i alla fall

```
n=size(x)
A=0
for i in arange(0,n-1):
   A=A+(x[i+1]+x[i])*(y[i+1]-y[i])
A=abs(A)/2
print(A)
```

Uppgift 2. Skriv två funktioner, `polylen` och `polyarea`, för beräkning av längden respektive arean enligt exempel 6. Pröva funktionerna på en triangel och en rektangel.

4.3.2 `while`-satser

En `while`-sats tillåter en grupp av satser att bli repeterade under kontroll av ett logiskt villkor:

```
while uttryck:
    satser
```

Uttrycket i while-satsen är ett logiskt uttryck. Satserna repeteras så länge det logiska uttrycket är sant.

Exempel 7. Man kan beräkna $\sqrt{c}$ med upprepade additioner och divisioner med iterationsformeln

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{c}{x_k}\right),\ k = 0, 1, 2, \cdots$$

där $x_0 = c$. Iterationen avbryts då $d = |x_{k+1} - x_k| < \text{tol}$, där tol är måttet på önskad noggrannhet i approximationen.

Vi beräknar en approximation av t.ex. $\sqrt{2}$ med noggrannheten $\text{tol} = 10^{-16}$. Först ser vi på några steg i iterationen

$$\begin{array}{ll} x_0 = c & x_0 = 2 \\ x_1 = \frac{1}{2}(x_0 + \frac{c}{x_0}) & x_1 = \frac{1}{2}(2 + \frac{2}{2}) = 1.5 \\ x_2 = \frac{1}{2}(x_1 + \frac{c}{x_1}) & x_2 = \frac{1}{2}(1.5 + \frac{2}{1.5}) = 1.4166\cdots \\ x_3 = \frac{1}{2}(x_2 + \frac{c}{x_2}) & x_3 = \frac{1}{2}(1.4166\cdots + \frac{2}{1.4166\cdots}) = 1.4142\cdots \\ \quad\vdots & \quad\vdots \end{array}$$

Sedan använder vi PYTHON enligt

```
c=2
tol=0.5e-10
x=c
d=1
while d>tol:          # så länge d>tol görs följande
     xny=(x+c/x)/2
     d=abs(xny-x)
     x=xny
print(x)
```

Vi gör en skriptfil av koden ovan, alternativt kan vi göra det som en egendefinerad funktion, (`min_sqrt.m`)

```
def min_sqrt(c):
    tol=0.5e-10
    x=c
    d=1
    while d>tol:
        xny=(x+c/x)/2
        d=abs(xny-x)
        x=xny
    return x
```

Vi använder den så här i Console (för att beräkna $\sqrt{5}$)

```
>>> x=min_sqrt(5)
>>> x
2.23606797749979
```

Uppgift 3. Det gäller att

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots$$

Hur många termer måste man ta med i summan för att approximera π med fem korrekta decimaler?

Termerna i summan kan skrivas $\frac{(-1)^i}{2i+1}$ för $i = 0, 1, \cdots$.

(a). Bilda successivt delsummor

$$s_n = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots + \frac{(-1)^n}{2n+1}$$

för allt större heltal n och avbryt när s_n tillräckligt nära $\frac{\pi}{4}$. Använd en `while`-sats.

(b). Använd sedan en `for`-sats och beräkna $\frac{\pi}{4}$ med 1000 termer.

Ibland har man nytta av följande konstruktion

```
while 1:
    sats
    if uttryck:
        break
```

Här kommer upprepning ske ända tills uttrycket i `if`-satsen blir sannt (får värdet 1), då avbryts `while`-satsen och programkoden efter denna utförs.