Python

• Jonathan Nilsson • LGMA50 • Våren 2019 • Version 1.4 •

Introduktion till Python

Programmering med Python

Python är ett populärt programmeringsspråk som används både inom industrin och på många utbildningar. Flera andra programmeringsspråk är också i bruk, som t.ex. Java, Matlab, och C++. Dessa är ungefär lika kraftfulla när det kommer till att göra effektiva matematiska beräkningar, men varje språk har sina för- och nackdelar. Den största fördelen med Python är att notationen är minimal så att koden blir lättläst. Python är dessutom gratis, har öppen källkod, program behöver inte kompileras, och man slipper hålla koll på krångliga saker såsom minneshantering eller typsättning av variabler.

Om man inte har programmerat tidigare kan det kännas svårt i början, det krävs helt enkelt lite tid för att vänja sig vid att skriva och tolka ett program. Det är också svårt att lära sig att programmera bara genom att läsa om det. Det bästa sättet är att prova sig fram och testa olika kommandon för att lära sig hur de fungerar. I instruktionen nedan har jag därför skrivit en guide till vad du kan prova att skriva in i Python för att komma igång. Det är viktigt att inte ha för bråttom när man går igenom texten. Tänk igenom hur de olika kommandona fungerar och försöka gissa vad resultatet kommer att bli innan du kör ett kommando. Var inte heller rädd för att göra dina egna små experiment med de olika operationerna för att se hur de fungerar.

Om man skriver ett helt program, och det inte fungerar kan det vara svårt att hitta felet. Det är därför bra att försöka köra små delar av programmet medan man skriver det - då ser man lättare när ett fel uppstår. Det är också en bra idé att försöka vänja sig vid att söka online efter svar på de frågor som dyker upp. En bra introduktion till Python finns på `https://www.w3schools.com/python/` - här kan man läsa om hur grundläggande kommandon fungerar. I den här kursen kommer vi dock i princip bara att behöva använda grundläggande räkneoperationer, if-satser, loopar, och funktioner.

Att använda denna text

Den här texten innehåller för tillfället en grundläggande introduktion till Python, med fokus på talteoretiska beräkningar. Sist i texten finns kursens datorlaborationer, och ett icke-obligatoriskt programmeringsprojekt för den som siktar på högre betyg. Jag rekommenderar att man försöker ta en titt på den grundläggande introduktionen på egen hand, på så vis har man mer tid att fokusera på uppgifterna under laborationerna.

Kom igång

Den här instruktionstexten är tänkt att användas tillsammans med *Anaconda* - en plattform för Pythonprogrammering. Anaconda finns installerat i datorsalen MVF22 som vi har bokad, men du kan också installera Anaconda på din egen dator: gå till `https://www.anaconda.com/download/`, ladda ned **Python 3.6** versionen, och följ installationsinstruktionerna. Applikationen som man startar programmeringsplattformen med kallas *"Spyder"*. I Windows 10 trycker man på Windows-fönstret längst ned till vänster och skriver "Spyder" för att hitta programmet.

I standardvyn i Spyder finns tre fönster. Till vänster finns en text-editor där man kan skriva in sin programkod. Nere till höger finns en "konsol" där man kan skriva in kor-

tare kommandon, ungefär som på en vanlig miniräknare. I konsolen visas också in- och utdata när du kör dina program som du skrivit i det vänstra fönstret. Uppe till höger finns ett hjälpfönster där man snabbt kan få hjälp om hur man använder kommandon, man kan också i detta fönster växla till att se ett filsystem, eller till att se en lista över sina variabler.

Python som miniräknare

I konsolfönsret nere till höger står det `In [1]:` Här kan du skriva in kommandon som på en miniräknare och trycka enter för att köra dem. Operationerna + - * / fungerar precis som på en miniräknare. Parenteser fungerar också som vanligt, och decimalkomma skrivs med en punkt.

Python använder också lite notation som inte är så vanlig. "Upphöjt till" skrivs t.ex. som $**$ alltså med två multiplikationstecken. För att beräkna 2^{10} skriver man alltså `2**10`.

Genom att skriva $a//b$ beräknar du *heltalsdivisionen* av a med b, vilket är vanlig division fast avrundat nedåt. Det gäller t.ex. att $14//5 = 2$ och $1//2 = 0$.

Du kan enkelt beräkna resten vid heltalsdivision med Python med hjälp av operationen $\%$ (har ingenting att göra med procenträkning). För att beräkna resten när a divideras med b skriver du `a % b`. Vi har t.ex. $53\%7 = 4$ och $63\%9 = 0$.

Prova själv - räkneoperationer

Testa ovanstående kommandon tills du känner att du förstår dem. Utforska också följande:

- När behövs parenteser i Python? I vilken ordning beräknas 3+4*5 och 2**3**4 och 2*3%5?
- Vad händer om du försöker dela med noll?
- Hur snabbt är python? Prova att multiplicera två ungefär 20-siffror tal. Beräkna också 2^n för några stora värden på n. Var går gränsen för vad Python klarar?

Obs!

Ibland kan man råka skriva in nånting som skulle ta mycket lång tid för Python att beräkna. Du kan då stänga konsolen genom att klicka på det lilla röda krysset ovanför konsol-fönstret, då får du upp en ny konsol efter ett par sekunder.

För att få tillgång till fler matematiska funktioner i Python kan du skriva in följande rad i konsolen:

```
from math import *
```

Du har nu tillgång till flera matematiska funktioner. Du kan nu exempelvis beräkna roten ur 3 genom att skriva `sqrt(3)`. Du har också tillgång till trigonometriska funktioner, exponentialfunktionen, och logaritmer. Prova att skriva

```
cos(0)
sin(pi/6)
exp(1)
log(100)
```

```
log10(100000)
```

Variabler

Precis som på en miniräknare kan man skapa variabler och ge dem värden. Prova t.ex. att i konsolen skriva in

```
a=5
```

Du har nu skapat du en variabel a och tilldelat den värdet 5. Prova att på nästa rad skriva

```
a*a
```

Vad tror du händer om du härnäst skiver

```
a=a+1
```

Vilket värde har variabeln a nu? Prova genom att skriva `a` i konsolen.

Här ska man tänka på att "=" tecknet i Python används för att tilldela (ställa in) värdet på variabeln till vänster till det högra värdet. Att skriva `a=a+1` har alltså innebörden "sätt värdet på a till det aktuella värdet på a plus 1". Således har nu variabeln a värdet 6.

Variabler kan ha vilka namn som helst (utan mellanslag). Prova t.ex. att i konsolen i tur och ordning skriva följande:

```
hej=3
häst=7
apa=hej*häst
apa
```

I mer komplicerade program är det dock en bra idé att använda variabelnamn som påminner en om vad variabeln gör istället för djur. Variabler kan exempelvis heta "summan" eller "maxvärde" eller "svar".

Datatyper

Vi har hitills sett exempel på räkningar med heltal och decimaltal.

Strängar

Python kan också arbeta med text. En följd av tecken kallas en *sträng*. Strängar skrivs i Python inom citationstecken. Prova att i konsolen skriva in

```
x="Choklad"
y="muffins"
```

Du har nu skapat två variabel x och y som båda innehåller strängar. Strängar kan slås ihop med hjälp operationen +, eller upprepas med *. Prova att i konsolen skriva in följande. Gissa vad resultatet kommer att bli innan du trycker enter!

```
x+y
y+x
x*5
(x+y)*100
```

```
x*y
```

Booleska variabler - sant eller falskt

En Boolesk variabel är en variabel som kan ha två möjliga värden: "True" eller "False". Det är den kanske enklaste datatypen.

För att testa om två värden a och b är lika i Python skriver man `a==b`. Resultatet blir då antingen "True" eller "False". Lägg märke till att detta är annorlunda än att skriva $a = b$, vilket ju sätter värdet på a till värdet på b. Prova att i konsolen skriva in

```
x=5
x==7
x*x==25
```

Prova själv - olikheter

Det finns även andra operationer som ger värdet "True" eller "False". Prova att i konsolen skriva in följade, och försök lista ut hur de olika operationera fungerar!

```
x=5
x>3
x<5
x<=5
x!=7
x!=5
```

Sant/falskt variabler kan kombineras med de logiska operationerna "and" och "or". Prova att skriva in följande, och gissa vad resultaten kommer att bli!

```
5==5 and 2==3
5==5 or 2==3
5==5 or 2==2
```

Skriv också in följande, och fundera på varför svaren blir olika. Är uttrycket innanför parantesen sant eller falskt?

```
(0==1 and 0==0) or 1==1
0==1 and (0==0 or 1==1)
```

Växla mellan datatyper

Ibland vill man växla mellan olika datatyper. Du kan t.ex. konvertera ett decimaltal till ett heltal. Prova att skriva följande:

```
a=8.762
int(a)
round(a)
```

Prova själv - avrundning

Skriv i konsolen (om du inte redan gjort det)
`from math import *`
Du har nu också tillgång till funktionerna "floor" och "ceil" i konsolen.
Undersök vad skillnaden är mellan funktionerna floor, ceil, round, och int!

Ibland kan man få en sträng bestående av siffror - då behöver man omvandla strängen till ett heltal eller decimaltal för att kunna räkna med den numeriskt. Decimaltal kallas också flyt-tal, och man skriver float för att omvandla till flyt-tal. Prova att skriva in följande i konsolen:

```
s="3.1415"
s+1
s=float(s)
s+1
```

Från början är s en sträng, och vi kan inte addera ett heltal till en sträng. Men efter att vi gjort om s till ett decimaltal kan vi räkna med den som vanligt. Vad händer om du istället för float(s) skriver int(s)?

Det är också vanligt att man vill sätta ihop tal med strängar. Skriv i konsolen:

```
x=2019
"Det är nu år " + x +". Gott nytt år!"
```

Du får då ett felmeddelande. Detta beror på att du försöker slå ihop två olika datatyper: strängar och heltal. Detta kan lösas genom att konvertera variabeln x från ett heltal till en sträng innan vi slår ihop dem. Detta gör man genom att skriva $str(x)$. Skriv alltså istället:

```
x=2019
"Det är nu år " + str(x) + ". Gott nytt år!"
```

Listor

Listor är en viktig datatyp i Python. En lista skrivs inom hak-parenteser och datan separeras med komman. En lista kan innehålla olika datatyper. Skriv in följande i konsolen:

```
x=[5,-7,3+50+100, "hej", 12.345, "grus", 1+1==2]
x
```

Du har nu skapat variabel x som innehåller in lista bestående av 7 element av blandade typer: tre heltal, ett decimaltal, två strängar, och en sant/falsk-variabel.

Du kan plocka ut ett individullt element ur en lista från en viss position. Första elementet i listan har position 0, andra har position 1 och så vidare. Skriv

```
x[1]
x[0]+x[4]
x[7]
```

Längden av listan x (antal element i x) får du genom att skriva `len(x)`. Du kan också lägga till element sist i listan x genom att t.ex. skriva `x.append(12)`. Prova följande i konsolen:

```
len(x)
x.append(3)
x.append("Fermat")
len(x)
```

Om man vill skriva ut sista elementet i en lista x men inte vet hur lång den är, hur kan man skriva? Att skriva `x[len(x)]` fungerar inte, vad är felet och hur kan man rätta till det?

Python har också en enkel metod för att testa om ett givet element ligger i en lista. Prova att skriva:

```
15 in x
"hej" in x
```

Listor kan också användas för att ställa in värden på flera variabler samtidigt: kommandot

```
[a,b,c]=[5,12,13]
```

ställer in variablerna a=5, b=12, c=13 på en enda rad.

Prova själv - listor av listor

En lista kan innehålla en annan lista som element! Detta ger ett enkelt sätt att spara data i matris-form. Prova att i konsolen skriva

```
x=[[1,2,3],[4,5,6],[7,8,9]]
x[1][0]
```

Varför blir resultatet som det blir? Vad ska a och b vara för att x[a][b]=8?

Prova själv - sortering av listor

Det finns en funktion som automatiskt sorterar listor i Python. Prova att skapa en lista som heter **tal** och som innehåller ett antal heltal och decimaltal. Skriv sedan i konsolen

```
tal.sort()
tal
```

I vilken ordning sorteras talen? Vad händer om du sorterar en lista bestående av strängar? Eller en lista med både tal och strängar?

Slumptal

Ibland kan det vara användbart att skapa slumptal i Python. För att kunna använda slump-verktyg börjar man med att skriva följande i konsolen:

```
from random import *
```

Du har nu tillgång till slumptalsgeneratorn. Prova att skapa tre slumpade heltal mellan 0 och 5 genom att skriva:

```
randint(0,5)
randint(0,5)
randint(0,5)
```

Du kan skapa ett slumpat decimaltal mellan 0 och 1 genom att skriva `random()`.

Genom att använda slumptal kan du t.ex. välja ut ett slumpat element ur en lista. Testa t.ex. att i konsolen skriva:

```
djur=["katt","hund","orm","dinosaurie"]
"Jag har en " + djur[randint(0,3)] + " som husdjur!"
```

Skriv in andra raden flera gånger så ser du att du får olika resultat.

Här är det egentligen smartare att ersätta talet "3" med "len(djur)-1": då fungerar programmet som det är tänkt även om du lägger till fler djur i listan!

Att skriva program i Python

Från och med nu ska vi sluta använda konsolen, och istället skriva program i det vänstra fönstret. Börja med att skapa en tom fil och spara den någonstans, t.ex. på skrivbordet. När man skrivit ett program och sedan startar det genom att trycka F5 så kör man alla raderna kod efter varandra. Detta ger möjlighet att utföra mer komplicerade operationer. Man kan också lättare ändra sin kod och sedan köra hela programmet igen.

En skillnad mot att använda konsolen är att program inte ger någon output om man inte ber om det. För att få en utskrift från ett program måste man använda kommandot "print".

Prova att skriva följande enkla program i det vänstra fönstret, spara filen och kör den genom att trycka F5.

```
1 a=3
2 b=5
3 c=19
4 print("Produkten a*b*c blir " + str(a*b*c))
5 print("Summan a+b+c blir " + str(a+b+c))
```

Du kan sedan enkelt ändra värdena på a,b,c på rad 1-3 och köra programmet igen genom att trycka F5.

I ett program kan man be den som kör programmet att skriva in något genom att använda kommandot "input".

Skriv följande program i det vänstra fönstret, spara det och kör det genom att trycka F5.

```
1 x=input("Vad heter du? : ")
2 print("Du heter " + x + "! Vilket bra namn!")
```

Här använder vi kommandot "input" för att låta användaren ange en sträng x i konsolen. Sedan skapar vi en sammansatt sträng och skriver ut den med print-kommandot.

Obs!

Resultatet av input kommandot blir alltid en sträng. Om man vill att personen ska ange ett heltal eller decimaltal måste man alltså göra om strängen till en annan datatyp. Man kan t.ex. skriva

```
1 x=int(input("Ange ett heltal: "))
2 y=float(input("Ange ett decimaltal: "))
```

Då kan vi efteråt räkna med x och y som tal och inte som strängar.

Ibland glömmer man bort vad delar av ens kod gör. Därför är det bra att skriva *kommentarer* i Pythonkoden. En kommentar påverkar inte programmet alls. I koden skriver man tecknet # följt av kommentaren på samma rad. Ovanstående program kan till exempel kommenteras såhär:

```
1 #Detta är ett enkelt program
2 x=input("Vad heter du? : ") #Be personen ange sitt namn
3 print("Du heter " + x + "! Vilket bra namn!") #Gratulera
```

Som du ser får kommandon, strängar, och tal olika färg i kodfönstret. Färgerna spelar ingen roll för programmet, men det hjälper en lättare läsa och tolka sin kod.

If-satser

Börja med att skriva följande program och testa vad det gör.

```
1 x=int(input("Ange ett tal: "))
2 if x>1000000:
3     print("Det var ett megastort tal!")
```

Detta är ett exempel på en if-sats. Koden på den inskjutna tredje raden körs endast om villkoret ovanför håller. Man kan också skjuta in fler rader efter varandra så att man får ett helt "block" med kod som bara körs om villkoret håller. Man skjuter in rader genom att använda tab-knappen (ovanför Caps Lock på tangentbordet). Efter if-satsen fortsätter programmet som vanligt.

Man kan också skapa if-satser med fler än ett fall i sina villkor:

```
1 x=int(input("Ange ett tal: "))
2 if x>1000000:
3     print("Det var ett megastort tal!")
4 elif x>1000:
5     print("Det var ett stort tal!")
6 elif x>10:
7     print("Det var ett lagom stort tal!")
8 else:
9     print("Det var ett litet tal!")
```

Här använder vi två nya kommandon. "Else if" skrivs elif, detta kan användas för att skapa fler fall i if-satsen. elif-villkoret på rad 4 testas endast om ovanstående fall inte inträffade. På samma vis testas villkoret på rad 6 endast om de föregående två fallen inte inträffade. På rad 8 står det bara "else:" utan villkor. Detta sista fall körs när *inga* av ovanstående fall inträffar.

Fundera på vad som hade hänt om vi hade byt ut "elif"-raderna mot vanliga "if"-satser och sedan skrivit in ett tal större än en miljon!

Här är ett mer praktiskt exempel. Följande program beräknar BMI-värden (ens kroppsvikt delat med kvadraten av ens längd i meter) och avgör om detta svarar mot undervikt/normalvikt/övervikt. Läs igenom och se om du förstå hur programmet fungerar.

```
L=float(input("Ange längd i cm: "))
V=float(input("Ange vikt i kg: "))
bmi=V/((L/100)**2)
print("BMI-värdet var " + str(bmi)+".")
if bmi<18.5:
    print("Detta motsvarar undervikt")
elif bmi>=25:
    print("Detta motsvarar övervikt")
else:
    print("Detta motsvarar normalvikt")
```

Villkoret i if-satsen kan vara vad som helst av typen "sant/falskt", så det går bra att skriva komplicerade villkor. Om t.ex. x,y,z är tal och frukt är en lista med frukter så kan vi skapa if-satser som:

```
if (x>5 or y<0) and z==2:
```

eller

```
if "banan" in frukt:
```

Prova själv - jämnt eller udda

Skriv ett program som ber användaren att skriva in ett tal och sedan skriver ut i konsolen om talet är jämnt, eller udda. *Tips: Använd a%2 för att ta reda på om a är jämnt eller udda*

Loopar

Genom att använda **loopar** kan man upprepa delar av sin kod flera gånger, detta är mycket användbart som vi ska se framöver.

For-loopar

Den vanligaste typen av loop kallas för en for-loop. Testa t.ex. att köra följande program:

```
for x in range(1,8):
    print(x)
```

Som en mening kan programmet tolkas som "för varje heltal x i intervallet $[1, 8)$, skriv ut x". Programmet skapar alltså en loop-variabel x, och kör sedan den andra raden flera gånger där x först är 1, sedan 2, och så vidare.

Obs!

Notera att sista talet x som skrevs ut i föregående exempel är 7 och inte 8. Detta är en konvention i Python som man måste vänja sig vid: att när man skriver "x in range(a,b)" så är första värdet på x lika med a och det sista värdet för x är b-1.

Loopar kan t.ex. användas för att approximera summor och serier. Följande program beräknar de 100 första termerna i summan $\sum_{k=1}^{\infty} \frac{1}{k^2}$.

```
1 summa=0
2 for k in range(1,101):
3     summa=summa+1/(k*k)
4 print(summa)
```

I steg nummer k adderar vi alltså termen $\frac{1}{k^2}$ till variabeln summa. Notera att vi var tvungna att sätta summa till 0 i början. (Varför?) Vad tror du händer om vi också skjuter in den fjärde raden i programmet?

Man kan också loopa igenom en lista med en for-loop:

```
1 frukter=["äpple","kiwi","banan","ananas"]
2 for f in frukter:
3     print(f + " är gott!")
```

Här antar alltså loop-variabeln f i tur och ordning alla värden i listan frukter. Det spelar inte någon roll vad vi kallar denna variabel, den finns bara inne i loopen.

Prova själv - triangeltal

Ett triangeltal är ett tal på formen $1+2+3+\cdots+n$ för något n. Exempelvis är 3, 10, och 5050 triangeltal. Skriv ett program som skriver ut de första 100 triangeltalen! *Tips: detta går att göra med en enkel for-loop.*

While-loopar

En annan typ av loop är while-loopen, denna låter oss upprepa ett stycke kod så länge ett villkor är sant. Följande program visar hur sådana kan användas:

```
1 x=1
2 while x<5000:
3     x=2*x
4     print(x)
```

Programmet fortsätter att dubbla värdet på x och skriver ut resultaten ända tills x blir större än 5000. Vilket blir det sista talet som skrivs ut? Varför? Vad händer om man byter plats på rad 3 och 4?

Om du vill krascha Python kan du skriva en *oändlig loop* av typen while(True), exempelvis

```
1 while 1==1:
2     print("Hjälp, jag är fast i en loop!")
```

Programmet fortsätter att köras så länge 1=1, det vill säga för alltid (eller i praktiken tills datorns minne tar slut eller tills du stänger Python-konsolen).

Loopar i loopar

Koden som körs innuti en loop kan vara vad som helst - exempelvis en annan loop!

Prova att skriva och köra följande program:

```
1 for i in range(1,5):
2     for j in range(1,4):
3         print(str(i) + " och " + str(j))
```

Tänk igenom hur programmet fungerar, lägg speciellt märke till den dubbla rad-inskjutningen på rad 3. Hur många utskrivna rader fick du? Lägg också märke till ordningen av utskrifterna.

I följande program använder vi ännu en dubbelloop för att skriva ut en multiplikationstabell av storlek 10*10.

```
1 for rad in range(1,11):
2     x=""
3     for kol in range(1,11):
4         x=x+str(rad*kol)+" ")
5     print(x)
```

Här upprepas rad 2-5 tio gånger medans variabeln rad går från 1 till 10. För varje värde på rad börjar vi med att skapa en tom sträng x, och sedan låter vi variabeln kol gå från 1 till 10 medans vi lägger till rad*kol till strängen x. Tillslut skriver vi ut raden x. Programmets fjärde rad kommer alltså att köras totalt 100 gånger. Du kan enkelt göra en större tabell genom att öka talet 11 i range-kommandot.

Funktioner

Genom att skapa funktioner i Python kan man på ett enkelt sätt återanvända sin tidigare kod. Notationen för att definiera en funktion illustreras nedan, vi definierar funktionen $f(x) = x^2 + 3x + 5$ genom att skriva följande:

```
1 def f(x):
2     return x*x+3*x+5
```

Funktionens namn är f, och stoppar man in x i funktionen så returneras x^2+3x+5. Prova skriva in programmet och kör filen. Du får då ingen output, men funktionen f finns nu i datorns minne, och man kan räkna med den effektivt som med en vanlig funktion. Prova att direkt i konsolen skriva:

```
f(5)
f(f(0)+1)
häst=100
f(häst)
```

Det går också bra att skriva sådana rader i sitt program under definitionen av f, men då måste vi använda "print" kommandot.

Säg att vi vill hitta alla lösningar till ekvationen $x^2 + 3x + 5 \equiv 0 \mod 97$. Det kan vi göra genom att utöka ovanstående program såhär:

```
1 def f(x):
2     return x*x+3*x+5
3
4 for i in range(0,97):
5     if f(i)%97==0:
6         print(str(i)+ " är ett nollställe till f modulo 97")
```

Här är det viktigt att definitionen av f står ovanför for-loopen: om inte f är definierad kan ju programmet inte beräkna f(i) inne i for-loopen.

Programmet beräknar alltså $f(i)$ modulo 97 för alla i från 0 till 96, och när $f(i) \equiv 0$ modulo 97 gör programmet en utskrift om detta. En fördel med att göra på detta vis är att vi enkelt kan byta ut f mot en annan funktion i den övre definitionen och fortfarande använda den nedre koden.

En funktion kan heta vad som helst, kan ha hur många argument som helst (eller inga!), och kan innehålla hur många rader kod som helst innan den returnerar ett värde. Värdet som returneras kan också vara av vilken typ som helst. Funktionen behöver inte ens returnera något värde alls!

Här är ett exempel på en funktion som inte har några argument, och inte returnerar något värde:

```
def hälsning():
    for i in range(0,5):
        print("Ha en bra dag!")
```

Allt funktionen gör är alltså att skriva ut texten "Ha en bra dag!" fem gånger. Efter att vi definierat funktionen kan vi köra den genom att skriva

```
hälsning()
```

Frivilliga argument

Nu ändrar vi ovanstående funktion till följande:

```
def hälsning(n,namn="Fermat"):
    for i in range(0,n):
        print("Hej "+namn+"! Ha en bra dag!")
```

Den här funktionen skriver ut en mer personlig hälsning n gånger. Vi kan t.ex. köra programmet och sedan använda funktionen med

```
hälsning(3,"Gauss")
hälsning(1,"alla glada människor")
```

Men vi kan också skriva bara

```
hälsning(5)
```

För att hälsa till Fermat. Detta beror på att vi valt ett *standardargument* för variabeln "namn". Om vi inte anger en sträng som andra argument används alltså namn="Fermat".

Funktionen range() i våra for-loopar fungerar faktiskt på detta vis. Skriver man bara "range(8)" så betyder det samma sak som "range(0,8)". Nollan är alltså ett standardargument.

Rekursiva funktioner

En rekursiv funktion är en funktion som *anropar sig själv*! Vi visar med ett exempel på en funktion som räknar ut siffersumman av ett tal (siffersumman av 1574 är exempelvis $1+5+7+4=17$).

```
def siffersumma(n):
    if n==0:
```

```
        return 0
    return n%10+siffersumma(n//10)
```

Vi beräknar här siffersumman av ett tal n genom att ta den sista siffran ($n\%10$) plus siffersumman av resterande siffror ($n//10$). Om vi t.ex. skriver `siffersumma(38)` kommer programmet först att försöka beräkna `38%10+siffersumma(38//10)`, alltså `8+siffersumma(3)`. För att klara detta måste programmet först beräkna siffersumman av 3, vilket med programmet blir `3+siffersumma(0)`, och tillslut beräknas `siffersumma(0)=0`. Med andra ord, programmet "dyker djupare i sig själv" två gånger när det beräknar siffersumman av 38. Det är viktigt att ha if-satsen, annars blir det en oändlig rekursion och Python kraschar. Det är också såklart möjligt att skriva detta program utan rekursion och med en for-loop istället, men i vissa lägen är rekursion mer lämpligt.

Här kommer ett exempel till. Antag att vi har en lista "`saker`" som består av andra listor, som eventuellt också har listor i sig o.s.v, och säg att vi vill skriva ut allting i `saker` på en rad var. Låt t.ex. `saker=[1,"fem",2,[[2,[8]]],3,["häst",[5,4]]]`. Om vi loopar igenom listan med `for x in saker: print(x)` så får vi bara 6 rader utskrivna, eftersom "`saker`" innehåller två listor. Problemet kan lösas med rekursion:

```
def skrivut(a):
    if type(a)!=list: #testar om a är en lista
        print(a)  #skriv ut tal och strängar direkt
    else:     #annars, om a är en lista,
        for x in a:  #loopa igenom listan a
            skrivut(x) #rekursivt anrop för varje del-lista
```

Nu kan du skriva `skrivut(saker)` i konsolen och få det önskade resultatet.

Funktionsexempel

Här kommer ett exempel på en funktion som beräknar gcd av två positiva heltal.

```
def dåligGCD(a,b):
    for i in range(0,b):
        if a%(b-i)==0 and b%(b-i)==0:
            return b-i
    print("Något gick fel!")

print(dåligGCD(54321,9876))
```

Programmet går alltså igenom alla heltal från b och nedåt ända tills det hittar ett tal som delar både a och b. Poängen som det här programmet illustrerar är att så fort vi kommer ned till ett return-kommando så är funktionen klar, och efterföljande rader i funktionen körs ej. Enda sättet att få utskriften "Något gick fel" är alltså om hela for-loopen körs igenom utan att något värde returneras - detta kan bara hända om du skriver in a eller b som ett decimaltal eller ett negativt tal.

En annan viktig poäng är att vi måste gå igenom talen uppifrån och nedåt: vi vill ju returnera den *största* gemensamma delaren. Men i for-loopen går variabeln i från 0 och uppåt. Därför använder vi i programmet $(b-i)$ för att få en variabel som går från b och nedåt till 1. Notera att vi lika gärna kunde ha gått nedåt från a eftersom $gcd(a,b)$ är mindre än både a och b.

Varför tror du jag döpt programmet till dåligGCD? Du kan prova att beräkna gcd av två ännu större tal med programmet!

Tidtagning

Vissa räkneoperationer kan vara mycket krävande, som exempelvis att hitta primtalsfaktorerna till ett stort tal. Då kan det vara intressant att kunna mäta hur snabbt ens program är.

Genom att skriva

`from time import *`

i konsolen (eller överst i sitt program) får man tillgång till Pythons tids-paket.

Prova sedan att i konsolen skriva `time()`

"time()" är en funktion utan argument. Den returnerar den exakta tiden sedan kl 00:00 den första januari 1970 mätt i sekunder(!)

Tidtagning av program

Nedan är ett exempelprogram som beräknar hur lång tid det tar att beräkna summan $1 + 2 + 3 + \ldots + 100$ med en loop.

```
1 from time import *
2 start=time()
3 summa=0
4 for i in range(1,101):
5     summa=summa+i
6 print(summa)
7 end=time()
8 print("Det tog "+ str(end-start) + " sekunder.")
```

Vi sätter variabeln "start" till tiden precis innan for-loopen och "end" till tiden efter loopen är klar. Skillnaden mellan end och start blir ju då tiden det tog för programmet att köras. Python är dock så snabbt att svaret ges nästan omedelbart - prova att ändra så att programmet summerar upp till en miljon eller 10 miljoner istället. Hur lång tid tog det?

Raderna 3-6 kan såklart ersättas med vilket program som helst för att se hur lång tid det tar att köras.

Sleep

Man kan få ett program att pausa eller "somna" genom att använda kommandot sleep. Detta kan vara bra om man t.ex. vill hinna läsa output från ett snabbt program. Kommandot "sleep(x)" pausar programmet i x sekunder. Här kommer ett enkelt exempel:

```
1 from time import *
2 prod=1
3 for i in range(0,30):
4     print(prod)
5     sleep(0.5)
6     prod=prod*7
```

Programmet skriver alltså ut de första 30 potenserna av 7 med en halv sekunds mellanrum.

Exempelprogram: Euklides algoritm, gcd, lcm, och Eulers ϕ-funktion

Här följer ett exempel på hur man med relativt korta program kan uföra komplicerade beräkningar och återanvända sin kod. Följande program definierar funktioner som kan beräkna gcd, lcm, och värden för Eulers ϕ-funktion.

```
def gcd(a,b):
    while(b!=0):
        [a,b]=[b,a%b]
    return a

def lcm(a,b):
    return a*b//gcd(a,b)

def phi(n):
    svar=0
    for a in range(1,n+1):
        if gcd(n,a)==1:
            svar=svar+1
    return svar

print(gcd(8905634793,789438))
print(lcm(789423985,43278985))
print(phi(67832))
```

Beskrivning av koden

I funktionen gcd startar vi med två tal a och b. I varje loop-steg ersätts sedan det minsta talet b med resten $a\%b$, och det nya stora talet blir b. Varje loop steg motsvarar alltså ett steg i Euklides algoritm. Detta fortgår tills den sista resten blir 0, och då ligger den näst sista resten i a, så detta tal returneras. Om det är svårt att hänga med kan du fundera över vad som händer rad för rad när programmet beräknar gcd(5,2), alltså när $a = 5$ och $b = 2$ i början. Notera att koden fortfarande fungerar om $b > a$ - i första loop-steget skiftas då variablerna!

Vi vet att minsta gemensamma multipeln av två tal a och b ges av $a * b/gcd(a, b)$. Därför är det mycket enkelt att skriva en lcm-funktion genom att återanvända vår tidigare definierade funktion gcd. Lägg märke till att vi använde heltalsdivision // för att få output som ett heltal och inte ett decimaltal.

Funktionen phi beräknar värden för Eulers phi-funktion. $\phi(n)$ är ju antalet tal a mellan 1 och n som uppfyller $gcd(a, n) = 1$. Programmet loopar igenom alla tal mellan 1 och n, och beräknar $gcd(a, n)$ genom att använda vår tidigare definierade funktion. För varje a med $gcd(a, n) = 1$ adderar vi ett till variabeln "svar", och efter att vi gått igenom alla a returnerar vi svar.

Till slut använder vi våra funktioner för att beräkna $gcd(8905634793, 789438)$, $lcm(789423985, 43278985)$ och $\phi(67832)$.

Man bör skriva sina funktioner överst i ett program om de ska användas senare i programmet. I vårt exempel är det också viktigt att gcd-funktionen står överst eftersom både lcm-funktionen och phi-funktionen använder sig av gcd-funktionen.

Effektivitet

Notera att funktionen gcd är väldigt snabb: du kan prova att beräkna gcd(a,b) för två 30-siffriga tal a och b.

Å andra sidan är phi(n) ganska långsam: prova exempelvis att lägga till två siffror och beräkna $\phi(6783200)$ - det tar flera sekunder på en normalsnabb dator. Varför tror du phi är långsammare än gcd?

Det är också intressant att jämföra funktionen "gcd" med vår tidigare funktion "dåligGCD" ovan. Båda kan nästan omedelbart beräkna gcd av två 5-siffriga tal. Men om vi låter a och b ha 30 siffror så kommer fortfarande gcd(a,b) att beräknas på under en sekund medans dåligGCD(a,b) skulle ta längre tid på sig än universums ålder - detta trots koden i funktionen "gcd" faktiskt var kortare!

Datorlaboration I

Arbeta ensam eller i par med denna datorlaboration. Om man arbetar i par är det bäst att man har ungefär samma förkunskaper, så att båda får ut någonting av labben. Om ni blir klara med uppgifterna under labben får ni göra en kort redovisning där ni berättar hur era program fungerar, och ni kan då bli godkända direkt. Annars får man göra en **individuell** inlämning i efterhand på följande vis:

- Skapa en Python-fil som innehåller alla obligatoriska funktioner från uppgifterna. "I mån av tid"-uppgifterna behöver du inte göra.
- Lägg in kommentarer direkt i Python-filen som förklarar de viktigaste delarna av dina funktioner. Svara också på uppgiftens frågor i kommentarerna.
- Spara filen som ditt namn i formatet L1FörnamnEfternamn.py
  och maila till jonathn@chalmers.se **senast två veckor efter labben**.

Uppgift 1

- Skapa en funktion **fak(n)** som returnerar $n!$ (n-fakultet) genom att använda en loop. Se till att $0! = 1$ beräknas korrekt, alltså att $fak(0) = 1$.
- Skapa i samma fil en funktion **binom(n,k)** som beräknar binomialkoefficienten $\binom{n}{k}$. Funktionen binom ska använda sig av din funktion fak.
- (I mån av tid) Skapa i samma fil en funktion **Pascal(m)** som skriver ut de första m raderna i Pascals triangel. Funktionen ska använda din funktion binom.

Uppgift 2

- Skapa en funktion **ärPrimtal(n)** som testar om n är ett primtal eller inte genom att testa om n är delbart med 2,3,4,5,...,n-1. Funktionen ska returnera True eller False. Testa hur stora n programmet klarar på rimlig tid. Är exempelvis 179427121 ett primtal?
- Skapa i samma fil en funktion **primtalUnder(n)** som returnerar en lista som innehåller alla primtal mindre än n. Funktionen primtalUnder ska använda din funktion ärPrimtal.
- Förbättra din funktion ärPrimtal genom att förändra den så att den bara testar om n är delbart med varje heltal upp till $\sqrt{n}$. *Tips: använd en while-loop!* Varför räcker det att testa upp till $\sqrt{n}$? Testa hur mycket snabbare dina funktioner blir.
- (I mån av tid) Skapa en funktion **faktorisera(n)** som returnerar en sorterad lista med primtalsfaktorerna i n. Försök göra funktionen så effektiv som möjligt, och prova hur stora tal den kan faktorisera!

Spara din kod, vi kan komma att återanvända dessa funktioner på kommande laborationer!

Datorlaboration II

Arbeta ensam eller i par med denna datorlaboration. Om man inte redovisar muntligt på labben får man göra en **individuell** inlämning i efterhand på följande vis:

- Skapa en Python-fil som innehåller alla obligatoriska funktioner från uppgifterna och lägg in kommentarer direkt i Python-filen som förklarar de viktigaste delarna av dina funktioner. Svara också på uppgiftens frågor i kommentarerna. Det är viktigt att dina funktioner uppfyller precis det som står i instruktionen.
- Spara filen som ditt namn i formatet L2FörnamnEfternamn.py
  och maila till jonathn@chalmers.se **senast en vecka efter labben**.

Uppgifter

- Klistra in dina funktioner "ärPrimtal" (rot-versionen) och "primtalUnder" från Labb 1 överst i ditt program. Skapa en funktion **FermatTest(n,k=5)**. Funktionen ska testa om n är ett primtal genom att välja k stycken slumpmässiga tal a mellan 1 och n och för vart och ett av dessa testa om $a^n \equiv a \mod n$ med hjälp av Pythons inbyggda funktion "pow". Sedan ska programmet returnera antingen *True* eller *False*. False ska returneras om vi är *säkra* på att n är sammansatt. *True* ska returneras om n *nog är ett primtal*. Motivera teorin bakom testet. Testa också din funktion på några höga tal n. Diskutera vilken som är bäst: ärPrimtal eller FermatTest.
- Skapa en kopia av din funktion "primtalUnder" som heter **puFermat(n)** som använder "Fermattest" istället för "ärPrimtal" för att testa om tal är primtal, men i övrigt ser likadan ut (du kan behöva behandla primtalen 2,3 separat).
- Hitta sedan ett sätt att jämföra de två listorna
  primtalUnder(10**5) och puFermat(10**5).
  Påverkas resultatet om vi ändrar k? Vilken slutsats kan du dra?
- Skapa en funktion **letaMersenne(n)** som letar upp och skriver ut alla Mersenne-primtal $2^p - 1$ för $p \leq n$ på ett effektivt sätt. Funktionen bör använda dina funktioner FermatTest och primtalUnder. Hur många Mersenneprimtal hittar du på rimlig tid? Jämför med listan på `https://sv.wikipedia.org/wiki/Mersenneprimtal#Lista_%C3%B6ver_mersenneprimtal`

Obs!

Python kan snabbt beräkna $a^k \mod n$ med successiv kvadrering med den inbyggda funktionen **pow(a,k,n)**. Detta är otroligt mycket snabbare än att skriva `a**k%n`. Problemet är att matematikpaketet har en annan funktion med samma namn som bara tar två argument, så om vi har skrivit `from math import *` så skrivs funktionen pow över och vi kan inte längre använda den. För att kunna använda både "pow" och "sqrt" kan vi t.ex. istället skriva `from math import sqrt`. Om du tidigare importerat hela paketet math så behöver du starta om konsolen.
En annan lösning är att istället skriva `import math` överst i programmet. Då kan man skriva `math.sqrt(49)` för att beräkna $\sqrt{49}$.

Datorlaboration III

Arbeta ensam eller i par med denna datorlaboration. Om man inte redovisar muntligt på labben får man göra en **individuell** inlämning i efterhand på följande vis:

- Skapa en Python-fil som innehåller alla obligatoriska funktioner från uppgifterna och lägg in kommentarer direkt i Python-filen som förklarar de viktigaste delarna av dina funktioner. Svara också på uppgiftens frågor i kommentarerna. Det är viktigt att dina funktioner uppfyller precis det som står i instruktionen. I mån av tid-uppgiften behöver du inte göra. Spara filen som ditt namn i formatet L3FörnamnEfternamn.py och maila till jonathn@chalmers.se **senast en vecka efter labben**.

Uppgifter

- Klistra in dina funktioner "ärPrimtal" och "primtalUnder" från Labb 1 och 2 överst i ditt program. Börja med att skriva en funktion **faktor(n)** som tar ett *sammansatt* tal n och returnerar *en* faktor i n, alltså ett tal på mellan 2 och $n-1$ som delar n. Funktionen kan därefter användas för att splittra ett tal i två faktorer: n är ju produkten av $faktor(n)$ och $n//faktor(n)$.

- Skapa en funktion **LegTest(a,p)** som returnerar Legendresymbolen $\left(\frac{a}{p}\right)$, alltså 1 om ekvationen $x^2 \equiv a \mod p$ är lösbar, och annars -1. Funktionen ska göra detta genom att testa alla värden på x från 1 till $p-1$. När a är 0 mod p ska du returnera 0.

- Skriv en **rekursiv** funktion **Leg(a,p)** som också returnerar $\left(\frac{a}{p}\right)$, men gör detta genom att reducera Legendre-symbolen med kvadratiska reciprocitetssatsen och räkneregler för Legendre-symbolen. Se till att alla fall täcks in. Om du inte får funktionen att fungera så prova att göra små utskrifter i programmet så att du ser var felen uppstår. Du kan också prova att din funktion klarar att korrekt beräkna Legendre-symboler för små a och p, vi har t.ex.

$$\left(\frac{1}{13}\right) = 1 \qquad \left(\frac{2}{5}\right) = -1 \qquad \left(\frac{2}{7}\right) = 1 \qquad \left(\frac{6}{17}\right) = -1$$

  Prova att din funktion fungerar genom att skriva följande kod nederst i ditt program och köra det:

```
for i in range(100):
    primtal=primtalUnder(1000)
    p=primtal[randint(1,len(primtal)-1)] #slumpa p
    a=randint(1,1000) #slumpa a
    if LegTest(a,p)!=Leg(a,p):
        print("Olika i Leg("+str(a)+","+str(p)+")")
```

  Vad gör ovanstående kod?

- Jämför effektiviteten för funktionerna LegTest och Leg genom att beräkna Legendre-symboler för några relativt stora a och p.

- *(I mån av tid)* Skapa en funktion **Jacobi(a,b)** som beräknar Jacobi-symbolen $\left(\frac{a}{b}\right)$. Kom ihåg att svaret ska bli noll när a och b ej är relativt prima.

Pythonprojekt: Kryptografi

Detta är ett **icke-obligatoriskt** programmeringsprojekt på kursen, främst riktat till de som siktar på ett högre betyg (det är dock möjligt att få det högre betyget även utan detta, se kriterierna i kursplaneringen). Den största skillnaden mot datorlaborationerna är att projektet kräver mer självständigt arbete, instruktionerna är inte lika direkta och man måste därför hitta svar på sina frågor på egen hand.

Kryptering

I princip all internetkommunikation krypteras idag med talteoretiska metoder. Alla webadresser som börjar på "https://" är skyddade med krypton som oftast bygger på RSA-protokollet som vi har diskuterat i kursen. Nyckeln som används på dessa krypton har normalt 256 tecken, så för att knäcka koden skulle det alltså (i princip) krävas att man kan faktorisera en produkt av två primtal som har 256 siffror, något som dagens datorer inte klarar.

En annan algoritm som är på frammarch är kryptering med elliptiska kurvor, så kallad ECC. Denna algoritm är också baserad på talteori, men medan säkerheten i RSA bygger på det faktum att det är svårt att faktorisera stora tal så bygger ECC på ett annat svårt matematiskt problem: att hitta den diskreta logaritmen i abelska grupper, närmare bestämt i en viss typ av grupp associerad med en elliptisk kurva över en kropp $\mathbb{Z}_q$. Fördelen med ECC är att nycklarna blir kortare än i RSA algoritmen.

Båda metoderna är såkallade *assymmetriska krypton*, vilket betyder att användarna har dels en hemlig nyckel, men också en öppen nyckel som publiceras offentligt. Med din öppna nyckel kan vem som helst kryptera meddelanden och skicka till dig, men endast du kan avkryptera koden och läsa meddelandet. Två personer kan därmed kommunicera med varandra (t.ex. via internet) på ett säkert sätt även om signalen avlyssnas av en tredje part.

Välj ett av projekten **RSA-kryptering** eller **ECC-kryptering** eller **Kodknäckning** nedan. I projekten kan du fritt använda följande funktion som hittar *en* lösning till den diofantiska ekvationen $ax + by = c$ med Euklides algoritm:

```
def diof(a,b,c): #Returnerar ett [x,y] så att ax+by=c
    [q,r] = [a//b,a%b]
    if r == 0: return([0,c//b])
    else:
        [u,v]= diof(b,r,c)
        return [v,u-q*v]
```

Regler för inlämning och samarbete

Projektet lämnas in till jonathn@chalmers.se **senast den 20 mars** - dagen innan tentan - i form av en enda Pythonfil PFörnamnEfternamn.py. Du ska förklara de viktigaste delarna av hur dina funktioner fungerar direkt i Pythonkoden med kommentarer, och på så vis visa att du förstår vad du har skrivit. Du ska också i kommentarerna svara på de frågor som ställs i instruktionstexten. För att bli godkänd måste dina funktioner klara precis det som specifiseras i instruktionen. Det är såklart inte tillåtet att direkt kopiera program från internet. Det är tillåtet att samarbeta, dock högst två personer. I fall man samarbetar kan man skicka in samma program, men alla kommentarer ska skrivas individuellt. Ange också namn på din samarbetspartner i kommentarerna.

Projekt I: RSA-kryptering

I det här projektet ska du implementera ett fungerande RSA-protokoll i Python. Projektet kan brytas upp i två delar som kan lösas i princip separat: omvandling mellan tal och strängar, och kryptering/dekryptering.

Uppgifter

Skriv en funktion **tillTal(text)** som tar en sträng "text" (med små bokstäver a-z och mellanrum) som argument och returnerar motsvarande sammanslagna heltal enligt följande tabell:

$$\text{"mellanrum"} \leftrightarrow 10 \quad a \leftrightarrow 11 \quad b \leftrightarrow 12 \quad \cdots \quad z \leftrightarrow 36$$

tillTal("hej du") ska alltså returnera talet 181520101431.
Skriv också en funktion **tillText(n)** som gör det motsatta: tar ett tal n och returnerar motsvarande textsträng. tillText(181520101431) ska alltså returnera strängen "hej du"

Tips: Eventuellt kan du skriva två hjälpfunktioner som gör om ett tecken till en bokstav och tvärtom. Prova att söka på hur de inbyggda funktionerna "ord" och "chr" fungerar i Python. Du kan plocka ut tecken ur en sträng likt en lista: $"hej"[0]$ ger h och så vidare. Längden av en sträng ges också med kommandot len("hej").

Uppgifter

Skriv en funktion **koda(a,k,n)** som tar ett tal a (meddelandet som ska kodas) och returnerar a^k modulo n som det krypterade meddelandet. Du kan anta att n är stort så att a<n.

Skriv också en funktion **avkoda(b,k,p,q)** som returnerar ett tal x som uppfyller $x^k = b$ modulo (pq). Tips: För att klara funktionen avkoda kan du behöva använda funktionen **diof** ovan.

Du kan testa att ditt program fungerar genom att i konsolen (eller längst ned i ditt program) sedan skriva:

```
p=73647592031827384997
p=937459375490384744791
n=p*q
k=894357834587
kod=koda(tillTal("en kort text"),k,n)
print(kod)
m=avkoda(kod,k,p,q)
print(tillText(m))
```

Sista raden ska då skriva ut det ursprungliga meddelandet. Om du har någon kurskamrat som gör samma projekt, så prova gärna att skicka ett hemligt kodat meddelande till dem tillsammans med en nyckel (p,q,k) och se om de kan avkoda meddelandet!

Projekt II - ECC kryptering

ECC kryptering är lite krångligare att implementera så vi använder specifika små siffror i det här projektet. Vi betraktar den elliptiska kurvan $\Omega : y^2 = x^3 - x + 1$ där x, y tillhör kroppen $\mathbb{Z}_{983}$ alltså heltal modulo $q = 983$. I tillämpningar behöver man använda ett mycket större q, och kan därefter associera hemliga meddelanden som x-koordinaten hos en punkt på kurvan - vi hoppar över denna del i projektet.

Vi identifierar punkter på kurvan med en lista $[x, y]$ med två heltal $0 \leq x, y < 983$ som uppfyller kurvans ekvation. Vi definierar också $\infty = [0, 0]$: detta är ett speciellt element som vi säger tillhör Ω även om det inte uppfyller ekvationen.

Uppgifter

- Skriv en funktion **påKurvan(P)** som testar om en punkt $P = [x, y]$ ligger på kruvan Ω. Funktionen ska returnera True eller False. Tänk på att påKurvan([0,0]) också ska ge True!
- Skriv en funktion **minus(P)** som tar en punkt $P = [x, y]$ och returnerar $-P = [x, -y\%983]$.
- Skriv en funktion **inv(a)** som returnerar inversen av a i ringen $\mathbb{Z}_q$. Vi ska alltså ha a*inv(a)=1 modulo 983. Tips: använd funktionen **diof** ovan!
- Skriv en funktion **add(P,Q)** som adderar två punkter på kurvan, och returnerar resultatet (alltså en tredje punkt på kurvan). Läs om hur man adderar punkter på en elliptisk kurva på https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication#Point_addition
  Notera att vi måste skilja på olika fall:
  - $\infty + P = P$ och $\infty + Q = Q$ för alla P,Q
  - $P + Q = \infty$ när $Q = minus(P)$
  - Annars ges additionen av formlerna på wikipedia: notera dock att det finns en separat formel för specialfallet $P = Q$.

  I formeln för att ta fram summan av två punkter behöver du dividera med tal. Eftersom vi arbetar i $\mathbb{Z}_q$ behöver du istället multiplicera med inversen till talet du vill dividera med.

Se till att ditt program fungerar innan du går vidare, du kan testa att det genom att skriva:

```
[P,Q,R]=[[182,781],[541,390],[182,38]]
print(inv(125))     #ska ge 810
print(påKurvan(P)) #ska ge True
print(påKurvan(Q)) #ska ge True
print(påKurvan(R)) #ska ge False
print(add(P,Q))     #ska ge [95,876]
print(add(P,P))     #ska ge [820, 785]
print(add(P,[0,0]))      #ska ge [182,781]
print(add(P,minus(P)))   #ska ge [0,0]
```

Uppgifter

- Skriv en funktion **mult(P,d)** som returnerar punkten $dP = P + P + \cdots + P$ (d termer).
- Skriv en funktion **koda(M,P,*x*)** som tar ett meddelande M (i form av en punkt på Ω), en öppen känd startpunkt P på Ω, och en hemlig nyckel x (positivt heltal) och returnerar punkten $M + xP$ (det kodade meddelandet).
- Skriv en funktion **avkoda(kod,P,*x*)** som avkodar meddelandet "kod" (en punkt på kurvan).

När du är klar ska du alltså kunna skriva
`hemlig=koda([123,130],[182,781],338)`
`avkoda(hemlig,[182,781],338)`
och få tillbaka den ursprungliga punkten [123,130] på kurvan.

Projekt III - Kodknäckning

I det här projektet jobbar du som en "hacker", därav den mystiska färgsättningen. Du försöker knäcka en RSA-kod med n=pq på 20 siffror. Du ska alltså skriva ett antal funktioner som kan faktorisera så stora tal som möjligt.

Uppgifter

- Kopiera dina funktioner ärPrimtal (rot versionen), primtalUnder, och FermatTest från Lab I och II, och gör sedan extrauppgiften från Labb 1: skapa en funktion **faktorisera(n)** som returnerar en sorterad lista med primtalsfaktorerna i n genom att utgå från dina tidigare funktioner. Testa hur stora tal funktionen klarar att faktorisera på rimlig tid.
- För att kunna testa dina funktioner är det bra att ha en snabb metod att skapa stora primtal. Kopiera in funktionen FermatTest från Labb 2 och skriv en ny funktion **primtalÖver(n)** som returnerar det minsta primtalet över n genom att använda FermatTest på $n, n+1, n+2, \ldots$. Vilket är det minsta primtalet över en gogool (10^{100})?
- Skriv en funktion **Pollard(n)** som tar ett sammansatt tal n och returnerar en faktorisering $[a, b]$ där $ab = n$ och $a, b < n$. För att hitta a och b ska funktionen använda *Pollard's* $p-1$*-algoritm*, du kan läsa om denna på wikipedia: https://en.wikipedia.org/wiki/Pollard%27s_p_%E2%80%93_1_algorithm#Algorithm_and_running_time
- Algoritmen på wikipedia innehåller ett par otydligheter: hur ska man välja a? Och hur ska man välja ett större respektive mindre B i algoritmen? Prova olika metoder och testa vilken som ger snabbast resultat när du multiplicerar två höga primtal. Motivera i kommentarerna vilka val du gjort till din slutgiltiga funktion.
- Skriv en funktion **Pfactor(n)** som returnerar en sorterad lista med primtalsfaktorerna i n. Funktionen ska klara att snabbt faktorisera produkten av två 10-siffriga primtal. Funktionen bör använda sig av funktionen Pollard(n) för att bryta upp stora faktorer i mindre delar, och funktionen faktorisera för att hantera mindre faktorer (t.ex. tal under 10000). På samma vis är det bättre att använda ärPrimtal än FermatTest för att primtalstesta mindre tal.