Programmering i Python

Innehåll

Vad är en utvecklingsmiljö?

En utvecklingsmiljö består av Python-tolk, kommando-skal, editor, *debugger*, med mera som ger dig möjlighet att skriva, exekvera (köra) och felsöka dina program. Traditionellt har dessa varit separata program som används var för sig.

Numera är dessa program oftast integrerade i en gemensam utvecklingsmiljö, en *Integrated Development Environment* (IDE), för att underlätta programutveckling. Det finns flera olika kostnadsfria IDE att välja mellan – exempelvis *IDLE*, *Thonny* och *PyCharm* Community.

Installera en utvecklingsmiljö

Välj **en** av följande utvecklingsmiljöer och installera den. Välj *IDLE* om du är osäker.

i. ***IDLE***
Öppna https://www.python.org/, välj *Downloads*, ladda ned *Python 3.x.y* för operativsystemet på din dator och följ instruktionerna för att genomföra installationen. I skrivande stund finns *Python 3.10.5* att tillgå.

ii. ***Thonny***
Öppna https://thonny.org/, ladda ned *Thonny 3.x.y* för operativsystemet på din dator och följ instruktionerna för att genomföra installationen. I skrivande stund finns *Thonny 3.3.13* att tillgå.

iii. ***PyCharm Community***
Öppna https://www.jetbrains.com/pycharm/, välj *Download*, ladda ned *PyCharm Community* för operativsystemet på din dator och följ instruktionerna för att genomföra installationen. I skrivande stund finns *PyCharm 2022.1.3* att tillgå.

Hello World

För att prova att utvecklingsmiljön fungerar brukar man skriva ett så kallat ”Hello World”-program. Skapa en ny fil i editorn i din utvecklingsmiljö. Skriv detta korta program:

```
print("Hello World")
```

Spara din fil med namnet `hello_world.py`. Python-program ska alltid ha filändelsen `.py` och en bra utvecklingsmiljö hjälper dig med detta.

Exekvera programmet från din utvecklingsmiljö. Utvecklingsmiljön startar då Python som tolkar ditt program och skriver ut `Hello World` i sitt kommando-skal. Om du i stället får ett felmeddelande så har du troligen skrivit fel i programmet. Kontrollera det noga och prova igen. Fungerar det fortfarande inte så har din utvecklingsmiljö troligen inte installerats korrekt.

Kommentarer

Allt som skrivs till höger om ett '#' är en kommentar.

```
# <-- Detta tecken (hashmark) påbörjar en kommentar som Python struntar i.
```

Utskrifter till skärmen

Det finns flera inbyggda funktioner i Python. En av dessa funktioner skriver ut text, tal och värden på variabler till skärmen och heter `print()`. Vi provade denna funktion i förra kapitlet. Man kan skriva ut flera saker samtidigt genom att ange dessa med komman mellan varje.

```
print("a har värdet", a, "och b har värdet", b)

print(c) # Observera att decimaltal visas med decimalpunkt: inte decimalkomma.

print(d, e)
```

Tilldela värden till variabler

Variabler kan användas ungefär som i matematikens algebra. Variabeln kan tilldelas olika värden, d v s värdet varierar, därav namnet variabel. När en variabel tilldelas ett värde så får variabeln en viss datatyp beroende på vad värdet är.

Ett lika-med-tecken (=) anger att variabeln som står till vänster tilldelas värdet av det som står till höger. Observera att det inte är samma sak som en ekvation i matematiken.

```
a = 3     # Variabeln a tilldelas värdet 3 och får datatypen 'int' (heltal).

b = 2.0   # Variabeln b tilldelas värdet 2.0 och får datatypen 'float' (decimaltal).

c = a / b # Variabeln c tilldelas värdet 1.5 och får typen 'float' (decimaltal).

          # Observera att decimaltal anges med decimalpunkt - inte decimalkomma.

d = "Hej" # Variabeln d tilldelas värdet "Hej" och får datatypen 'str' (sträng).
```

Beräkningar

Man kan göra beräkningar med heltal, decimaltal eller båda med operatorerna för addition, subtraktion, multiplikation och division: +, -, * och /

```
print(a+b, a-c, a*b, a/c)
```

Exponenter anges med dubbla multiplikationstecken, d v s **

```
print(a**4)
```

Felmeddelanden

Man kan inte göra beräkningar med värden av olika datatyper, exempelvis heltal och strängar:

```
print(a+d)
```

Vid programkörning fås då felet:

```
    File "errors.py", line 666, in <module>
      print(a+d)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Felmeddelandet säger att man inte kan addera heltal och strängar.

Även erfarna programmerare tänker fel eller skriver fel när de programmerar. Då och då berättar Python-tolken vad man gjort för fel. Felmeddelanden hör helt enkelt ihop med programmering.

OBS! Läs felmeddelanden noggrant för de hjälper dig att rätta felen.

Inmatning från tangentbordet

Inmatning från tangentbordet kan göras genom att anropa en annan inbyggd funktion i Python, nämligen `input()`. Observera att det som skrivs på tangentbordet alltid kommer in som en sträng.

Man kan ange en liten text som berättar för användaren vad den bör göra:

```
n = input("Vad heter du? ")
print("Hej", n)
```

Ändra variabelns datatyp

Om man behöver ett heltal istället för en sträng så gör man om strängen till ett heltal med den inbyggda funktionen `int()`.

```
x = input("Hur gammal är du? ")
a = int(x)
print(a, "år är en bra ålder")
```

Om man behöver ett decimaltal istället för en sträng så gör man om strängen till ett decimaltal med den inbyggda funktionen `float()`. Observera att decimaltal anges med decimalpunkt - inte decimalkomma.

```
x = input("Hur mycket väger du? ")
v = float(x)
print("Jaha, du väger", v, "kg")
```

Olika typer av fel

Även om man skrivit programmet korrekt så kan det bli fel vid programkörning, ett så kallat exekveringsfel (*runtime error*). Kör programmet och prova att skriva in din vikt med decimalkomma.

```
Traceback (most recent call last):
  File ".\errors.py", line 666, in <module>
    v = float(x)
ValueError: could not convert string to float: '85,3'
```

Felmeddelandet säger att strängen inte kan göras om till ett decimaltal.

Förutom exekveringsfel kan man ha syntaxfel i sin källkod. Syntaxfel är sådana fel som bryter mot Pythons definition, dvs man använder inte språket korrekt. Exempel: fel användning av operationer kan ge `TypeError`.

```
x = "1"
x = x ++ x
```

Exempel: fel namn på en funktion ger `NameError` eftersom namnet på funktionen är okänt dvs funktionen finns inte.

```
x = "1"
a = inte(x)
```

Ett indrag visar att koden med samma indentering hör till samma kodblock. Kodblock behövs i villkor, slingor och funktioner. Mer om detta längre fram i detta material.

```
# Exempel: fel indentering (indrag) ger IndentationError
x = "1"
    a = int(x)
```

Villkor

Vi har hittills skrivit program som körs ovillkorligen från första kodrad till sista kodrad. Det är sällan som det är tillräckligt. Ofta behöver man göra olika saker i programmet beroende på vilka villkor som är uppfyllda eller inte uppfyllda.

Ett exempel med myndighetsålder:

```
age = int(input('Hur gammal är du? '))
if age < 18: # Villkoret är sant om åldern är lägre än 18 år.
    # Här börjar kodblocket för villkoret. Kodblocket har en indentering.
    print('Du är inte myndig.')
```

Vägval

Vi kan utöka med ett alternativ där åldern inte är lägre än 18 år. Det finns endast ett villkor och endast ett av de två vägvalen görs.

```
age = int(input('Hur gammal är du? '))
if age < 18: # Villkoret är sant om åldern är lägre än 18 år.
    print('Du är inte myndig.')
else: # Om villkoret inte är sant görs detta vägval istället.
    print('Du är myndig.')
```

Fler villkor och vägval

Vi kan utöka med fler villkor. Endast ett vägval görs eftersom villkoren provas ett i sänder. Om något villkor inte är uppfyllt så fortsätter programmet att prova nästa. Om ett villkor är uppfyllt så görs det vägvalet och programmet fortsätter sedan utan att prova fler villkor.

```
age = int(input('Hur gammal är du? '))
if age < 6:
    print('Du kan redan läsa fast du är så ung!')
elif age < 13:
    print('Du går väl i skolan då.')
elif age < 20:
    print('Tonåring!')
elif age < 65:
    print('Jobbar och sliter?')
```

Jämförelser

Tecknet < kallas jämförelseoperator. Det finns fler jämförelseoperatorer:

```
<   mindre än
>   större än
<=  mindre än eller lika med
>=  större än eller lika med
==  lika med (OBS dubbla lika-med-tecken)
!=  skiljt från
```

Vi kan med hjälp av dessa jämförelseoperatorer i våra villkor skapa det programflöde vi vill ha, det vill säga de vägval som ska göras beroende på vilka villkor som uppfylls. Om inget av villkoren uppfyllts så utförs det kodblock som står efter `else`.

```
number = int(input('Skriv ett heltal, vilket som helst: '))
if number < 0:
    print(number, 'är negativt')
elif number == 0:
    print(number, 'är lika med noll')
elif number > 0:
    print(number, 'är positivt')
else:
    print('Detta alternativ är omöjligt!')
```

Upprepa kodrader

Om vi behöver repetera något i vårt program så kan vi göra det genom att upprepa kodrader. Om vi exempelvis vill skriva ut alla tal mellan 1 och 10 så kan vi skriva så här:

```
print(1)
print(2)
print(3)
print(4)
print(5)
print(6)
print(7)
print(8)
print(9)
print(10)
```

Vi kan i praktiken göra det om det gäller väldigt få repetitioner. Men om det gäller fler repetitioner än tio, till exempel tiotusen? Då behövs tiotusen kodrader för att göra detta! Det tar lång tid att skriva, lång tid att få rätt, programmet blir onödigt långt och det blir inte så flexibelt.

Slingor med `for`

Det finns dock alternativ: slingor (*loops*) som repeterar något flera gånger. Vi låter Python göra det repetitiva jobbet åt oss. Om vi vet hur många gånger något ska repeteras så använder vi lämpligen en `for`-loop. Om vi exempelvis vill skriva ut alla tal mellan 1 och 10 så kan vi skriva så här:

```
for n in range(1, 11, 1):
    print(n, end=' ')
print('\n----')
```

Genom att skriva 2 rader kod kan vi utföra 10 repetitioner. Vad gör koden då?

```
for                 Anger att vi vill repetera något.
n                   Är en variabel som hålla reda på vilken repetition.
in range(1, 11, 1)  Ger värden från och med 1 till 11 med steglängden 1.
print(n, end=' ')   Skriver ut varje värde på n, dvs vilken repetition det
                    är, och lägger till ett mellanslagstecken utan ny rad.
```

När koden körs börjar n på 1, n skrivs ut, n ökar med steglängden 1 till 2, n skrivs ut, n ökar med steglängden 1 till 3, och så vidare tills dess n blivit 11 vilket gör att repetitionen avbryts. Då har alla talen mellan 1 och 10 skrivits ut. Det kan verka underligt att vi skriver `range(1, 11, 1)` istället för `range(1, 10, 1)` men kom ihåg att värdet 11 inte är det sista värdet utan det värde där repetitionen avbryts.

Om vi exempelvis vill skriva ut alla tal mellan 1 och 100 så behöver vi endast ändra koden en smula: `11` ändras till `101`.

```
for n in range(1, 101, 1): # 1 2 3 ... 98 99 100
    print(n, end=' ') # Skriv ut n och sedan ett mellanslagstecken
print('\n----') # Ny rad och en tydlig avskiljare
```

Användning av `range()`

En `for`-loop kan skrivas på flera olika sätt beroende på vad som behövs. Om startvärde utelämnas startar `for`-loopen på 0.

```
for n in range(5): # 0 1 2 3 4
    print(n, end=' ') # Skriv ut n och sedan ett mellanslagstecken
print('\n----') # Ny rad och en tydlig avskiljare
```

Om steglängden utelämnas används steglängden 1.

```
for n in range(1, 11): # 1 2 3 4 5 6 7 8 9 10
    print(n, end=' ') # Skriv ut n och sedan ett mellanslagstecken
print('\n----') # Ny rad och en tydlig avskiljare
```

Steglängden kan exempelvis vara 2.

```
for n in range(1, 6, 2): # 1 3 5
    print(n, end=' ') # Skriv ut n och sedan ett mellanslagstecken
print('\n----') # Ny rad och en tydlig avskiljare
```

Man kan börja högt, sluta lågt och negativ steglängd.

```
for n in range(10, 5, -2): # 10 8 6
    print(n, end=' ') # Skriv ut n och sedan ett mellanslagstecken
print('\n----') # Ny rad och en tydlig avskiljare
```

Nästlade slingor

Man kan även ha slingor i slingor, det kallas nästlade slingor.

```
for m in range(1, 9, 1): # 1 2 3 4 5 6 7 8
    for n in range(1, 9, 1): # 1 2 3 4 5 6 7 8
        print(m*n, end='\t') # Skriv ut produkterna m*n i varsin kolumn.
    print() # Ny rad
```

Slingor med `while`

Om man inte vet hur många gånger man behöver repetera något så kan man använda en `while`-loop för att repetera. Det kräver dock att man använder ett villkor och att variabeln som finns i villkoret kan ändras inuti slingan så att slingan avbryts. Gör man inte detta på rätt sätt kan man få en så kallad evig loop.

Ett vanligt fall när `while`-loop är att föredra är när antal repetitioner påverkas av en användare av programmet och vi därför inte vet hur många gånger vi ska repetera slingan.

I nedanstående exempel läses först ett värde från tangentbordet och villkoret i `while`-loopen provas.

```
 i) om värdet är 0 så uppfylls inte villkoret och inget mer görs.
ii) om värdet är skiljt från 0 så görs en utskrift, ett nytt värde läses
    från tangentbordet och villkoret provas igen.
```

Sedan kan ii) repeteras ett okänt antal gånger tills i) inträffar.

```
a = float(input("Skriv 0 för att avsluta: "))
while a != 0.0:
    print("Kvadraten av", a, "=", a*a)
    a = float(input("Skriv 0 för att avsluta: "))
```

Slingor sparar tid för programmeraren, minskar risken för fel samt gör koden mer lättläst och flexibel.

Definiera egna funktioner

Vi har provat att anropa inbyggda funktioner, t ex `print()` och `input()`. I många fall kan vi dra nytta av att skriva våra egna funktioner, speciellt för att lösa matematiska problem. Funktioner i Python kan skrivas så de liknar matematiska funktioner.

Men först ett grundläggande exempel:

```
def f(): # Vi definierar först funktionen f.
    # Här börjar kodblocket för funktionen. Kodblocket har en indentering.
    print('Running f()') # Vi låter funktionen berätta att den kör.
f() # Vi anropar (kallar på) funktionen så att den körs.
```

En funktion kan även innehålla parametrar. Sådana parametrar är variabler som kan användas inuti funktionen. Parametrar anges inom parentesen för funktionen:

```
def g(x): # Funktionen g med parametern x.
    print(x*x) # Vi låter funktionen skriva ut produkten av x i kvadrat.
g(3) # Vi anropar funktionen med argumentet 3.
```

Men oftast vill vi inte skriva ut resultatet i varje funktion utan i stället låta funktionen utföra beräkningen och svara med resultatet. Detta kallas att returnera resultatet. Det returnerade resultatet kan vi sedan använda på det sätt vi vill, exempelvis tilldela till en variabel eller skriva ut.

```
def h(x):
    return x*x
s = h(2) # Tilldela variabeln s värdet av h(2).
print(h(1.34164078649987378)) # Skriv ut värdet av h(1.34164078649987378).
```

Ta som exempel den räta linjens funktion: y = kx + m. Vi kan ganska enkelt överföra den till Python:

```
def y(k, x, m): # Vi definierar funktionen y med dess parametrar k, x och m.
    return k*x + m # Vi låter funktionen beräkna kx + m och returnera svaret.
```

För att använda vår nya funktion behöver vi anropa den med värden på de parametrar som vi vill beräkna, till exempel k = 2, x = 1.5 och m = -1 :

```
print(y(2, 1.5, -1.2))
```

Vi kan använda funktionen för att beräkna en punkt på en annan linje:

```
a = y(2, 3, 3)
print(a)
```

Programexempel: Multiplicera med addition

Problem:

Multiplikation av heltal kan ses som en upprepad addition. Exempelvis är 3·5 = 5+5+5. Skriv ett program som beräknar produkten av två positiva heltal med hjälp av addition.

Analys:

Vilka är indata? Två tal a och b som kan matas in via tangentbordet.
Vilket resultat förväntas? Produkten p av de två talen a och b där produkten beräknas genom upprepad addition.
Vad behöver vi? Inmatning av talen a och b. En slinga som repeterar a gånger och adderar b till p. Utskrift av p.

Skapa en ny fil för att prova följande lösningsförslag:

```
a = int(input('Det första talet: '))
b = int(input('Det andra talet: '))
p = 0
while a > 0:
    p = p + b
    a = a - 1
print(p)
```

Vad gör programmet i detalj? Gör programmet rätt?

Programexempel: Vilka heltal är jämna?

Problem:

Undersök vilka tal från och med 1 till och med 20 som är jämna.

Analys:

Vilka är indata? Två tal 1 och 20.
Vilket resultat förväntas? Alla jämna tal från och med 1 till och med 20.
Vad behöver vi? Om vi dividerar ett jämnt tal med 2 så får vi ett heltal utan decimaler. Om vi dividerar ett udda tal med 2 kommer vi däremot alltid att få ett decimaltal som svar (det kommer att sluta på ,5). För att undersöka om svaret blir ett heltal jämför vi svaret med och utan decimaler. Om dessa två är lika är det ett heltal.

Skapa en ny fil för att prova följande lösningsförslag:

```
for n in range(1, 21):
    q = n / 2
    if q == int(q):
        print(n)
```

Vad gör programmet i detalj? Gör programmet rätt?

Finns det någon enklare lösning? I så fall, hur ser den lösningen ut?

Programexempel: Summera positiva heltal

Prolem:

Skriv en funktion som räknar ut summan av alla positiva heltal från 1 till n.

Analys:

Vilka är indata? Ett tal n som kan matas in via tangentbordet.
Vilket resultat förväntas? Summan av alla positiva heltal 1..n.
Vad behöver vi? En funktion som tar ett argument n, repeterar ett tal 1..n, adderar talet till summan och returnerar summan. Inmatning av talet n. Anrop av funktionen. Utskrift av summan.

Skapa en ny fil för att prova följande lösningsförslag:

```
def summa(n):
    s = 0
    for t in range(1, n+1):
        s = s + t
    return s
n = int(input('Tal: '))
print(summa(n))
```

Vad gör programmet i detalj? Gör programmet rätt?

Programexempel: Vilket x ger att f(x) = g(x)?

Problem:

Vi har en funktion f(x) = 3x - 5 och en funktion g(x) = -2x + 7.
Visa vilket värde på x som ger f(x) = g(x).

Analys

Problemet är av typen linjärt ekvationssystem som kan lösas grafiskt genom att rita x, f(x) och g(x) i samma koordinatsystem. Där funktionerna graf skär varandra finns lösningen. Det kan också lösas genom att pröva olika värden på x tills dess f(x) blir lika med g(x).
Det är den senare metoden som vi ska använda när vi programmerar lösningen. Vi provar helt enkelt en serie med värden på x och ser när f(x) blir lika med g(x). Denna metod utan finess kallas "brute force".

Skapa en ny fil för att prova följande lösningsförslag:

```
def f(x):
    return 3*x - 4
def g(x):
    return -2*x + 6
for x in range(-5, 6):
    if f(x) == g(x):
        print('x =', x, 'ger f(x) = g(x) =', f(x))
```

Vad gör programmet i detalj? Gör programmet rätt?

Programexempel: Är f(x) oändlig?

Problem:

Vi har en funktion f(x) = 1/x för alla x skiljt från 0 (noll.
Visa att f(x) går mot oändligheten när x går mot noll.

Analys:

Vi ser på funktionen att x behöver vara väldigt liten för att gå mot oändligheten men den får inte bli 0 (noll). Inom matematiken talar man om att x har ett gränsvärde som är skiljt från 0 (noll).
För att visa att funktionen går mot oändligheten behöver vi anropa funktionen f(x) med ett x som blir mindre och mindre. Vi kan inse att vi bör skapa en slinga för detta samt skriva ut både x och f(x).
Vi vet också att en for-loop inte kan använda decimaltal (float) så vi väljer en while-loop i stället.

Skapa en ny fil för att prova följande lösningsförslag:

```
def f(x):
    return 1/x
d = 1e-12
x = 1
while x >= d:
    print('x={:.12f}, f(x)={:.0f}'.format(x, f(x)))
    x = x / 2
```

Vad gör programmet i detalj? Gör programmet rätt?

Användning av moduler

Förutom alla de inbyggda funktionerna i Python så kan man även nyttja moduler med ytterligare funktioner. Vid installation av Python får man tillgång till en mängd moduler, bland annat för matematik och grafik, i ett så kallat standardbibliotek.

Det finns även moduler som inte är en del av Pythons standardbibliotek. Sådana behöver installeras på våra datorer innan vi kan använda dem. Det kan man göra så här för exempelvis `matplotlib`:

1. Öppna ett kommandoskal i ditt operativsystem (t ex PowerShell i Windows eller bash i Linux).
2. Skriv så här i kommandoskalet: `pip install matplotlib`
3. Tryck Enter och låt installationen av `matplotlib` färdigställas.

För att få tillgång till modulerna behöver man först importera dem i sina egna program. Exempel på detta ges i följande kapitel.

Modulexempel: `math`

Exemplen här försöker visa på sätt att använda olika moduler.

Vi använder modulen `math` i det första exemplet. När vi importerat `math` kan vi sedan använda de konstanter och funktioner som finns i modulen. Vi gör det genom att skriva modulens namn, en punkt och sedan namnet på den konstant/funktion vi vill använda.

Skapa en ny fil som heter `module_math.py` för att prova exemplet.

```
import math

print(math.pi)              # Konstanten Pi (ca 3.14159)

print(math.gcd(12, 15))     # Största gemensamma delaren för 12 och 15.

print(math.log(math.e))     # Naturliga logaritmen av konstanten e (ca 2.71828).

print(math.sin(math.tau))   # Sinus av två Pi radianer.
```

Modulexempel: `random`

I det andra exemplet använder vi modulen `random`. Det är en modul som kan skapa slumptal. Skapa en ny fil som heter `module_random.py` för att prova exemplet.

```
import random
print(random.randint(1,6)) # Slumpa fram ett tal 1-6.
c = [2, 3, 4, 5, 6, 7, 8, 9, 10, 'J', 'Q', 'K', 'A'] # En del av en kortlek.
for _ in range(7):          # Vi ska kupera korten sju gånger.
   random.shuffle(c)        # Kupera korten slumpmässigt.
   print(c)                 # Visa korten.
```

Modulexempel: `turtle`

I det tredje exemplet använder vi modulen `turtle`. Det är en grafik-modul som gör det enkelt att rita ut figurer och streck på skärmen. Det heter `turtle` eftersom man låter en sköldpadda vandra över skärmen och dra en penna med sig för att rita de figurer man vill ha. Här får sköldpaddan rita en kvadrat.

Skapa en ny fil som heter 'module_turtle.py' för att prova exemplet.

```
import turtle               # Importera modulen.
t = turtle.Turtle()         # Turtle() skapar vår sköldpadda.
for i in range(4):          # Vi repeterar för varje sida av kvadraten.
    t.forward(50)           # Sköldpaddan går 50 små steg framåt.
    t.right(90)             # Sköldpaddan svänger 90 grader åt höger.
turtle.done()               # Sköldpaddan är färdig!
```

Modulexempel: `matplotlib`

Vi ska nu plotta en funktion f(x) = 5x - 2 för -10 < x < 10 med hjälp av `matplotlib`. Skapa en ny fil som heter `module_matplotlib_1.py` för att prova exemplet.

```
from matplotlib.pyplot import *
def f(x):           # Definierar funktionen
    return 5*x - 2
x = []              # Skapar en tom lista för kategorivärden på x-axeln
y = []              # Skapar en tom lista för funktionsvärden på y-axeln
for n in range(-9, 10):
    x.append(n)     # Lägger till kategorivärde
    y.append(f(n))  # Lägger till funktionsvärde
plot(x, y)          # Plottar kategori- och funktionsvärden
show()              # Visar grafen
```

Vi ska nu plotta en andragradsfunktion f(x) = x^2 - 5x - 20 för -10 < x < 10 med hjälp av `matplotlib`. Skapa en ny fil som heter `module_matplotlib_2.py` för att prova exemplet.

```
from matplotlib.pyplot import *
def f(x):           # Definierar funktionen
    return x**2 - 5*x - 20
x = []              # Skapar en tom lista för kategorivärden på x-axeln
y = []              # Skapar en tom lista för funktionsvärden på y-axeln
for n in range(-9, 10):
    x.append(n)     # Lägger till kategorivärde
    y.append(f(n))  # Lägger till funktionsvärde
plot(x, y)          # Plottar kategori- och funktionsvärden
show()              # Visar grafen
```