# Rosetteprogrammering

Alvin Gavel

Som en del av mitt nya jobb på KIB så kommer jag att behöva lära mig ett par nya programspråk. De mest brådskande är *R* och *PHP*, men även *bash* kommer säkert visa sig bra att kunna. Som övning har jag bestämt mig för att välja ut ett par enkla programmeringsövningar och försöka lösa dem först i *Python*, som jag redan kan, och sedan i de tre andra språken. Jag lägger upp resultaten av övningarna allteftersom på Github i repositoriet `https://github.com/Alvin-Gavel/Comparative_coding`.

För varje övning finns en mapp med namnet `Foo` som innehåller bland skript med namn av formen `Foo.xy`, där `xy` är den typiska filändelsen för programspråket i fråga. Givet att du har python installerat så kan python-skripten köras med att i terminalen skriva in

```
python Foo.py x y z
```

där `x`, `y` och `z` är argumenten som krävs av funktionen i fråga. På samma sätt kan R-skripten köras med

```
Rscript Foo.R x y z
```

och PHP-skripten med

```
php Foo.php x y z
```

Att köra bash-skripten kan bli knepigare, men i bästa fall går det med

```
bash foo.sh x y z
```

För att göra det hela snabbare så finns även ett skript som heter `Test_all.sh` som kör allihopa med samma argument:

```
Test_all.sh x y z
```

Samtliga skript i mappen körs då med de givna argumenten, och borde ge exakt samma respons.

I vissa fall har jag skrivit ett par extra skript: De vanliga skripten har jag försökt göra så lika varandra som möjligt, för att det ska vara lätt att jämföra koden. I vissa fall har det krävt att jag gjort saker på onödigt bökiga sätt eftersom jag hållit mig till de konstruktioner som finns i de flesta språk. Ibland har jag därför lagt till ett skript med namnet `Foo_optimised.x` som drar bättre nytta av olika behändiga konstruktioner som är kännetecknande för just det språket. Ibland har jag istället lagt till ett skript med namnet `Foo_using_bar.x` där `bar` är något vanligt bibliotek som man verkligen borde använda i sammanhanget.

# Innehåll

# 1 Fibonaccital

Den första övningen var att skriva funktioner som givet ett tal $n$ skriver ut fibonaccitalet $F_n$, alltså de tal som defineras genom

$$F_1 = 1 \tag{1}$$
$$F_2 = 1 \tag{2}$$
$$F_n = F_{n-1} + F_{n-2}, \qquad (n > 2). \tag{3}$$

Lite godtyckligt bestämde jag att funktionerna ska ge 1 som svar på alla $n < 1$.

## 1.1 Python

I filen `Fibonacci.py` har jag implementerat den här ganska generella lösningen:

```
#!/usr/local/bin/python3
import sys

def fibonacci(n):
   F_n = 1
   F_nm1 = 0
   for i in range(n):
     F_old = F_n
     F_n = F_n + F_nm1
     F_nm1 = F_old
   print(F_nm1)
   return

n = int(sys.argv[1])
fibonacci(n)
```

Denna lösning drar dock inte nytta av en fiffig egenskap hos Python. Det går att göra flera variabeldefinitioner på samma rad, och man kan ibland utnyttja det för att slippa krångel med temporära variabler. Tillexempel så är det möjligt att låta variablerna `foo` och `bar` byta värden genom att helt enkelt skriva `foo, bar = bar, foo`. Vi kan alltså korta ner lösningen ovan till den lite smidigare versionen i `Fibonacci_optimised.py`:

```
#!/usr/local/bin/python3
import sys

def fibonacci(n):
   F_n = 1
   F_nm1 = 0
   for i in range(n):
     F_n, F_nm1 = F_n + F_nm1, F_n
   print(F_nm1)
   return

n = int(sys.argv[1])
fibonacci(n)
```

## 1.2 R

Lösningen i R är nästan helt identisk med den första av lösningarna i Python.

```
#!/usr/bin/env Rscript

fibonacci <- function(n) {
   F_n <- 1
   F_nm1 <- 0
   for (i in 1:n) {
      F_old <- F_n
      F_n <- F_n + F_nm1
      F_nm1 <- F_old
   }
   cat(F_nm1, '\n')
}

n <- commandArgs(trailingOnly=TRUE)[1]
fibonacci(n)
```

Den enda rad i Python-koden som inte har en direkt motsvarighet i R-koden är returnsatsen i slutet på funktionsdefinitionen, och det är mest en stilfråga. Det skulle gå att ta bort returnsatsen ur Python-koden och den skulle fortfarande gå att köra, och det skulle gå att lägga till en tom returnsats till R-koden.

## 1.3 PHP

Även PHP ser ungefär likadant ut som Python. Den mest synbara skillnaden är mängden dollartecken och semikolon som dykt upp. Man kan också se att forloopen är mer explicit med när den ska ta slut och vad som sker med indexet i varje iteration. Det gör å ena sidan språket lite mer flexibelt, men samtidigt att det blir lite mer plottrigt.

```
<?php

function fibonacci($n) {
   $F_n = 1;
   $F_nm1 = 0;
   for ($i = 0; $i < $n; $i++) {
     $F_old = $F_n;
     $F_n = $F_n + $F_nm1;
     $F_nm1 = $F_old;
   }
   echo "$F_nm1\n";
}

fibonacci($argv[1]);
?>
```

## 1.4 bash

Tillsist löser vi problemet i bash.

```
#!/usr/bin/env bash

fibonacci() {
  n=$1
  local F_n=1
  local F_nm1=0
  for i in $(seq 1 $n)
  do
    F_old=$F_n
    F_n=$(($F_n+$F_nm1))
    F_nm1=$F_old
  done
  echo $F_nm1
}

fibonacci $1
```

Lösningen ser ungefär likadan ut som i de andra språken, men någonting som inte framgår är hur kinkigt språket är. Tillexempel gör det i Python ingen skillnad om man skriver `foo = 1` eller `foo=1`. bash däremot kräver att variabeltilldelningen sker utan några mellanslag. Den kräver också dubbla parenteser runt beräkningen av $F_n + F_{n-1}$.

# 2 Primtalstest

Den andra övningen var att skriva funktioner som givet ett tal $n$ testar om talet är ett primtal och sedan skriver ut svaret i form av en fullständig mening. Som primtalstestningsalgoritm använder vi Eratosthenes såll. Det vill säga att vi testar att dividera med alla heltal som skulle kunna vara delare.

## 2.1 Python

Den enklaste tänkbara lösningen finns i `Primality_test.py`:

```
#!/usr/local/bin/python3
import sys

def isprime(n):
   found_divisor = False
   if n <= 1:
      return False
   elif n == 2:
      return True
   else:
      for i in range(2, n):
         found_divisor = found_divisor or n % i == 0
      return not(found_divisor)

def verbose_answer(n):
   wasit = isprime(n)
   str_not = '' if wasit else ' not'
   print("{} is{} prime".format(n, str_not))
   return

verbose_answer(int(sys.argv[1]))
```

Det är dock inte helt optimalt att vi testar att dela med alla heltal upp till och med $n - 1$. Vi vet ju att om vi tagit oss upp till $\sqrt{n}$ utan att hitta någon delare, då kommer inget tal större än så att vara en delare heller. I kodfilen `Primality_test_using_math.py` använder jag biblioteket `math` som innehåller en kvadratrotsfunktion. Funktionen `isprime` får då detta utseende:

```
import math

def isprime(n):
   found_divisor = False
   if n <= 1:
      return False
   elif n == 2:
      return True
   else:
      for i in range(2, int(math.ceil((math.sqrt(n)))) + 1):
         found_divisor = found_divisor or n % i == 0
      return not(found_divisor)
```

## 2.2 R

Lösningen i R kräver inte att vi importerar något särskilt bibliotek, eftersom språket är skrivet av och för matematiker och kvadratrotsfunktionen därför redan finns bland standardfunktionerna.

```
#!/usr/bin/env Rscript

isprime <- function(n) {
   found_divisor = FALSE
   if (n <= 1) {
      return(FALSE)
   } else if (n == 2) {
      return(TRUE)
   } else {
      for (i in 2:ceiling(sqrt(n))) {
         found_divisor <- found_divisor | n %% i == 0
      }
      return(!found_divisor)
   }
}

verbose_answer <- function(n) {
   wasit <- isprime(n)
   str_not <- if (wasit) '' else 'not '
   cat(paste0(n, " is ", str_not, "prime\n"))
}

n <- as.numeric(commandArgs(trailingOnly=TRUE)[1])
verbose_answer(n)
```

## 2.3 PHP

Lösningen för PHP ser ungefär likadan ut som den i R:

```
<?php

function isprime($n) {
   $found_divisor = false;
   if ($n <= 1) {
      return false;
   } else {
      for ($i = 2; $i <= sqrt($n); $i++) {
         $found_divisor = ($found_divisor or $n % $i == 0);
      }
      return !$found_divisor;
   }
}

function verbose_answer($n) {
   $wasit = isprime($n);
   $wasit ? $str_not = '' : $str_not = ' not';
   echo "$n is$str_not prime\n";
}

verbose_answer($argv[1]);
?>
```

Den enda väsentliga skillnaden syns i forloopen: I R itererar vi över en lista med värden, och när den listan definieras så krävs `ceiling`-funktionen för att göra den övre gränsen till ett heltal. I PHP så räcker det med att specificera att indexet måste vara under ett visst värde.

## 2.4 bash

I min lösning i bash har jag inte ens vågat mig på att försöka beräkna en kvadratrot. Det går säkert att göra, men språket är inte till för att göra någon form av beräkningar. Istället blir lösningen ekvivalent med den första av Python-lösningarna:

```
#!/usr/bin/env bash

isprime() {
  n=$1
  local found_divisor=false
  if (($n <= 1))
  then
    echo "0"
  elif [ $n == 2 ]
  then
    echo "1"
  else
    for i in $(seq 2 $[$n-1])
    do
      found_divisor=$[$found_divisor || $[$[$n % $i] == 0]]
    done
    echo "$[! $found_divisor]"
  fi
}

verbose_answer() {
  n=$1
  local wasit=$(isprime $n)
  if [ $wasit == 1 ]
  then
    local str_not=''
  else
    local str_not=' not'
  fi
  echo "${n} is${str_not} prime"
}

verbose_answer $1
```

# 3 Mandelbrotmängden

Den tredje övningen var att skriva funktioner som skriver ut mandelbrotmängden. Det vill säga, mängden av komplexa tal $c$ sådana att talföljden

$$z_n = z_{n-1}^2 + c \tag{4}$$
$$z_1 = c \tag{5}$$

inte divergerar när $n \to \infty$.

Mandelbrotmängden är en fraktal, så det är väsentligt med vilken upplösning den skrivs ut. Alltså måste funktionerna ta dimensionerna $x$ och $y$ längs med reella resp. imaginära axeln som argument. Det finns också ingen generell metod för att avgöra om en punkt kommer att divergera, utan oftast så beräknar man för varje $c$ helt enkelt något tal $z_n$ där $n$ är hyfsat stort och antar att om $|z_n| < 2$ så är talföljden förmodligen konvergent för det talet $c$. Alltså måste funktionerna också ta antalet iterationer $n$ som argument.

Det går inte riktigt att rita bilder i terminalfönster, men eftersom de brukar använda konstant teckenbredd så kan vi använda tecknen som pixlar, och låta asterisker beteckna punkterna där vi tror att talföljden är konvergent. Om vi kör någon av funktionerna med t.ex. $x = 81$, $y = 41$ och $n = 100$ så ska den skriva ut:

```
                                                  *
                                                 ******
                                                *******
                                                  ***       *
                                             **    ****************
                                             ****************************
                                            ***************************
                                           ******************************
                                          ********************************
                                          ********************************
                          ** *****       ************************************
                         ************   ************************************
                        *************** ************************************
                     ** *************** ************************************
**************************************************************************
                     ** *************** ************************************
                        *************** ************************************
                         ************   ************************************
                          ** *****       ************************************
                                           ******************************
                                          ********************************
                                           ******************************
                                            ***************************
                                             ****************************
                                             **    ****************
                                                       ***       *
                                                      *******
                                                       ******
                                                        *
```

## 3.1 Python

En lösning som bara håller sig till de grundläggande konstruktionerna i språket finns i kodfilen `Mandelbrot.py`. Den är inte snygg, men den får jobbet gjort:

```
#!/usr/local/bin/python3
import sys

def mandelbrot(x_steps, y_steps, iterations):

  min_x = -2.0
  max_x = 0.5
  span_x = max_x - min_x
  step_x = span_x / (x_steps - 1)

  min_y = -1.2
  max_y = 1.2
  span_y = max_y - min_y
  step_y = span_y / (y_steps - 1)

  complex_plane = []
  for x_pos in range(x_steps):
    re = min_x + x_pos * step_x
    column = []
    for y_pos in range(y_steps):
      im = min_y + y_pos * step_y
      column.append([re, im, re, im, False])
    complex_plane.append(column)

  for x_pos in range(x_steps):
    for y_pos in range(y_steps):
      for i in range(iterations):
        diverged = complex_plane[x_pos][y_pos][4]

        if not diverged:
          re = complex_plane[x_pos][y_pos][0]
          im = complex_plane[x_pos][y_pos][1]

          zi_re = complex_plane[x_pos][y_pos][2]
          zi_im = complex_plane[x_pos][y_pos][3]

          zi1_re = zi_re**2 - zi_im**2 + re
          zi1_im = 2 * zi_re * zi_im + im

          diverged = zi1_re**2 + zi1_im**2 > 4

          complex_plane[x_pos][y_pos][2] = zi1_re
          complex_plane[x_pos][y_pos][3] = zi1_im
          complex_plane[x_pos][y_pos][4] = diverged
```

```
  for y_pos in range(y_steps):
    row = ''
    for x_pos in range(x_steps):
      if complex_plane[x_pos][y_pos][4]:
        row += ' '
      else:
        row += '*'
    if '*' in row:
      print(row)
  return

x_steps = int(sys.argv[1])
y_steps = int(sys.argv[2])
iterations = int(sys.argv[3])
mandelbrot(x_steps, y_steps, iterations)
```

Det finns flera saker som gör den här lösningen osnygg, men de flesta av dem har sin grund i att vi bygger upp det komplexa talplanet som en lista med listor med listor. Vi skulle kunna snygga till det lite om vi använda språkets inbyggda stöd för komplexa tal. Vi skulle kunna göra det mycket snyggare om vi också hade stöd för matrisaritmetik. Vi får detta om vi importerar modulen `numpy`. Den innehåller en stor mängd användbara funktioner, och bör användas varje gång man gör någonting som är det minsta matematiskt komplicerat. I kodfilen `Mandelbrot_using_numpy.py` har vi den här mycket mer kompakta lösningen:

```
#!/usr/local/bin/python3
import sys

import numpy as np

def mandelbrot(x_steps, y_steps, iterations):

  min_x = -2.0
  max_x = 0.5

  min_y = -1.2
  max_y = 1.2

  real_axis = np.linspace(min_x, max_x, x_steps)
  imag_axis = np.linspace(min_y, max_y, y_steps)

  real_coords, imag_coords = np.meshgrid(real_axis, imag_axis)
  complex_plane = real_coords + 1j * imag_coords
  Z = real_coords + 1j * imag_coords

  for i in range(iterations):
    diverged = np.abs(Z) > 4
    Z[~diverged] = Z[~diverged]**2 + complex_plane[~diverged]

  for y_pos in range(y_steps):
    row = ''
```

```
    for x_pos in range(x_steps):
      if diverged[y_pos][x_pos]:
        row += ' '
      else:
        row += '*'
    if '*' in row:
      print(row)
  return

x_steps = int(sys.argv[1])
y_steps = int(sys.argv[2])
iterations = int(sys.argv[3])
mandelbrot(x_steps, y_steps, iterations)
```

Där beräkningen tidigare bestod av tre lager av forloopar som verkar på varje enskilt matriselement finns nu bara en forloop som verkar på matrisen som helhet. Vi har också lyckats korta ner steget att konstruera matrisen till att börja med.

### 3.2 R

I R är situationen ungefär likadan. Det går att lösa problemet med konstruktioner som finns i alla språk, men det blir inte snyggt. Min implementering ser ut såhär:

```
#!/usr/bin/env Rscript

mandelbrot <- function(x_steps, y_steps, iterations) {

  min_x <- -2.0
  max_x <- 0.5
  span_x <- max_x - min_x
  step_x <- span_x / (x_steps - 1)

  min_y <- -1.2
  max_y <- 1.2
  span_y <- max_y - min_y
  step_y <- span_y / (y_steps - 1)

  complex_plane <- list()
  for (x_pos in 1:x_steps) {
    re <- min_x + (x_pos - 1) * step_x
    column <- list()
    for (y_pos in 1:y_steps) {
      im <- min_y + (y_pos - 1) * step_y
      column <- append(column, list(list(re, im, re, im, FALSE)), y_pos)
    }
    complex_plane <- append(complex_plane, list(column), x_pos)
  }

  for (x_pos in 1:x_steps) {
    for (y_pos in 1:y_steps) {
      for (i in 1:iterations) {
        diverged <- complex_plane[[x_pos]][[y_pos]][[5]]

        if (!diverged) {
          re <- complex_plane[[x_pos]][[y_pos]][[1]]
          im <- complex_plane[[x_pos]][[y_pos]][[2]]

          zi_re <- complex_plane[[x_pos]][[y_pos]][[3]]
          zi_im <- complex_plane[[x_pos]][[y_pos]][[4]]

          zi1_re <- zi_re^2 - zi_im^2 + re
          zi1_im <- 2 * zi_re * zi_im + im

          diverged <- (zi1_re^2 + zi1_im^2 > 4)

          complex_plane[[x_pos]][[y_pos]][[3]] <- zi1_re
          complex_plane[[x_pos]][[y_pos]][[4]] <- zi1_im
          complex_plane[[x_pos]][[y_pos]][[5]] <- diverged
```

```
        }
      }
    }
  }

  for (y_pos in 1:y_steps) {
    row <- ''
    for (x_pos in 1:x_steps) {
      if (complex_plane[[x_pos]][[y_pos]][[5]]) {
        row <- paste(row, " ", sep="")
      } else {
        row <- paste(row, "*", sep="")
      }
    }
    if (grepl('*', row, fixed = TRUE)) {
      row <- paste(row, "\n", sep="")
      cat(row)
    }
  }
}

args <- commandArgs(trailingOnly=TRUE)
x_steps <- as.numeric(args[1])
y_steps <- as.numeric(args[2])
iterations <- as.numeric(args[3])

mandelbrot(x_steps, y_steps, iterations)
```

Med R är det dock möjligt att göra en snyggare lösning utan att använda några extrabibliotek, eftersom R redan har stöd för både komplexa tal och matrisalgebra. I `Mandelbrot_optimised.R` finns den här mycket kompaktare lösningen:

```
#!/usr/bin/env Rscript

mandelbrot <- function(x_steps, y_steps, iterations) {

  min_x <- -2.0
  max_x <- 0.5

  min_y <- -1.2
  max_y <- 1.2

  real_axis <- seq(from = min_x, to = max_x, length.out = x_steps)
  imag_axis <- seq(from = min_y, to = max_y, length.out = y_steps)
  complex_plane_temp <- complex(real = rep(real_axis, times = 1, each = y_steps
      ),
                        imag = rep(imag_axis, times = x_steps, each = 1))
  complex_plane <- matrix(complex_plane_temp, x_steps, y_steps)
  Z <- matrix(complex_plane_temp, x_steps, y_steps)
```

```
  for (i in 1:iterations) {
     Z <- Z^2 + complex_plane
     diverged <- (abs(Z) > 4)
    }

  for (y_pos in 1:y_steps) {
     row <- ''
     for (x_pos in 1:x_steps) {
        if (diverged[y_pos, x_pos]) {
           row <- paste(row, " ", sep="")
        } else {
           row <- paste(row, "*", sep="")
        }
     }
     if (grepl('*', row, fixed = TRUE)) {
        row <- paste(row, "\n", sep="")
        cat(row)
     }
  }
}

args <- commandArgs(trailingOnly=TRUE)
x_steps <- as.numeric(args[1])
y_steps <- as.numeric(args[2])
iterations <- as.numeric(args[3])

mandelbrot(x_steps, y_steps, iterations)
```

Den är nästan identisk med den kompakta lösningen för Python. Den enda större skillnaden är att R inte har någon motsvarighet till funktionen `meshgrid` i `numpy`, så vi har fått använda `rep`. (Om vi av någon anledning verkligen velat använda `meshgrid` så hade vi kunnat importera den från R-biblioteket `pracma`.)

## 3.3 PHP

PHP är egentligen inte till för att man ska göra komplicerade beräkningar i det. Det är möjligt att det finns bibliotek som skulle snygga till koden, men jag nöjer mig med den här osnygga lösningen:

```
<?php

function mandelbrot($x_steps, $y_steps, $iterations) {

   $min_x = -2.0;
   $max_x = 0.5;
   $span_x = $max_x - $min_x;
   $step_x = $span_x / ($x_steps - 1);

   $min_y = -1.2;
   $max_y = 1.2;
   $span_y = $max_y - $min_y;
   $step_y = $span_y / ($y_steps - 1);

   $complex_plane = array();
   for ($x_pos = 0; $x_pos < $x_steps; $x_pos++) {
      $re = $min_x + $x_pos * $step_x;
      $column = array();
      for ($y_pos = 0; $y_pos < $y_steps; $y_pos++) {
         $im = $min_y + $y_pos * $step_y;
         $column[] = array($re, $im, $re, $im, False);
      }
      $complex_plane[] = $column;
   }

   for ($x_pos = 0; $x_pos < $x_steps; $x_pos++) {
      for ($y_pos = 0; $y_pos < $y_steps; $y_pos++) {
         for ($i = 0; $i < $iterations; $i++) {
            $diverged = $complex_plane[$x_pos][$y_pos][4];

            if (!$diverged) {
               $re = $complex_plane[$x_pos][$y_pos][0];
               $im = $complex_plane[$x_pos][$y_pos][1];

               $zi_re = $complex_plane[$x_pos][$y_pos][2];
               $zi_im = $complex_plane[$x_pos][$y_pos][3];

               $zi1_re = $zi_re**2 - $zi_im**2 + $re;
               $zi1_im = 2 * $zi_re * $zi_im + $im;

               $diverged = $zi1_re**2 + $zi1_im**2 > 4;

               $complex_plane[$x_pos][$y_pos][2] = $zi1_re;
               $complex_plane[$x_pos][$y_pos][3] = $zi1_im;
```

```
            $complex_plane[$x_pos][$y_pos][4] = $diverged;
          }
        }
      }
    }

    for ($y_pos = 0; $y_pos < $y_steps; $y_pos++) {
      $row = '';
      for ($x_pos = 0; $x_pos < $x_steps; $x_pos++) {
        if ($complex_plane[$x_pos][$y_pos][4]) {
          $row .= " ";
        } else {
          $row .= "*";
        }
      }
      if (strpos($row, '*') !== false) {
        $row .= "\n";
        echo($row);
      }
    }
}

$x_steps = $argv[1];
$y_steps = $argv[2];
$iterations = $argv[3];

mandelbrot($x_steps, $y_steps, $iterations);
?>
```

## 3.4 bash

Att implementera detta i bash lämnas som en övning åt läsaren. Det första steget blir att implementera flyttalsaritmetik, eftersom bash normalt bara befattar sig med heltal.