Programmering 1

Table of Contents

1. Verktyg

1.1. Variabler

1.1.1. Tilldelning

För att kunna använda värden (t.ex tal) i program måste man först lagra dem i en variabel. Att lagra ett värde i en variabel gör man med hjälp av tilldelning.

En tilldelning består av tre komponenter: en `variabel`, ett `tilldelningstecken` och ett `värde`.

```
hitpoints = 100 ①
name = 'Finn the Human' ②
```

① Utläses som "hitpoints *tilldelas* 100".

② Utläses som "name *tilldelas* Finn the Human".

1.2. Datatyper

Alla värden som lagras i en variabel är av en viss *datatyp*.

Datorn måste veta vilken typ av data som är lagrad i en variabel, så den vet vad den kan göra med variabeln.

Till exempel kan man addera eller dividera två tal med varandra, men det går inte att dividera ett ord.

1.3. Heltal (Integer)

Integer är en datatyp som representerar heltal. I Ruby kan heltal ha vilket heltalsvärde som helst från och med negativ oändlighet till och med positiv oändlighet.

I Ruby tilldelar man en variabel ett heltalsvärde genom att skriva heltalsvärdet till höger om tilldelningstecknet.

Tilldela en variabel en Integer

```
...
a_negative_number = -5 ①
a_small_number = 0 ②
a_large_number = 13 + 37 ③
...
```

① `a_negative_number` tilldelas *minus* 5 (heltal kan vara negativa).

② `a_small_number` tilldelas 0, som är ett heltal.

③ `a_large_number` tilldelas resultatet av operationen (i det här fallet 50, även det ett heltal)

1.3.1. Inkrementering

Att öka värdet på en variabel av datatypen `Integer` kallas *inkrementering*, eller att *inkrementera*.

Oftast används inkrementeringsverktyget i samband med loopar.

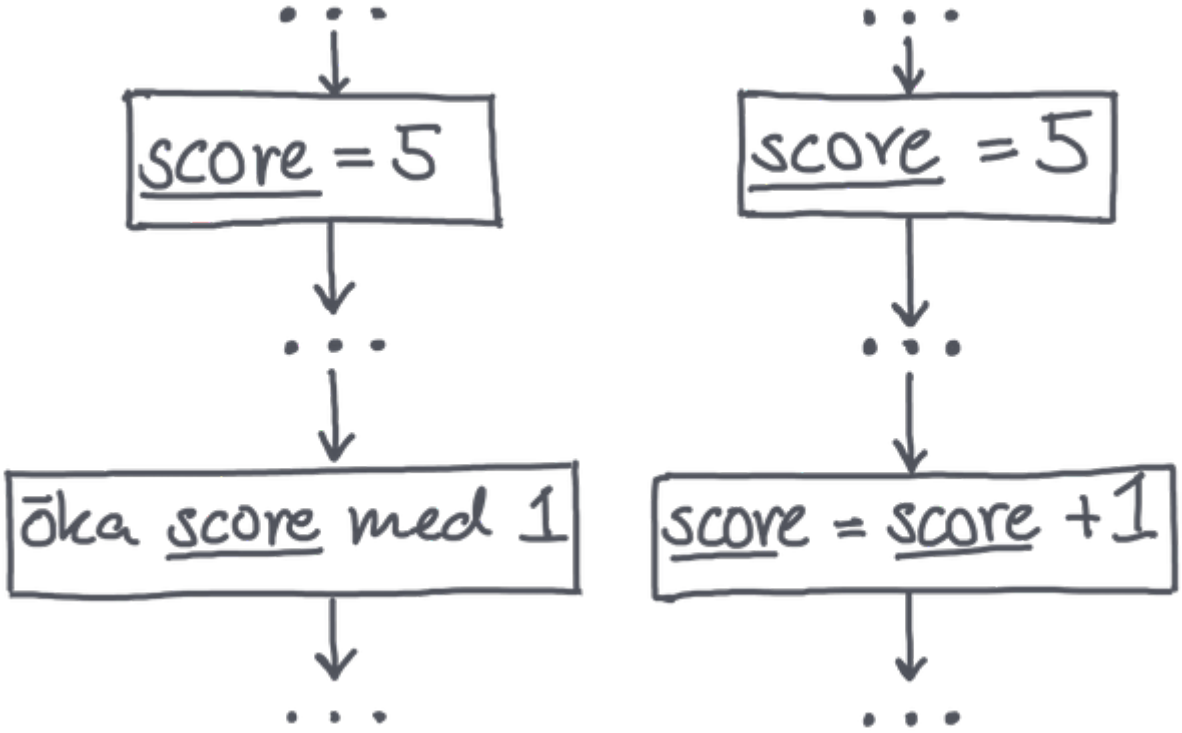

Figure 1. Två sätt att rita inkrementering i flödesschema

Inkrementering som ruby-kod

```
...
score = 5 ①
...
score = score + 1②
...
```

① För att kunna inkrementera en variabel måste den först ha tilldelats ett värde

② Inkrementering. Operationen till höger om tilldelningstecknet utförs först. Resultatet tilldelas variabeln till vänster om tilldelningstecknet.

1.3.2. Dekrementering

Att minska värdet på en variabel av datatypen `Integer` kallas *dekrementering*, eller att *dekrementera*.

Oftast används dekrementeringsverktyget i samband med loopar.

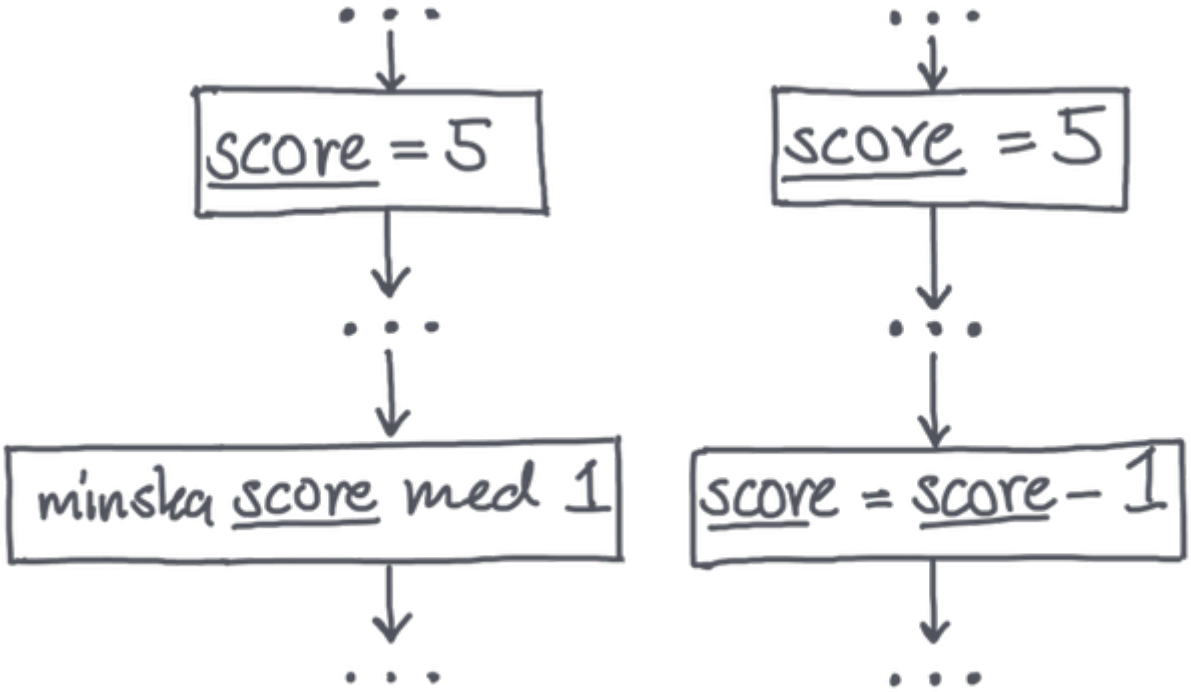

Figure 2. Två sätt att rita dekrementering i flödesschema

Dekrementering som ruby-kod

```
...
score = 5 ①
...
score = score - 1②
...
```

① För att kunna dekrementera en variabel måste den först ha tilldelats ett värde

② Dekrementering. Operationen till höger om tilldelningstecknet utförs först. Resultatet tilldelas variabeln till vänster om tilldelningstecknet.

1.4. Sträng (String)

String är datatyp som representerar en ordnad sekvens av tecken (t.ex tecken, ord, meningar, eller hela böcker).

För att skapa en sträng i Ruby behöver man omge värdet man ska tilldela variabeln med " (uttalas *dubbelfnutt*).

Tilldela en variabel en String

```
...
first_string  = "1337" ①
second_string = "An escalator can never break: it can only become stairs" ②
third_string  = " " ③
fourth_string = "" ④
...
```

① Det kan se ut som strängen innehåller ett tal, men den innehåller i själva verket 4 tecken (tecknet ett, tecknet tre, tecknet tre och tecknet sju).

② Strängar kan innehålla alla möjliga tecken och vara hur långa som helst.

③ Den här strängen innehåller bara ett tecken; tecknet *mellanslag*

④ Den här strängen innehåller inga tecken alls; det är en *tom sträng*.

1.4.1. Indexering

Alla tecken i en sträng har en position, eller `index`. Det första tecknet har index 0, och det sista tecknets index är antalet tecken i strängen minus 1:

Strängen "kittens" har 7 tecken, och det största indexet är därmed 6.

Man kan använda indexeringsverktyget för att ta reda på vilket tecken som finns på ett visst index i en sträng.

Indexeringsverktyget i ruby-kod

```
...
first_string  = "1337"
second_string = "kittens"
third_string  = " " ③
fourth_string = "" ④

first_string[3]  #=> "7" ①
second_string[4] #=> "e"
third_string[1]  #=> nil ②
fourth_string[0] #=> nil ③
...
```

① Observera att det är en `String`, inte en `Integer` som returneras.

② Längden på strängen är 1. Alltså är sista (och första) index 0. Det finns *inget* (`nil`) på index 1.

③ Det finns *inget* i en tom sträng. Inte ens på index 0.

Man kan fråga en sträng hur många tecken den innehåller genom att anropa metoden `length` eller `size`, som returnerar strängens längd.

Metoderna length och size returnerar längden på strängen

```
...
first_string  = "1337"
second_string  = " "
third_string = ""
...
first_string.length #=> 4
first_string.size #=> 4 ①
second_string.length #=> 1
third_string.length #=> 0
...
```

① `length` och `size` gör samma sak

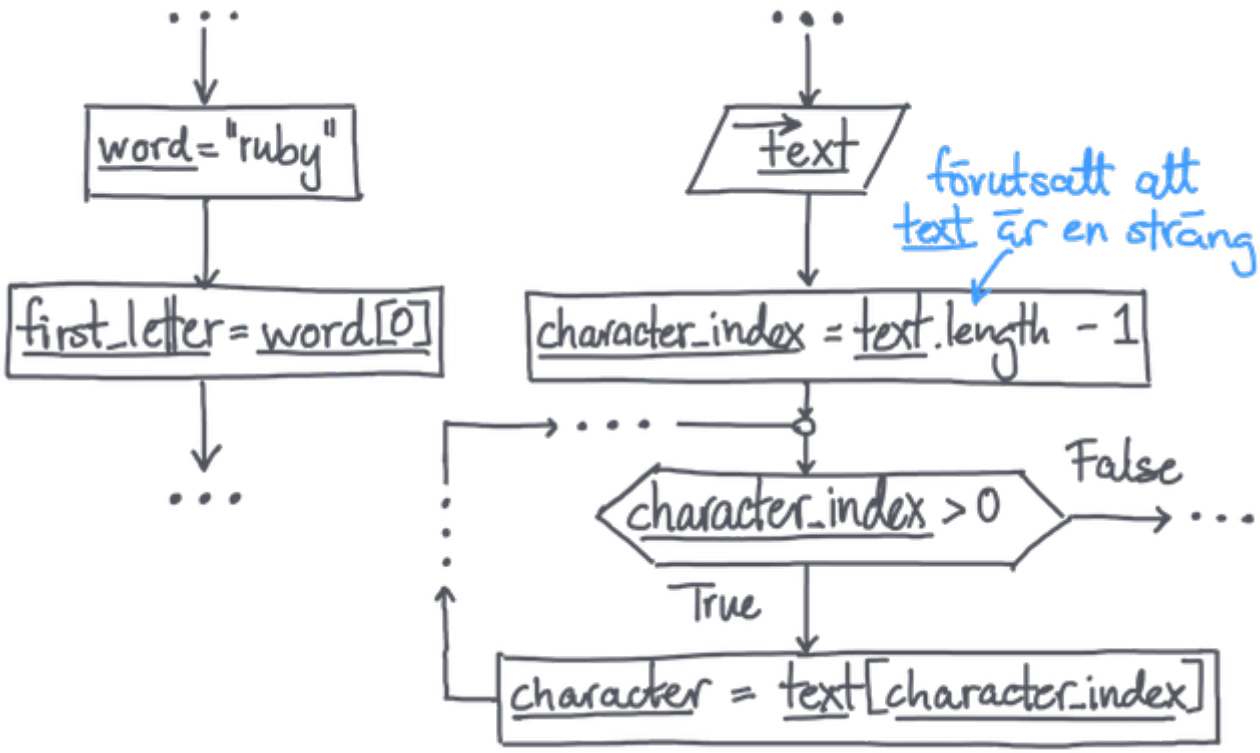

Figure 3. Två exempel på hur indexeringsverktyget kan användas

1.4.2. Konkatenering

Att slå ihop, eller kombinera, två strängar kallas för *konkatenering*.

Konkatenering kan t.ex användas för att skapa nya strängar, kombinera två strängar till en sträng, eller för att lägga till ett teecken i slutet av en sträng.

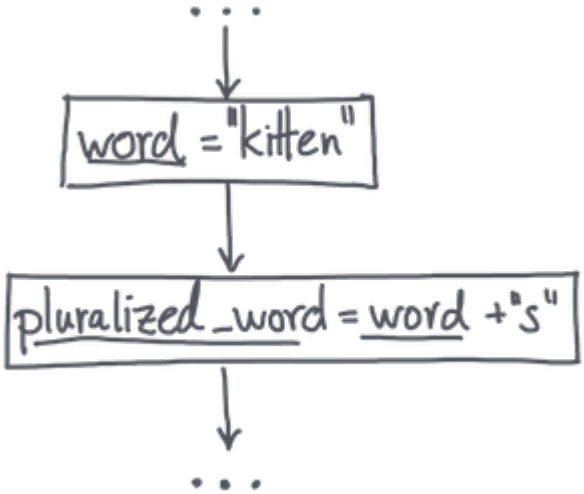

Figure 4. Konkatenering i flödesschema

Konkatenering i ruby-kod

```
...
first_word = "banana"
second_word = "pie"
new_word = first_word + " " ①
new_word = new_word + second_word + "!" ②
```

① Om man vill ha ett mellanrum mellan två konkatenerade strängar måste man lägga in det manuellt.

② Först konkateneras `new_word` med `second_word`. Resultatet av konkateneringen (`"banana pie"`) konkateneras sen med `"!"`.

1.5. Uppdelning av programflödet

Villkor delar upp programflödet i två möjliga vägar.

Ett villkor har **en** ingång, och **två** utgångar - den ena utgången är märkt `sann` eller `true` och den andra är märkt `falskt` eller `false`.

Villkor ritas som hexagoner. I hexagonen står en `jämförelseoperation`.

En jämförelseoperation består av tre delar: en `operand`, en `jämförelseoperator` och sen en till `operand`.

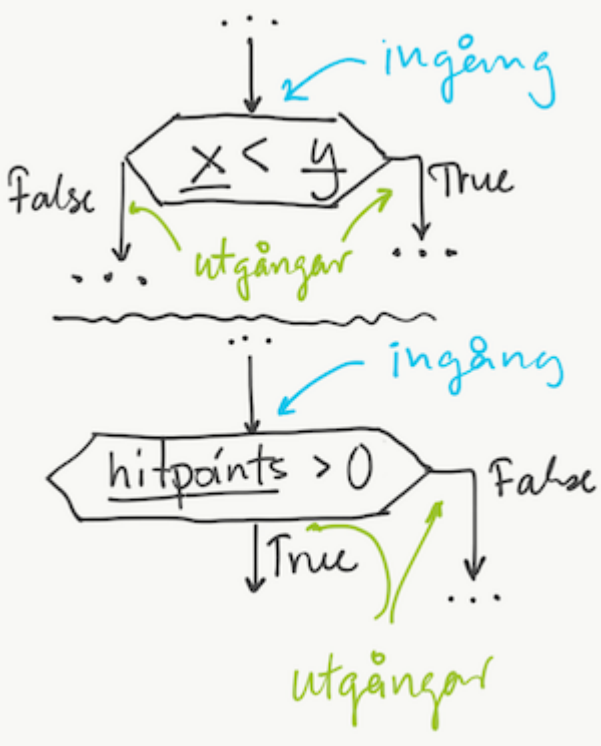

Figure 5. Två villkor i flödesschema

Alla tillåtna jämförelseoperationer

```
number1 = 10
number2 = 15

number1 > number2 ①
number1 < number2 ②
number1 == number2 ③
number1 != number2 ④
number1 >= number2 ⑤
number1 <= number2 ⑥
```

① ⇒ `false` - Utläses som "number1 är **större än** number2".

② ⇒ `true` - Utläses som "number1 är **mindre än** number2".

③ ⇒ `false` - Utläses som "number1 är **samma som** number2".

④ ⇒ `true` - Utläses som "number1 är ***inte* samma som** number2".

⑤ ⇒ `false` - Utläses som "number1 är **större än *eller* samma som** number2".

⑥ ⇒ `true` - Utläses som "number1 är **mindre än *eller* samma som** number2".

Beroende på om jämförelseoperationen utvärderas till `sant` eller `falskt` fortsätter programflödet genom respektive utgång.

1.5.1. If

Om du vill att ditt program ska göra en sak **enbart om ett villkor är sant**, kan du använda `if`. If-verktyget använder ett villkor, och om villkoret utvärderas till `sant` kommer något ske. Om villkoret **inte** utvärderas till sant kommer programflödet fortsätta efter villkoret, som om ingenting hänt.

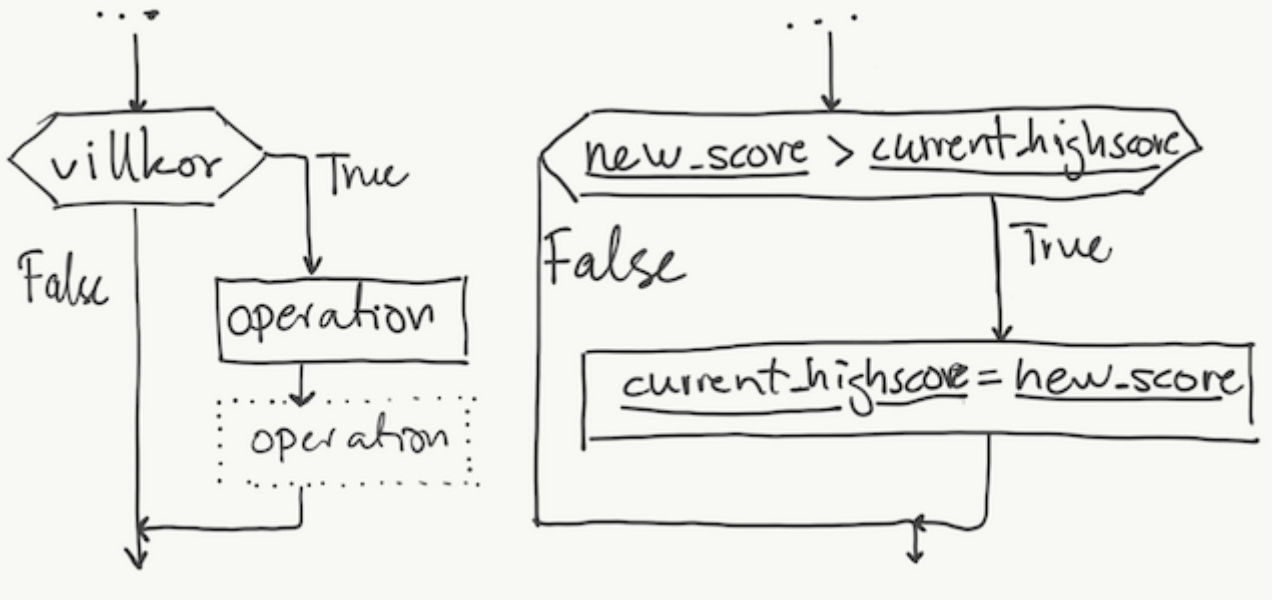

Figure 6. If-verktyget i ett flödesschema

√ sdfdssas

```
...
if new_score > current_highscore
    current_highscore = new_score ①
end
②
...
```

① Raderna mellan jämförelseoperationen och `end` motsvarar `true`-utgången i flödesschemat

② Efter `end` fortsätter programflödet. Om villkoret är falskt hoppar programflödet direkt hit istället för att gå in i `if-satsen`.

I exemplet ovan kommer `current_highscore` tilldelas `new_score` **enbart** om `new_score` är större än `current_highscore`

If-verktyget fungerar **enbart** på `true`-utången.

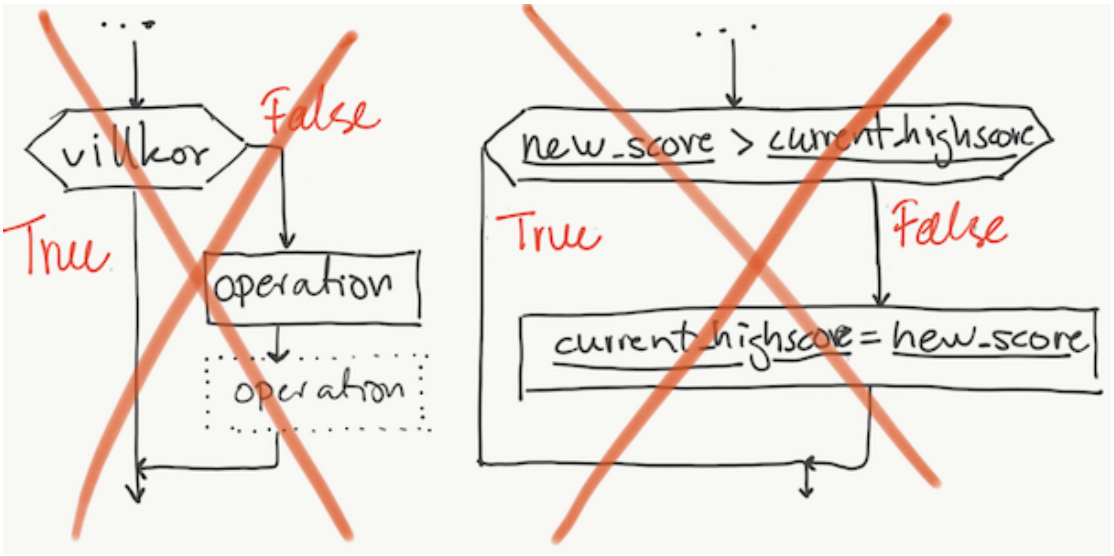

Figure 7. Felaktigt if-verktyg (operationen utförs vid `false`).

Ett felaktigt if-verktyg i ruby.

```
...
if new_score <= current_highscore
    #do nothing
end
current_highscore = new_score ①
...
```

① Eftersom programflödet kommer hit **oavsett** om villkoret utvärderats till `true` eller `false` kommer denna rad **alltid** att köras.

1.5.2. If-else

Om du vill att ditt program ska göra en sak **om ett villkor är sant**, ***och en annan sak om villkoret är falskt*** kan du använda `if-else`.

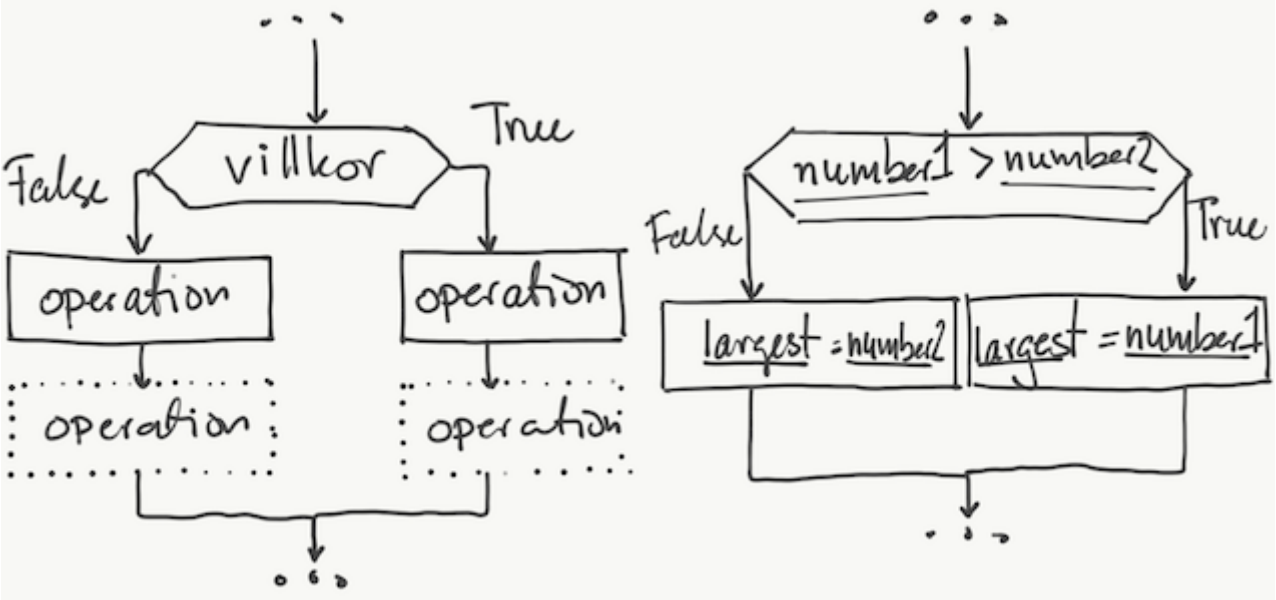

Figure 8. If-else-verktyget i ett flödesschema

If-else börjar som en `if`, men om villkoret **inte** utvärderas till sant kommer programflödet hoppa till koden som ligger i `else-blocket`.

If-else-verktyget som ruby-kod

```
...
if number1 > number2
    largest = number1 ①
else
    largest = number2 ②
end
... ③
```

① Koden mellan villkoret och `else` (`if`-blocket) körs **enbart** om villkoret är sant.

② Koden mellan `else` och `end` (`else`-blocket) körs **enbart** om villkoret är falskt.

③ Oavsett om `if` eller `else`-blocket körts kommer programflödet fortsätta efter `end`.

1.5.3. If-else-if-else

Om ditt program har *fler än två* olika saker det ska göra baserat på **olika** villkor kan du använda if-else-if-else.

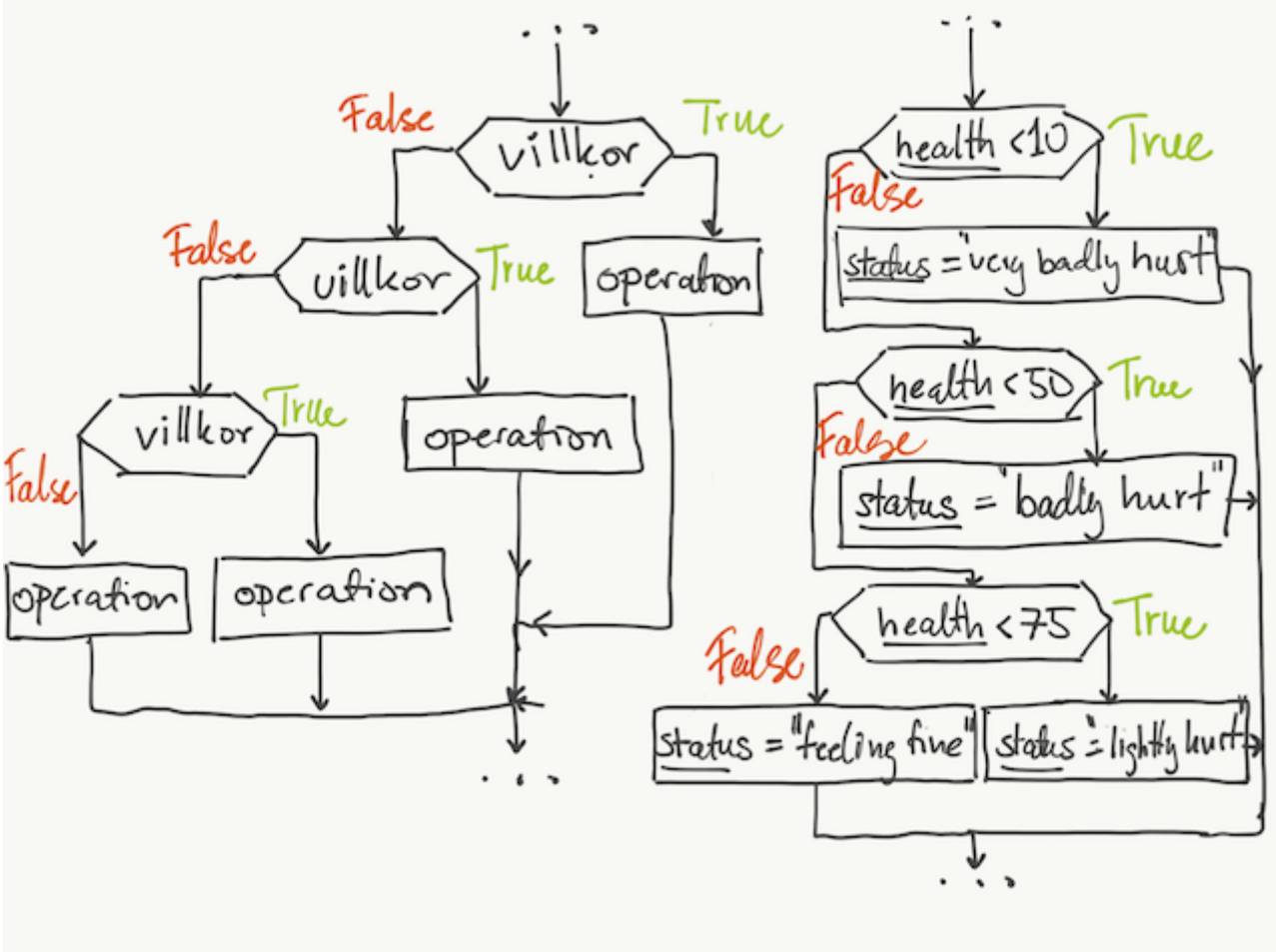

Figure 9. If-else-if-else-verktyget i ett flödesschema

If-else-if-else börjar som en `if`, men om villkoret **inte** utvärderas till sant kommer programflödet hoppa till och utvärdera **nästa** `elsif-block`.

```
...
if health < 10 ①
    status = "very badly hurt" ②
elsif health < 50 ③
    status = "badly hurt" ④
elsif health < 75 ⑤
    status = "lightly hurt" ⑥
else ⑦
    status = "feeling fine" ⑧
end
... ⑨
```

① Om villkoret är falskt fortsätter programflödet direkt till nästa `elsif`.

② Detta kodblock körs **enbart** om `health` är mindre än 10. Därefter hoppar programflödet **direkt** till raden efter `end`.

③ Om villkoret är falskt fortsätter programflödet direkt till nästa `elsif`.

④ Detta kodblock körs **enbart** om `health` är mindre än 50. Därefter hoppar programflödet **direkt** till raden efter `end`.

⑤ Om villkoret är falskt fortsätter programflödet direkt till `else`.

⑥ Detta kodblock körs **enbart** om `health` är mindre än 75. Därefter hoppar programflödet **direkt** till raden efter `end`.

⑦ Om **inget** av `elsif-satserna` utvärderats till true kommer programflödet hoppa till `else`.

⑧ Detta kodblock körs **enbart** om **inget** av de tidigare villkoren utvärderats till sant.

⑨ Oavsett vilket av kodblocken som körts kommer programflödet fortsätta här.

1.5.4. Nästade eller kedjade if

Om ditt program har **flera olika villkor** som måste vara uppfyllda för att något ska hända kan du använda verktyget `kedjad if`.

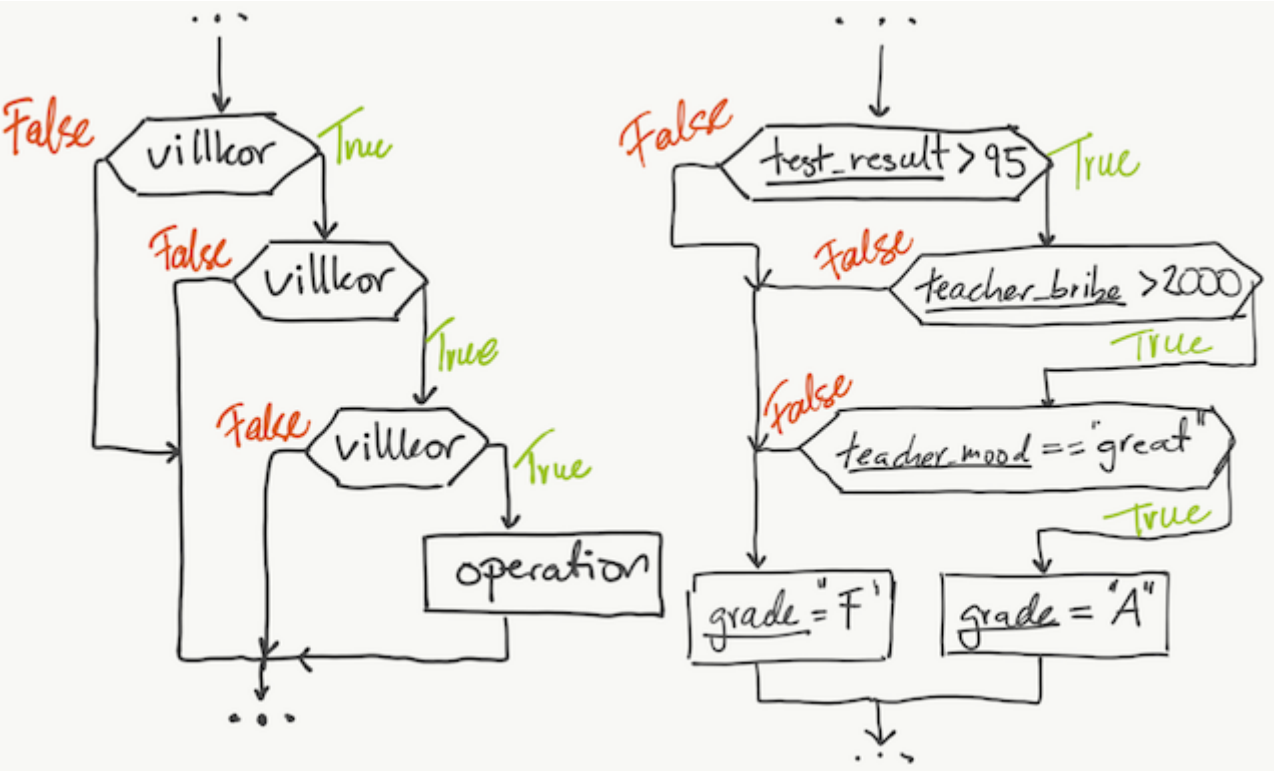

Figure 10. Kedjade-if i flödesschema

Kedjade if som ruby-kod

```
...
if test_result > 95
    if teacher_bribe > 2000
        if teacher_mood == "great"
            grade = "A"
        else
            grade = "F"
        end
    else
        grade = "F"
    end
else
    grade = "F"
end
...
```

1.6. Upprepning av programflödet

Loopar leder programflödet tillbaks tills samma ställe om och om igen, tills ett villkor har blivit uppfyllt.

Loopar återanvänder symbolen för villkor, men på strecket som leder till ingången ritar vi en liten cirkel där loopen återansluter till loop-villkoret.

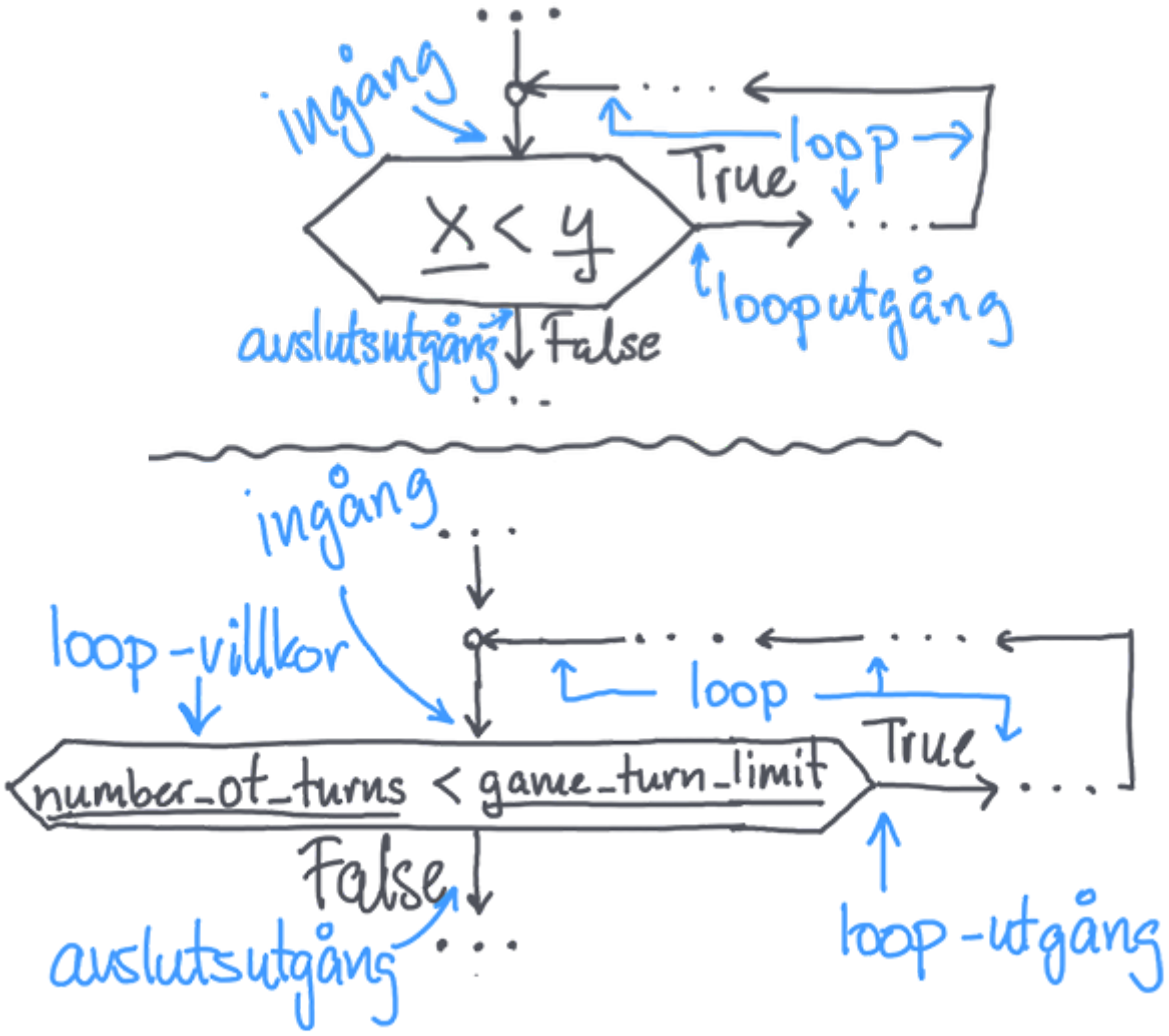

Figure 11. Två loopar

Loop-utgången **måste** ske på `true`-utången.

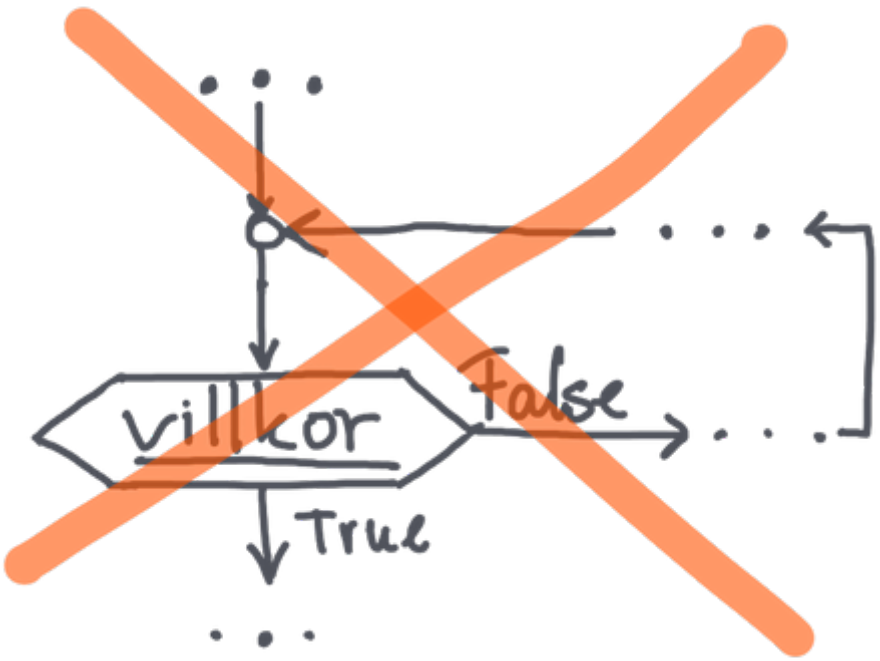

Figure 12. Felaktigt loop-verktyg (loop sker vid `false`).

1.6.1. Inkrementerande loop

En inkrementerande loop passar bra att använda om du i förväg vet (eller kan räkna ut) hur många gånger programflödet behöver upprepas, t.ex. om programmet alltid ska göra något 5 gånger, eller om vet att du ska göra något lika många gånger som du har tal i en lista.

Inkrementerande loopar använder sig av en *räknar-variabel*. En räknar-variabel håller koll på hur många gånger loopen upprepats. Vanliga namn på räknar-variabler är `counter`, `iterations` eller `i`, men en räknar-variabel kan heta precis vad som helst.

För att en räknande loop ska fungera behöver följande steg ske:

1. Innan loopen påbörjas måste man *initiera* räknar-variabeln, det vill säga, tilldela den ett värde. I de flesta fall initierar man räknar-variabeln med värdet 0 (eftersom det är så många gånger man upprepat programflödet innan man börjar loopa).
2. Räknar-variabeln används sen inne i villkoret som avgör om programflödet ska upprepas eller ej.
3. Inne i loopen måste räknar-variabeln *inkrementeras*, det vill säga ökas. Om man inte ökar räknar-variabeln kommer loopen fortsätta i all evighet, eftersom loop-villkoret aldrig blir falskt.

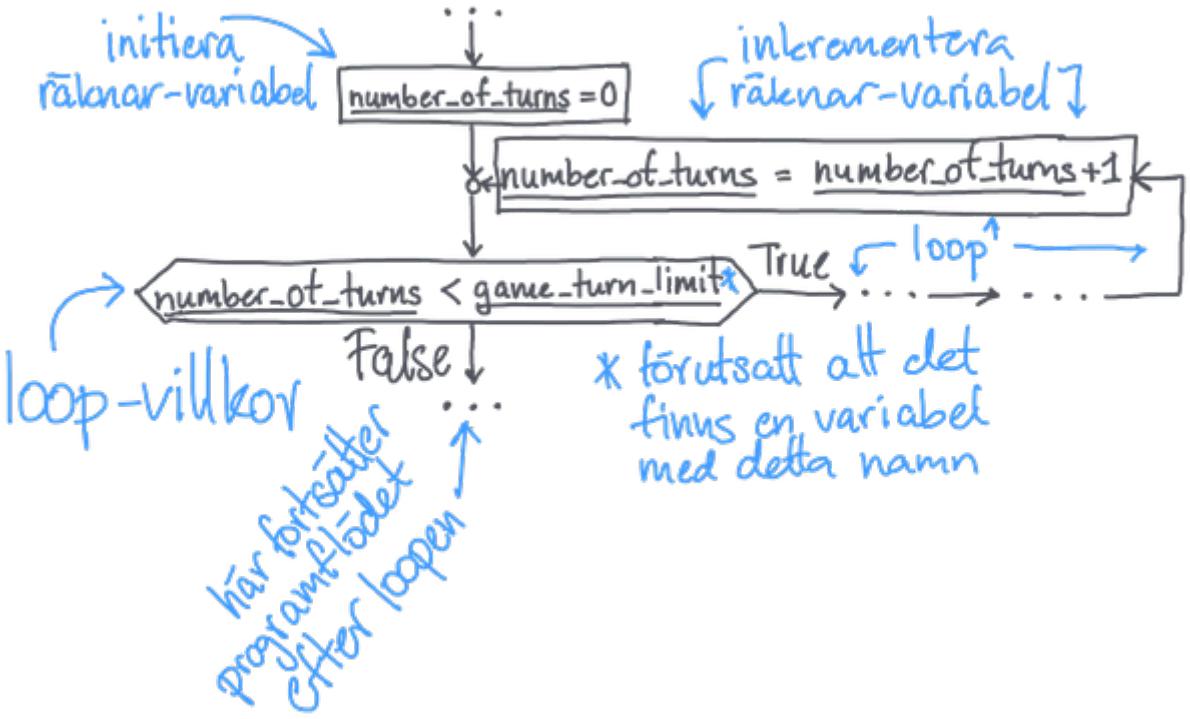

Figure 13. En inkrementerande loop i flödesschema

Inkrementerande loop som ruby-kod

```
...
number_of_turns = 0 ①
while number_of_turns < game_turn_limit ②
    ... ③
    number_of_turns = number_of_turns   + 1 ④
end
... ⑤
```

① Initierar räknar-variabeln.

② Så länge räknar-variabeln är mindre än `number_of_iterations` kommer loopen upprepas

③ Koden som ska upprepas.

④ Inkrementera räknarvariabeln - om man missar detta steg får man en oändlig loop.

⑤ Här fortsätter programflödet när/om loop-villkoret är falskt.

1.6.2. Dekrementerande loop

En dekrementerande loop har i stort sett samma användningsområde som en inkrementerande loop, det vill säga, du vet (eller kan räkna ut) hur många gånger du behöver upprepa programflödet.

Skillnaden är att räknar-variabeln räknar neråt, det vill säga dekrementerar.

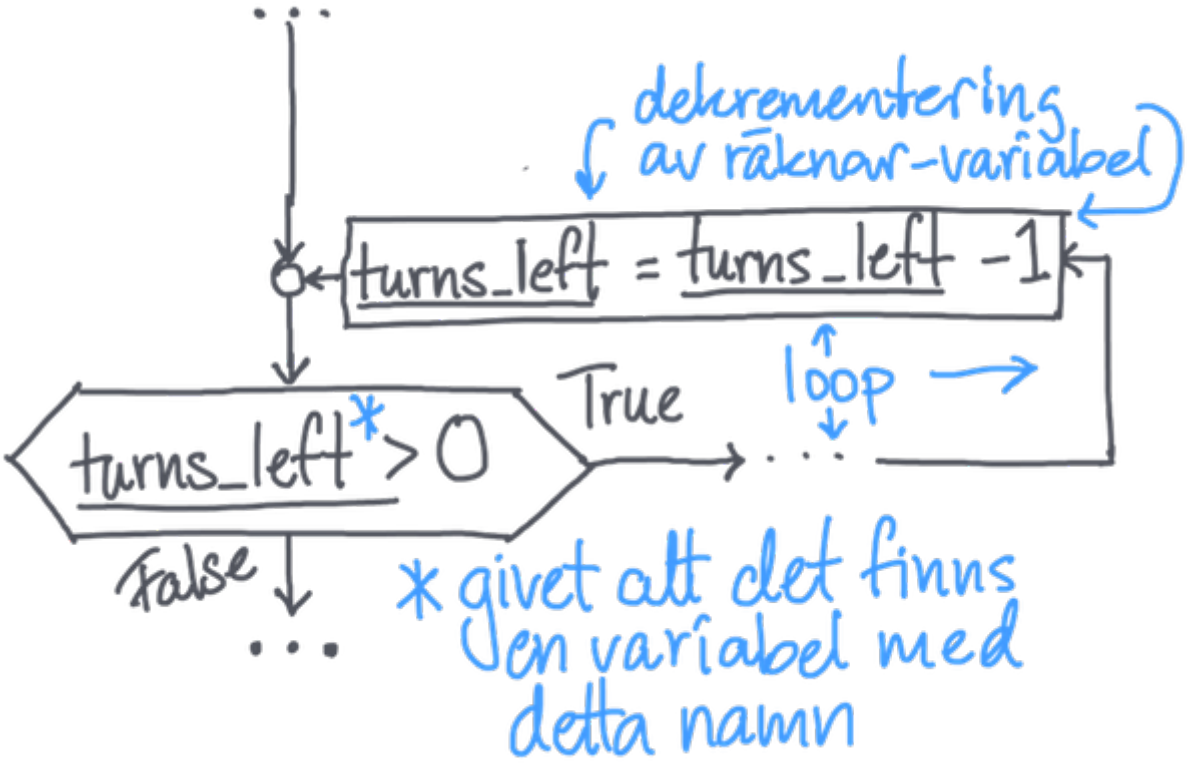

Figure 14. Dekrementerande loop i flödesschema

Dekrementerande loop som ruby-kod

```
...
turns_left = 6 ①
...
while turns_left > 0 ②
    ... ③
    turns_left = turns_left -1 ④
end
... ⑤
```

① Initierar räknar-variabeln.

② Så länge `turns_left` är större än 0 kommer loopen upprepas.

③ Koden som ska upprepas.

④ Dekrementera räknarvariabeln - om man missar detta steg får man en oändlig loop.

⑤ Här fortsätter programflödet när/om loop-villkoret är falskt.

1.6.3. Loop med okänt antal iterationer

Om du inte i förväg vet (eller kan räkna ut) hur många gånger programflödet ska upprepas behöver du använda en loop med brytvärde.

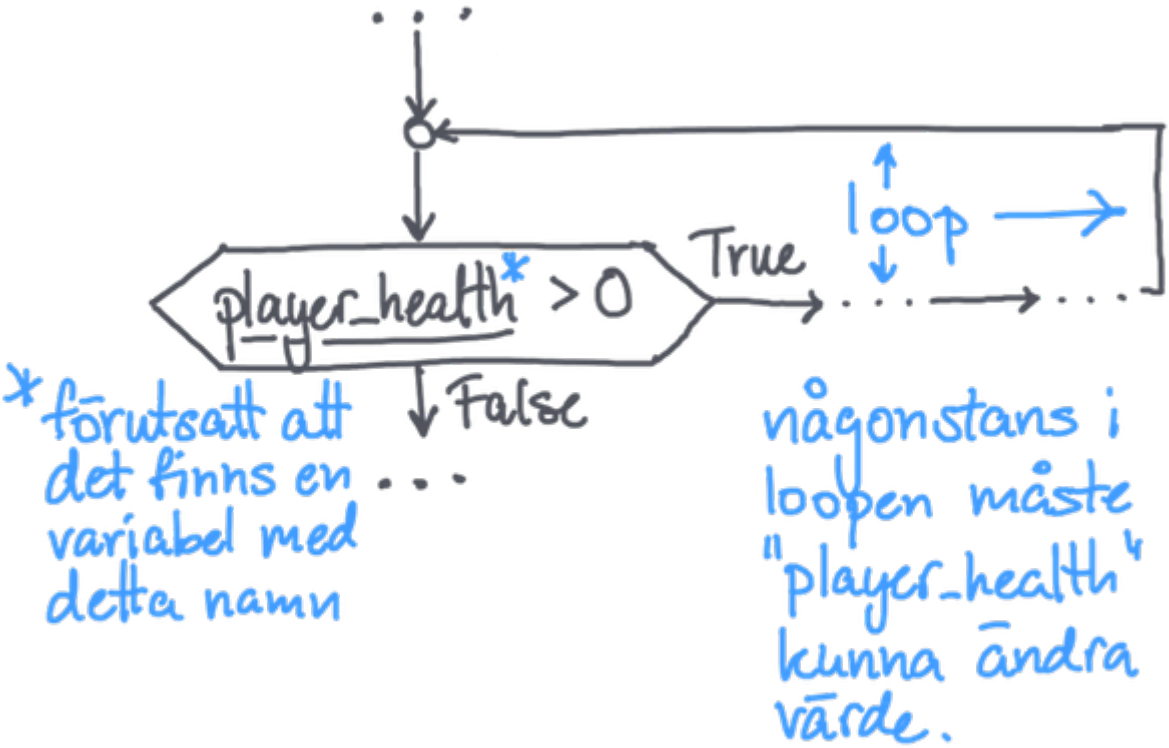

Figure 15. Loop med okänt antal iterationer

Loop med okänt antal iterationer

```
...
while player_health > 0 ①
    ... ②
    ... ③
end
... ④
```

① Så länge som `player_health` är större än 0 kommer loopen upprepas.

② Koden som ska upprepas.

③ Någonstans i loopen måste `player_health` kunna ändras - annars blir det en oändlig loop.

④ Här fortsätter programflödet när/om loop-villkoret är falskt.

1.7. Funktioner

1.7.1. Funktionsmaskinen

Överallt inom programmering finns funktioner. Dessa är som maskiner där man stoppar in saker och får ut ett resultat. Det man kan stoppa in i och få ut från maskiner är data av olika datatyper, maskinen behandlar sedan datan på något vis och ger ett resultat med någon data och datatyp.
Programmerare skapar dessa maskiner och använder dem för att lösa problem. Alla program som du använder består av många funktioner som arbetar tillsammans. Det är även ett smidigt sätt att gruppera sin kod och undvika s.k. spagettikod.

Exempel

- Double

Figure 16. En maskin som dubblar talet man stoppar in

- Starts with

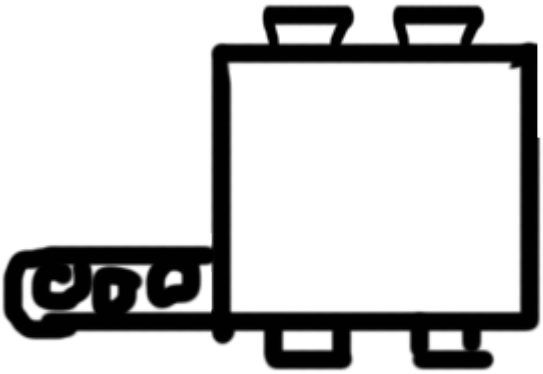

Figure 17. En maskin som avgör man det första tecknet på det första du stoppar in är samma som det andra du stoppar in.

1.7.2. Funktionsdefinition och Parametrar

Varje funktion måste definieras med namn och input. Inputten läggs i variabler som kan användas inne i funktionen. Dessa variabler kallas för parametrar. Följande exempel visar en funktionsdefinition med en parameter.

```
...
def function_name(input1)① ② ③
    #Inner workings of function④
end⑤
...
```

① Varje funktionsdefinition börjar med nyckelordet `def`.

② `function_name` är namnet på funktionen och kan vara nästan vad som helst.

③ Efter namnet på funktionen följer parenteser och namnet på parametern.

④ Koden för funktionen skrivs här och ska indenteras.

⑤ `end` visar var koden till funktionen slutar.

Figure 18. Resultatet blir en maskin som den här

1.7.3. Flera inputs

Om en funktion har flera parametrar så avskiljer man varje parameter med ett kommatecken.

Funktion med två inputs

```
...
def function_name(input1, input2)①
    #Inner workings of function
end
...
```

① Varje parameter avskiljs med ett kommatecken

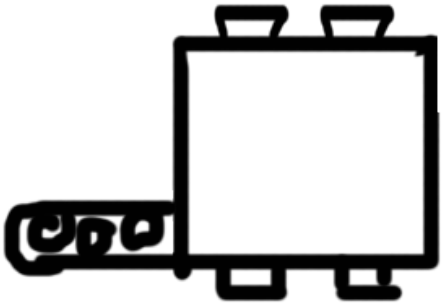

Figure 19. Resultatet blir en maskin som den här

Funktion med tre inputs

```
...
def function_name(input1, input2,
input3)
    #Inner workings of function
end
...
```

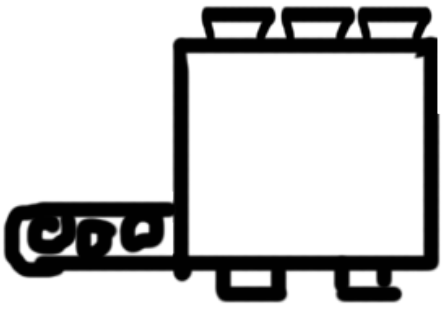

Figure 20. Resultatet blir en maskin som den här

1.7.4. Funktioner utan input

En funktion kan även definieras utan parametrar. Då lämnar man parentesen tom.

```
...
def function_name()
    #Inner workings of function
end
...
```

Figure 21. Resultatet blir en maskin som den här

1.7.5. Funktionsanrop och inputdata

Att stoppa in data i en maskin och starta den är ett funktionsanrop. All data som man stoppar in i funktioner måste komma någonstans ifrån. Den data man stoppar in i funktionen kallas för funktionens argument. När man väl har data att stoppa in i funktionen så kan man anropa den. Detta gör man på följande sätt.

```
...
function_name(argument)
...
```

Datan måste dock komma någonstans ifrån, och det finns olika sätt att få data till sina funktioner.

Hårdkodad data

Det vanligaste som du kommer göra är att skapa din egna data direkt i programmet. Detta kan du göra antingen genom att spara datan i variabler och skicka med i funktionsanropet.

```
...
argument = "Some string value"
function_name(argument)
...
```

eller skriva värdet direkt i funktionsanropet

```
...
function_name("Some string value")
...
```

Data från Tangentbordet

Man kan även låta den som kör programmet mata in data från tangentbordet. All data som matas in från tangentbordet är strängar och har ett "\n"-tecken i slutet som representerar ny rad. Så om man vill att användaren ska mata in siffror måste man omvandla datatypen till det man är ute efter.

```
...
argument = gets()
function_name(argument)
...
```

Man bör endast använda tangentbordet för inmatning av data om det är syftet med programmet. t.ex ett spel eller något som kräver att användaren gör något val.

Data från Filer

Man kan även läsa in data från filer och på samma sätt som från tangentbordet är all data som läses in strängar.

Läs in hela innehållet i en fil som en sträng

```
...
file_content = File.read("path/to/file")① ②
...
```

① "path/to/file" är en relativ sökväg till filen

② Om filen innehåller radbrytningar kommer dessa vara ett "\n"-tecken i strängen. ==== Läs in varje rad i en fil som en array av strängar

```
...
file_content = File.readlines("path/to/file")
...
```

1.7.6. Output

En funktion kan endast ge en output och man ska göra något med den behöver man göra något av följande när man anroper den.

Spara resultatet i en variabel

```
...
result = function_name(argument)
...
```

Anropa funktionen direkt där du behöver den

I ett villkor

```
...
if function_name(argument) == "I am a tea cup"
    #Do stuff
end
...
```

Eller skriva ut resultatet i terminalen

```
...
puts function_name(argument)
...
```

1.7.7. Output - Return

När du vill ge output från en funktion så använder man `return` följt av den data du vill ska komma ut.

```
...
return output
...
```

`return` avbryter resten av funktionen och programmet fortsätter där du anropade funktionen.

1.7.8. Komplett exempel

- double

```
...
def double(num)
    return num * 2
end
...
```

- starts_with

```
...
def starts_with(string, character)
    first_character = string[0]
    return first_character == character
end
...
```

1.8. Sammansatta verktyg

1.8.1. Iterativt Uppbyggd Output

När man vill omvandla en sträng eller en array till en annan sträng eller array är det oftast enklast att bygga upp en **ny** sträng eller array snarare än att modifiera den ursprungliga. Det gör du enklast genom att använda verktyget "Iterativt Uppbyggd Ouptut"

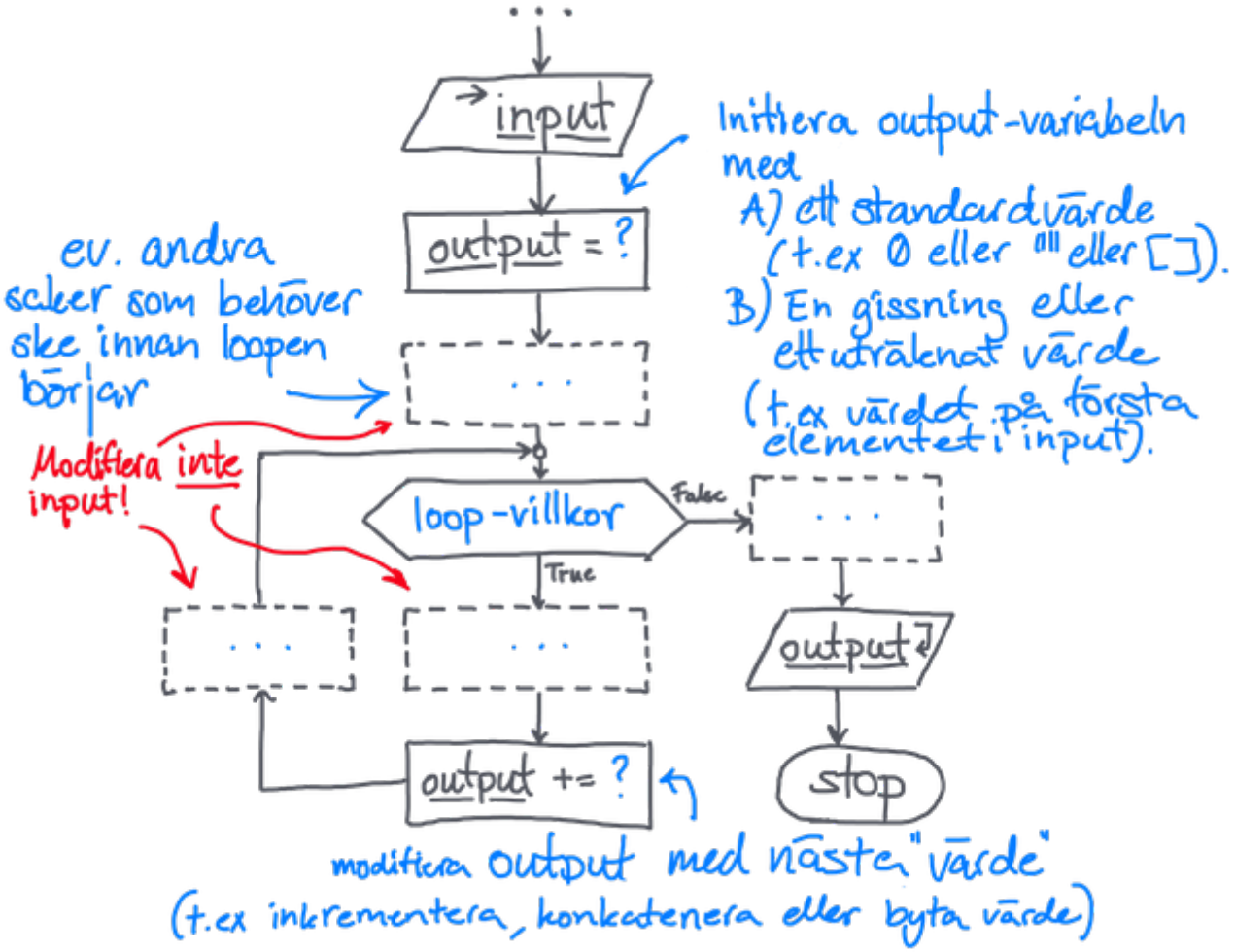

Figure 22. Iterativt Uppbyggd Output

1.9. Funktionsanrop

Ibland vill man i en del av en ny funktion använda samma funktionalitet som finns i en annan funktion.

I stället för att rita samma flöde i det nya flödesschemat, eller skriva (eller kopiera in) samma kod i den nya funktionen, kan man istället anropa den funktion man behöver, inne i den nya funktionen.

På så viss spar man tid och minskar risken för buggar (förutsatt att den funktion man anropar fungerar korrekt).

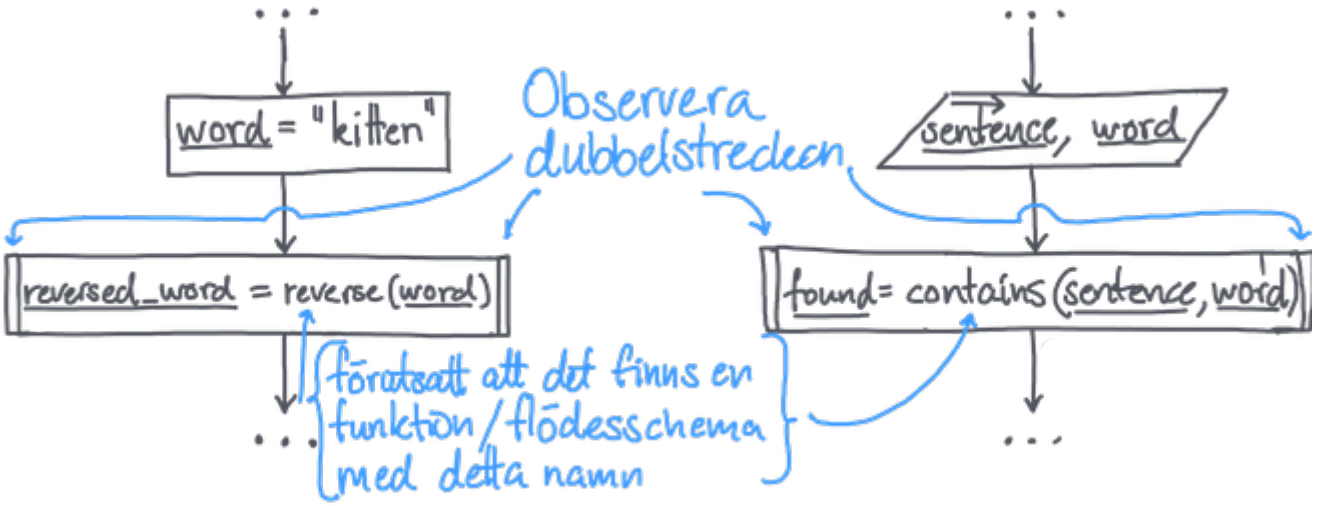

Figure 23. Funktionsanrop i flödesschema

Funktionsanrop i ruby-kod

```
...
def new_function(a_word)
    ...
    reversed_word = reverse(a_word) ①
    ...
end
...
```

① Förutsatt att det faktiskt finns en funktion som heter `reverse` och som tar en sträng som indata.

2. Övningar

2.1. Villkor

2.1.1. Smallest of two

Funktionen ska ta två integers som **input** och ge det minsta av dem som **output**.

Exempeltabell

Input		Output
10	2	2
-1	0	-1
1337	9001	1337

Exempelkörning

```
smallest_of_two(10,2) #-> 2
```

2.1.2. Largest of three

Funktionen ska ta tre integers som **input** och ge det största värdet av dem som **output**.

Exempeltabell

Input			Output
10	2	2	10
-2	-3	-1	-1
8	1337	0	1337

Exempelkörning

```
largest_of_three(10,2,2) #-> 10
```

2.1.3. Smallest of four

Funktionen ska ta fyra integers som **Input** och ge det minsta värdet av dem som **output**.

Exempeltabell

Input				Output
10	2	2	1	1
-2	-3	-1		10
-3	8999	1337	9001	1338

Exempelkörning

```
smallest_of_four(10,2,2,1) #-> 1
```

2.1.4. Ticket Price

Funktionen ska ta en integer **age** som **Input** och ska beroende på input ge olika output som följer tabellen nedan.

Ålder	Pris
Under 18	10
Mellan 18 och 64	20

Över 64	15

Exempeltabell

Input	Output
10	10
40	20
70	15

Exempelkörning

```
ticket_price(10) #-> 10
ticket_price(30) #-> 20
ticket_price(70) #-> 15
```

2.2. Loopar

2.2.1. Sum

Exempeltabell

| **Input** | **Output**

Exempelkörning

```

```

2.2.2. Factorial

Exempeltabell

| **Input** | **Output**

Exempelkörning

```

```

2.2.3. Collatz

Exempeltabell

| **Input** | **Output**

Exempelkörning

```
```

2.2.4. Contains

Exempeltabell

| Input | Output

Exempelkörning

```
```

2.2.5. Count

Exempeltabell

| Input | Output

Exempelkörning

```
```

2.2.6. Reverse

Exempeltabell

| Input | Output

Exempelkörning

```
```

2.2.7. Palindrom

Exempeltabell

| Input | Output

Exempelkörning

```
```

2.2.8. Rövarspråket

Exempeltabell

| **Input** | **Output**

Exempelkörning

```
```

3. Biblioteket

3.1. Funktioner och operatorer

3.1.1. Next Number

Beskrivning

Tar ett heltal som input och ger nästa tal som output.

Testdata

Input	Output
5	6
2	3
-3	-2

Testkörning

```
next_number(5) #=> 6
next_number(2) #=> 3
next_number(-3) #=> -2
```

3.1.2. Previous Number

Beskrivning

Tar ett heltal som input och ger föregående tal som output.

Testdata

Input	Output
5	4
2	1
-3	-4

Testkörning

```
previous_number(5) #=> 4
previous_number(2) #=> 1
previous_number(-3) #=> -4
```

3.1.3. Square

Beskrivning

Tar ett heltal som input och ger kvadraten på talet som output. Kvadraten på ett tal innebär talet multiplicerat med sig själv.

Testdata

Input	Output
5	25
2	4
-3	9

Testkörning

```
square(5) #=> 25
square(2) #=> 4
square(-3) #=> 9
```

3.1.4. Cube

Beskrivning

Tar ett heltal som input och ger kubiken på talet som output. Kubiken på ett tal innebär talet multiplicerat med sig själv tre gånger.

Testdata

Input	Output
5	125
2	8
-3	-27

Testkörning

```
cube(5) #=> 125
cube(2) #=> 8
cube(-3) #=> -27
```

3.2. Villkor

3.2.1. Is Negative

Beskrivning

Tar ett heltal som input och avgör om talet är negativt.

Testdata

Input	Output
5	false
2	false
-3	true

Testkörning

```
is_negative(5) #=> false
is_negative(2) #=> false
is_negative(-3) #=> true
```

3.2.2. Is Even

Beskrivning

Tar ett heltal som input och avgör om talet är jämnt.

Testdata

Input	Output
5	false
2	true
-10	true

Testkörning

```
is_even(5) #=> false
is_even(2) #=> true
is_even(-10) #=> true
```

3.2.3. Is Odd

Beskrivning

Tar ett heltal som input och avgör om talet är udda.

Testdata

Input	Output
5	true
2	false
-10	false

Testkörning

```
is_odd(5) #=> true
is_odd(2) #=> false
is_odd(-10) #=> false
```

3.2.4. Absolute

Beskrivning

Tar ett heltal som input och ger absolutvärdet på talet som output. Absolutvärdet innebär det positiva motsvarigheten av talet.

Testdata

Input	Output
5	5
100	100
-200	200
-10	10

Testkörning

```
absolute(5) #=> 5
absolute(100) #=> 100
absolute(-200) #=> 200
absolute(-10) #=> 10
```

3.2.5. Between

Beskrivning

Tar tre tal som input och avgör om det första ligger mellan den andra och tredje.

Testdata

Input 1	Input 2	Input 3	Output
1	1	10	true
1	5	100	false
0	-1	1	true

Testkörning

```
between(1, 1, 10) #=> true
between(1, 5, 100) #=> false
between(0, -1, 1) #=> true
```

3.2.6. Between Strict

Beskrivning

Tar tre tal som input och avgör om det första ligger strikt mellan den andra och tredje.

Testdata

Input 1	Input 2	Input 3	Output
1	1	10	false
1	5	100	false
0	-1	1	true

Testkörning

```
between_strict(1, 1, 10) #=> false
between_strict(1, 5, 100) #=> false
between_strict(0, -1, 1) #=> true
```

3.2.7. Min of Two

Beskrivning

Tar två tal som input och ger det minsta av dem som output.

Testdata

Input 1	Input 2	Output

100	20	20
-2	10	-2
0	1	0

Testkörning

```
min_of_two(100, 20) #=> 20
min_of_two(-2, 10) #=> -2
min_of_two(0, 1) #=> 0
```

3.2.8. Min of Three

Beskrivning

Tar tre tal som input och ger det minsta av dem som output

Testdata

Input 1	Input 2	Input 3	Output
1	2	3	1
100	2	256	2
1337	-1337	0	-1337

Testkörning

```
min_of_three(1, 2, 3) #=> 1
min_of_three(100, 2, 256) #=> 2
min_of_three(1337, -1337, 0) #=> -1337
```

3.2.9. Min of Four

Beskrivning

Tar tre fyra som input och ger det minsta av dem som output

Testdata

Input 1	Input 2	Input 3	Input 4	Output
1	2	3	4	1
100	2	256	1	1
1337	-1337	0	-1338	-1338

Testkörning

```
min_of_four(1, 2, 3, 4) #=> 1
min_of_four(100, 2, 256, 1) #=> 1
min_of_four(1337, -1337, 0, -1338) #=> -1338
```

3.2.10. Max of Two

Beskrivning

Tar två tal som input och ger det största av dem som output.

Testdata

Input 1	Input 2	Output
100	20	100
-2	10	10
0	1	1

Testkörning

```
max_of_two(100, 20) #=> 100
max_of_two(-2, 10) #=> 10
max_of_two(0, 1) #=> 1
```

3.2.11. Max of Three

Beskrivning

Tar tre tal som input och ger det största av dem som output

Testdata

Input 1	Input 2	Input 3	Output
1	2	3	3
100	2	256	256
1337	-1337	0	1337

Testkörning

```
max_of_three(1, 2, 3) #=> 3
max_of_three(100, 2, 256) #=> 256
max_of_three(1337, -1337, 0) #=> 1337
```

3.2.12. Max of Four

Beskrivning

Tar tre fyra som input och ger det största av dem som output

Testdata

Input 1	Input 2	Input 3	Input 4	Output
1	2	3	4	4
100	2	256	1	256
1337	-1337	0	-1338	1337

Testkörning

```
max_of_four(1, 2, 3, 4) #=> 4
max_of_four(100, 2, 256, 1) #=> 256
max_of_four(1337, -1337, 0, -1338) #=> 1337
```

3.3. Loopar

3.3.1. Sum To

Beskrivning

Tar ett tal som input och ger summan av alla tal från 0 till talet som output.

Testdata

Input	Output
3	6
5	15
10	55

Testkörning

```
sum_to(3) #=> 6
sum_to(5) #=> 15
sum_to(10) #=> 55
```

3.3.2. Factorial

Beskrivning

Tar ett tal som input och ger produkten av alla tal från 1 till talet som output.

Testdata

Input	Output
3	6
5	120
10	3628800

Testkörning

```
factorial(3) #=> 6
factorial(5) #=> 120
factorial(10) #=> 3628800
```

3.3.3. Power

Beskrivning

Tar två tal som input och ger potensen som output, med första input som bas och andra som exponent.

Testdata

Input 1	Input 2	Output
3	2	9
5	3	125
10	6	1000000

Testkörning

```
power(3, 2) #=> 9
power(5, 3) #=> 125
power(10, 6) #=> 1000000
```

3.4. Arrayer

3.4.1. First Of

Beskrivning

Tar en Array som input och ger det första elementet i arrayen som output.

Testdata

Input	Output
[1, 2, 3]	1
[1337, 2, -1]	1337
[0, 0, 0]	0

Testkörning

```
first_of([1, 2, 3]) #=> 1
first_of([1337, 2, -1]) #=> 1337
first_of([0, 0, 0]) #=> 0
```

3.4.2. Last Of

Beskrivning

Tar en Array som input och ger det sista elementet i arrayen som output.

Testdata

Input	Output
[1, 2, 3]	3
[1337, 2, -1]	-1
[0, 0, 0]	0

Testkörning

```
last_of([1, 2, 3]) #=> 3
last_of([1337, 2, -1]) #=> -1
last_of([0, 0, 0]) #=> 0
```

3.4.3. Append

Beskrivning

Tar en array och ett heltal som input och ger en ny array som output, där hetalet läggs till i slutet på arrayen.

Testdata

Input 1	Input 2	Output
[1, 2, 3]	4	[1, 2, 3, 4]

Testkörning

```
append([1, 2, 3], 4) #=> [1, 2, 3, 4]
```

3.4.4. Concat

Beskrivning

Tar två arrayer som input och ger en ny array som output, där båda arrayerna sätts ihop.

Testdata

Input 1	Input 2	Output
[1, 2, 3]	[4, 5, 6]	[1, 2, 3, 4, 5, 6]
[10, 10, 10]	[11, 11, 11]	[10, 10, 10, 11, 11, 11]

Testkörning

```
concat([1, 2, 3], [4, 5, 6]) #=> [1, 2, 3, 4, 5, 6]
concat([10, 10, 10], [11, 11, 11]) #=> [10, 10, 10, 11, 11, 11]
```

3.4.5. Prepend

Beskrivning

Tar En array och ett heltal som input och ger en ny array som output, där heltalet läggs i början på arrayen.

Testdata

Input 1	Input 2	Output
[1, 2, 3]	4	[4, 1, 2, 3]
[10, 10, 10]	11	[11, 10, 10, 10]

Testkörning

```
prepend([1, 2, 3], 4) #=> [4, 1, 2, 3]
prepend([10, 10, 10], 11) #=> [11, 10, 10, 10]
```

3.4.6. Sum

Beskrivning

Tar en array av integers som input och ger summan av alla tal som output

Testdata

Input	Output
[1, 2, 3, 4, 5]	15
[1337, 1337, 1337]	4011

Testkörning

```
sum([1, 2, 3, 4, 5]) #=> 15
sum([1337, 1337, 1337]) #=> 4011
```

3.4.7. Average

Beskrivning

Tar en array av integers som input och ger medelvärdet av talen som output

Testdata

Input	Output
[1, 2, 3, 4, 5]	3.0
[1337, 1337, 1337]	1337.0

Testkörning

```
average([1, 2, 3, 4, 5]) #=> 3.0
average([1337, 1337, 1337]) #=> 1337.0
```

3.4.8. Is Empty

Beskrivning

Tar en sträng som input och avgör om strängen är tom

Testdata

Input	Output
""	true
"hej hopp"	false

Testkörning

```
is_empty("") #=> true
is_empty("hej hopp") #=> false
```

3.5. Strängar

3.5.1. Starts With

Beskrivning

Tar en sträng och ett tecken som input och avgör om strängen börjar på det tecknet

Testdata

Input 1	Input 2	Output
"hej hopp"	"h"	true
"Hello World"	"k"	false
"!!zomg!!"	"!"	true

Testkörning

```
starts_with("hej hopp", "h") #=> true
starts_with("Hello World", "k") #=> false
starts_with("!!zomg!!", "!") #=> true
```

3.5.2. Ends With

Beskrivning

Tar en sträng och ett tecken som input och avgör om strängen slutar på det tecknet

Testdata

Input 1	Input 2	Output
"hej hopp"	"p"	true
"Hello World"	"W"	false
"!!zomg!!"	"!"	true

Testkörning

```
ends_with("hej hopp", "p") #=> true
ends_with("Hello World", "W") #=> false
ends_with("!!zomg!!", "!") #=> true
```

3.5.3. Chomp

Beskrivning

Tar en sträng som input och ger en ny sträng som output, där den eventuellt tagit bort "\n"-tecknet från slutet.

Testdata

Input	Output
"hej hopp\n"	"hej hopp"
"Hello World!"	"Hello World!"
"Foobar\n\n"	"Foobar\n"

Testkörning

```
chomp("hej hopp\n") #=> "hej hopp"
chomp("Hello World!") #=> "Hello World!"
chomp("Foobar\n\n") #=> "Foobar\n"
```

3.5.4. Contains Char

Beskrivning

Tar en sträng och ett tecken som input och avgör om tecknet finns i strängen.

Testdata

Input 1	Input 2	Output
"hej hopp"	"h"	true
"Hello World"	"%"	false
"Hello World"	"H"	true

Testkörning

```
contains_char("hej hopp", "h") #=> true
contains_char("Hello World", "%") #=> false
contains_char("Hello World", "H") #=> true
```

3.5.5. Index of Char

Beskrivning

Tar en sträng och ett tecken som input och ger tecknets position i strängen som output om det finns, annars nil.

Testdata

Input 1	Input 2	Output
"hej hopp"	"h"	0
"Hello World!"	"!"	11
"Hello World"	"!"	nil

Testkörning

```
index_of_char("hej hopp", "h") #=> 0
index_of_char("Hello World!", "!") #=> 11
index_of_char("Hello World", "!") #=> nil
```

3.5.6. Count

Beskrivning

Tar en sträng och ett tecken som input och ger antalet förekomster av tecknet i strängen som output.

Testdata

Input 1	Input 2	Output
"omg omg omg"	"g"	3
"Bananpaj och grillkorv"	"a"	3
".../o(. .)"	"."	12

Testkörning

```
count_char("omg omg omg", "g") #=> 3
count_char("Bananpaj och grillkorv", "a") #=> 3
count_char(".../o(. .  )", ".") #=> 12
```

3.5.7. Remove

Beskrivning

Tar en sträng och ett tecken som input och ger en ny sträng som output, där tecknet är borttaget.

Testdata

Input 1	Input 2	Output
"omg omg omg"	"g"	"om om om"

"I am a teapot"	"a "	"Imtepot"

Testkörning

```
remove_char("omg omg omg", "g") #=> "om om om"
remove_char("I am a teapot", "a ") #=> "Imtepot"
```

3.5.8. Left Strip

Beskrivning

Tar en sträng som input och ger en ny sträng som output, där all whitespace på vänster sida är borttaget. (whitespace inkluderar mellanslag" ", tabbar \t och radbrytningar \n)

Testdata

Input	Output
" Hello World! "	"Hello World! "
"\tFoobar"	"Foobar"
"\nTesttest\n"	"Testtest\n"

Testkörning

```
left_strip("   Hello World!  ") #=> "Hello World!  "
left_strip("\tFoobar") #=> "Foobar"
left_strip("\nTesttest\n") #=> "Testtest\n"
```

3.5.9. Right Strip

Beskrivning

Tar en sträng som input och ger en ny sträng som output, där all whitespace på höger sida är borttaget. (whitespace inkluderar mellanslag" ", tabbar \t och radbrytningar \n)

Testdata

Input	Output
" Hello World! "	" Hello World!"
"\tFoobar"	"\tFoobar"
"\nTesttest\n"	"\nTesttest"

Testkörning

```
right_strip("   Hello World!  ") #=> "   Hello World!"
right_strip("\tFoobar") #=> "\tFoobar"
right_strip("\nTesttest\n") #=> "\nTesttest"
```

3.5.10. Strip

Beskrivning

Tar en sträng som input och ger en ny sträng som output, där all whitespace på båda sidor är borttaget. (whitespace inkluderar mellanslag" ", tabbar \t och radbrytningar \n)

Testdata

Input	Output
" Hello World! "	"Hello World!"
"\tFoobar"	"Foobar"
"\nTesttest\n"	"Testtest"

Testkörning

```
strip("   Hello World!  ") #=> "Hello World!"
strip("\tFoobar") #=> "Foobar"
strip("\nTesttest\n") #=> "Testtest"
```

3.5.11. Replace Char

Beskrivning

Tar tre strängar som input och en ny sträng som output där alla förekomster av sträng2 i sträng1 är ersatt med sträng3

Testdata

Input	Output
"Hello World!"	"\"H\""
"\"F\""	"Hello World!"
"Foobar"	"\"F\""
"\"B\""	"Foobar"

Testkörning

```
replace_char("Hello World!", "\"H\"", "\"F\"") #=> "Hello World!"
replace_char("Foobar", "\"F\"", "\"B\"") #=> "Foobar"
```

3.5.12. Slice

Beskrivning

Tar en sträng och två integers som input och ger en ny sträng som output, där alla tecken i strängen mellan talen är urklippta.

Testdata

Input 1	Input 2	Input 3	Output
"Hello World"	2	8	"llo Worl"
"Foobar"	0	1	"F"

Testkörning

```
slice("Hello World", 2, 8) #=> "llo Worl"
slice("Foobar", 0, 1) #=> "F"
```

3.5.13. Split Char

Beskrivning

Tar en sträng och ett tecken som input och ger en array som output, där elementen i arrayen är alla delar av strängen som är avskiljda med tecknet

Testdata

Input 1	Input 2	Output
"1;2;3;4;5"	";"	["1", "2", "3", "4", "5"]
"Hello World"	" "	["Hello", "World"]
"This is a line\nthis is another line\nthis is a line too"	"\n"	["This is a line", "this is another line", "this is a line too"]

Testkörning

```
split_char("1;2;3;4;5", ";") #=> ["1", "2", "3", "4", "5"]
split_char("Hello World", " ") #=> ["Hello", "World"]
split_char("This is a line\nthis is another line\nthis is a line too", "\n") #=>
["This is a line", "this is another line", "this is a line too"]
```

3.6. Mer Arrayer

3.6.1. Filter

Beskrivning

Tar en array och ett värde som input och ger en ny array som output, där den nya arrayen endast innehåller värdet.

Testdata

Input 1	Input 2	Output
[8, 2, 0, 2, 5]	"2"	[]
["bosse", "olof", "kalle", "olof"]	"olof"	["olof", "olof"]

Testkörning

```
filter([8, 2, 0, 2, 5], "2") #=> []
filter(["bosse", "olof", "kalle", "olof"], "olof") #=> ["olof", "olof"]
```

3.6.2. Exclude

Beskrivning

Tar en array och ett värde som input och ger en ny array som output, där den nya arrayen **inte** innehåller värdet.

Testdata

Input 1	Input 2	Output
[8, 2, 0, 2, 5]	"2"	[8, 2, 0, 2, 5]
["bosse", "olof", "kalle", "olof"]	"olof"	["bosse", "kalle"]

Testkörning

```
exclude([8, 2, 0, 2, 5], "2") #=> [8, 2, 0, 2, 5]
exclude(["bosse", "olof", "kalle", "olof"], "olof") #=> ["bosse", "kalle"]
```

3.6.3. Unique

Beskrivning

Tar en array som input och en ny array som output, där den nya arrayen inte innehåller några dubletter.

Testdata

Input	Output
[8, 2, 0, 2, 5]	[8, 2, 0, 5]
["bosse", "olof", "kalle", "olof"]	["bosse", "olof", "kalle"]

Testkörning

```
unique([8, 2, 0, 2, 5]) #=> [8, 2, 0, 5]
unique(["bosse", "olof", "kalle", "olof"]) #=> ["bosse", "olof", "kalle"]
```

3.6.4. Count

Beskrivning

Tar en array och ett värde som input och ger antalet förekomster av värdet i arrayen som output

Testdata

Input 1	Input 2	Output
[8, 2, 0, 2, 5, 0, 0, 0]	0	4
["bosse", "olof", "kalle", "olof"]	"olof"	2

Testkörning

```
count([8, 2, 0, 2, 5, 0, 0, 0], 0) #=> 4
count(["bosse", "olof", "kalle", "olof"], "olof") #=> 2
```

3.6.5. Contains

Beskrivning

Tar en array och ett värde som input och av gör om arrayen innehåller värdet

Testdata

Input 1	Input 2	Output
[8, 2, 0, 2, 5, 0, 0, 0]	0	true
["bosse", "olof", "kalle", "olof"]	"gunilla"	false

Testkörning

```
contains([8, 2, 0, 2, 5, 0, 0, 0], 0) #=> true
contains(["bosse", "olof", "kalle", "olof"], "gunilla") #=> false
```

3.7. Mer Strängar

3.7.1. Contains String

Beskrivning

Tar två strängar som input och avgör om den första strängen innehåller den andra strängen.

Testdata

Input 1	Input 2	Output
"foobar"	"bar"	true
"foobar"	"baz"	false

Testkörning

```
contains_string("foobar", "bar") #=> true
contains_string("foobar", "baz") #=> false
```

3.7.2. Index String

Beskrivning

Tar två strängar som input och ger positionen av den andra strängen i den första strängen som output.

Testdata

Input 1	Input 2	Output
"foobar"	"bar"	3
"foobar"	"baz"	nil

Testkörning

```
index_string("foobar", "bar") #=> 3
index_string("foobar", "baz") #=> nil
```

3.7.3. Count String

Beskrivning

Tar två strängar som input och ger antalet förekomster av den andra strängen i den första strängen som output.

Testdata

Input 1	Input 2	Output
"omg omg omg"	"omg"	3
"examples are hard\nexamples are hard\nexamples are hard"	"examples"	3

Testkörning

```
count_string("omg omg omg", "omg") #=> 3
count_string("examples are hard\nexamples are hard\nexamples are hard", "examples")
#=> 3
```

3.7.4. Remove String

Beskrivning

Tar två strängar som input och ger en sträng som output, där den nya strängen inte innehåller den andra strängen.

Testdata

Input 1	Input 2	Output
"omg omg omg"	" "	"omgomgomg"
"this is a test"	" test"	"this is a"

Testkörning

```
remove_string("omg omg omg", " ") #=> "omgomgomg"
remove_string("this is a test", " test") #=> "this is a"
```

3.7.5. Replace String

Beskrivning

Tar tre strängar som input och ger en sträng som output, där den nya strängen har varje förekomst av sträng2 ersatt av sträng3.

Testdata

Input 1	Input 2	Input 3	Output
"omg omg omg"	"omg"	"foo"	"foo foo foo"
"examples are hard"	"hard"	"tough"	"examples are tough"

Testkörning

```
replace_string("omg omg omg", "omg", "foo") #=> "foo foo foo"
replace_string("examples are hard", "hard", "tough") #=> "examples are tough"
```

3.7.6. Split String

Beskrivning

Tar två strängar som input och en array som output där arrayen innehåller delarna av strängen avgränsat med den andra strängen.

Testdata

Input 1	Input 2	Output
"a bunch of text"	" "	["a", "bunch", "of", "text"]
"This is a line\nthis is another line\nthis is also a line"	"\n"	["This is a line", "this is another line", "this is also a line"]

Testkörning

```
split_string("a bunch of text", " ") #=> ["a", "bunch", "of", "text"]
split_string("This is a line\nthis is another line\nthis is also a line", "\n") #=>
["This is a line", "this is another line", "this is also a line"]
```

3.8. Sorteringsalgoritmer

3.8.1. Selection Sort

Beskrivning

Tar en array som input och ger en ny array som output, där arrayen är sorterad i storleksordning. Funktionen följer algoritmen för Selection Sort.

Testdata

Input	Output
[7, 3, 1, 3]	[1, 3, 3, 7]
[0, -1, 2, -1, 9, -1]	[-1, -1, -1, 0, 2, 9]

Testkörning

```
selection_sort([7, 3, 1, 3]) #=> [1, 3, 3, 7]
selection_sort([0, -1, 2, -1, 9, -1]) #=> [-1, -1, -1, 0, 2, 9]
```

3.8.2. Insertion Sort

Beskrivning

Tar en array som input och ger en ny array som output, där arrayen är sorterad i storleksordning. Funktionen följer algoritmen för Insertion Sort.

Testdata

Input	Output
[7, 3, 1, 3]	[1, 3, 3, 7]
[0, -1, 2, -1, 9, -1]	[-1, -1, -1, 0, 2, 9]

Testkörning

```
insertion_sort([7, 3, 1, 3]) #=> [1, 3, 3, 7]
insertion_sort([0, -1, 2, -1, 9, -1]) #=> [-1, -1, -1, 0, 2, 9]
```

3.8.3. Bubble Sort

Beskrivning

Tar en array som input och ger en ny array som output, där arrayen är sorterad i storleksordning. Funktionen följer algoritmen för Bubble Sort.

Testdata

Input	Output
[7, 3, 1, 3]	[1, 3, 3, 7]
[0, -1, 2, -1, 9, -1]	[-1, -1, -1, 0, 2, 9]

Testkörning

```
bubble_sort([7, 3, 1, 3]) #=> [1, 3, 3, 7]
bubble_sort([0, -1, 2, -1, 9, -1]) #=> [-1, -1, -1, 0, 2, 9]
```

3.8.4. Quick Sort

Beskrivning

Tar en array som input och ger en ny array som output, där arrayen är sorterad i storleksordning. Funktionen följer algoritmen för Quick Sort.

Testdata

Input	Output
[7, 3, 1, 3]	[1, 3, 3, 7]
[0, -1, 2, -1, 9, -1]	[-1, -1, -1, 0, 2, 9]

Testkörning

```
quick_sort([7, 3, 1, 3]) #=> [1, 3, 3, 7]
quick_sort([0, -1, 2, -1, 9, -1]) #=> [-1, -1, -1, 0, 2, 9]
```