

Programming 1

Table of Contents

1. Flödesscheman	1
1.1. Komponenter	1
1.1.1. Input	1
1.1.2. Output	1
1.1.3. State Change	2
1.1.4. Conditions	2
1.1.5. Loops	2
1.2. Funktioner i flödesscheman	3
2. Datatyper	3
2.1. Heltal (Integer)	3
2.2. Strängar (String)	3
2.2.1. Indexering	3
2.2.2. Konkatenering	4
3. Verktöglådan	5
3.1. Verktyg för ändring av State	5
3.1.1. Tilldelning	5
3.1.2. Inkrementering / Dekrementering	5
3.2. Verktyg för uppdelning av flödet (Villkor)	6
3.2.1. If	6
3.2.2. If-Else	6
3.2.3. If-Elseif-Else	7
3.3. Verktyg för upprepning av flödet (Loopar)	7
3.3.1. Räknande loop	7
3.3.2. Okänt antal iterationer	9
3.4. Verktyg för funktioner	9
3.4.1. Outputvariabel	9
3.4.2. Iterativt uppbyggd output	9

1. Flödesscheman

För att modellera hur en funktionsmaskin fungerar använder vi flödesscheman. Flödesscheman består av komponenter som antingen

- ändrar på state eller
- styr flödet beroende på state.

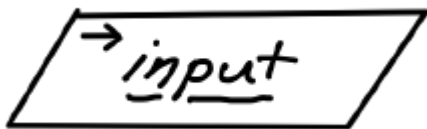
Komponenterna kopplas ihop med pilar och bildar ett flöde.

1.1. Komponenter

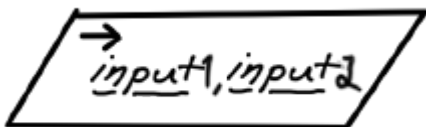
Alla komponenter i flödesscheman interagerar med **state** eller **input** på något vis. State består av variabler som indikeras med understruken text.

1.1.1. Input

Input-komponenten illustreras med en parallelogram med en liten pil till vänster. I rutan skrivs namnet på input så man kan referera till den som en variabel senare i flödet.

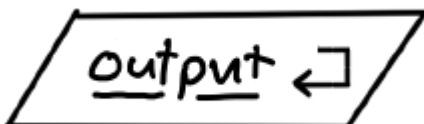


Om en situation kräver flera inputs separeras dessa med komma-tecken.



1.1.2. Output

Output-komponenten illustreras också med en parallelogram fast med en retur-pil på högersidan. I rutan skrivs variabeln som innehåller värdet som ska ges som output.



Om man i flödet når en **output**-komponent så avslutas flödet direkt.

1.1.3. State Change

För att ändra på state, antingen genom att

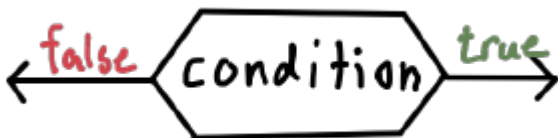
- Skapa en ny variabel, eller
- Ändra på en befintlig variabel

illustreras detta med en rektangel. I rutan skriver man på vilket sätt som variabeln ska ändras. Antingen genom att använda tilldelningstecken eller med ord beskriva vad som händer.



1.1.4. Conditions

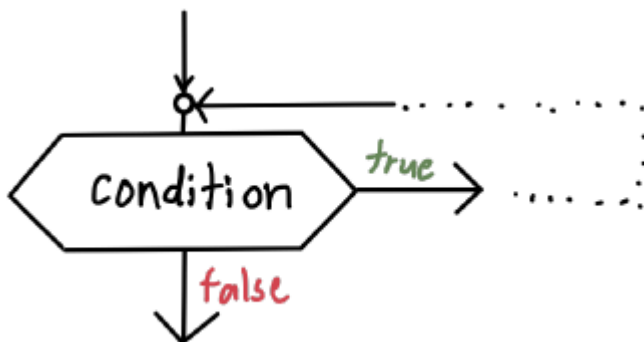
En **condition**-komponent illustreras med en utdragen hexagon. Den används för att dela flödet i två olika vägar. Den har alltid **en** ingång och **två** utgångar, en **true**-utgång och en **false**-utgång. I rutan skriver man en jämförelse mellan två variabler.



true-utgången går vanligtvis åt höger och **false**-utgången går vanligtvis rakt ner.

1.1.5. Loops

En **loop**-komponent illustreras nästan som en **condition**-komponent, egentligen är de samma sak. Skillnaden är en liten ring ovanför hexagonen och att false-utgången vanligtvis går rakt ner. I rutan skrivs **loopvillkoret**.



Loopvillkoret måste bero på något state som ändras i true-delen, och det måste ändras på ett sätt så att **loopvillkoret** någon gång blir falskt. Annars får man en oändlig loop.

1.2. Funktioner i flödesscheman

Flödesscheman kan beskriva bara en liten del av ett program eller så kan de beskriva en hel funktion. En hel funktion beskrivs genom att flödesscheman har en **input**-komponent och en **output**-komponent.

2. Datatyper

Internt i en dator lagras allting som binära tal; 1 eller 0. Att enbart jobba med ettor och nollor när man programmerar hade dock varit väldigt begränsande.

Därför har de flesta programmeringsspråk olika datatyper. Detta innebär att programmeringsspråket tolkar olika kombinationer av ettor och nollor på olika sätt. Tex kan programmeringsspråket tolka en viss sekvens av ettor och nollor som tecknet "A", och en annan sekvens av ettor och nollor som talet 67. Precis hur detta går till är inte relevant för kursen programmering 1, men om du är nyfiken kan du se [Tom Scotts "How To Read Text in Binary"](#).

2.1. Heltal (Integer)

2.2. Strängar (String)

Strängar är datatypen som hanterar text. I ruby skapar man och tilldelar en variabel en en sträng på följande sätt:

```
name = "Tom Scott" ①
```

① Tilldelar variabeln `name` värdet "Tom Scott" (observera citationstecknen).

Strängar i ruby avgränsas av citationstecken (även kallade dubbelfnuttar).

2.2.1. Indexering

Till skillnad från heltal, som enbart innehåller ett enda värde, innehåller strängar i själva verket flera värden. Varje värde (tecken) i en sträng går att komma åt var för sig.

För att kunna bestämma vilket tecken man vill kolla på använder man sig av "index". Detta innebär att man använder tecknets position (index) i strängen för att avgöra vilket tecken man vill kolla på.

Strängen "Tom Scott" innehåller 9 tecken ("T", "o", "m", " " (blanksteg), "S", "c", "o", "t", "t"). Varje tecken har en position (index). Man räknar index från det första tecknet till det sista, men man det första tecknets index är 0.

```
name = "Tom Scott"
#index 012345678
```

Detta innebär att en strängs sista teckens index alltid är "längden på strängen - 1" (dvs en sträng

som är tusen tecken lång har 999 som högsta index).

Index används när man vill kolla på ett enskilda tecken åt gången:

```
i = 2
name = "Tom Scott"
if name[2] == "m" ①
  output = "The third character is an "m"
end
```

① Index avgränsas av hakparenteser ([och]). Utläses som "tecknet på index 2".

Strängar vet alltid hur många tecken de innehåller. Man kan fråga en variabel innehållandes en sträng hur många tecken strängen innehåller:

```
name = "Tom Scott"

if name.length > 20 ①
  output = "That's a very long name"
end
```

① `.length` returnerar strängens längd. I det här fallet 9. Villkoret kan alltså (i det här fallet) utläsas som "9 är större än 20").

2.2.2. Konkaterering

I bland behöver man bygga upp en sträng tecken för tecken. Man slår då ihop (konkatenerar) två olika strängar. Operatoren för att konkatenera två strängar är `+`. Det är lätt att förväxla den med operatoren för att addera två tal, men de gör olika saker. För att konkateneringsoperatoren ska fungera måste både det som står till vänster om operatoren och det som står till höger om operatoren vara strängar.

```
output = ""
first_name = "Tom"
last_name = "Scot"
i = 3

output = output + first_name ①
output = output + " " ②
output = output + last_name ③
output = output + last_name[i] ④
```

① `output` konkateneras med `first_name` ("Tom") (och blir "Tom").

② `output` konkateneras med " " (och blir "Tom ").

③ `output` konkateneras med `last_name` ("Scot") (och blir "Tom Scot").

④ `output` konkateneras med `last_name[i]` ("t") (och blir "Tom Scott").

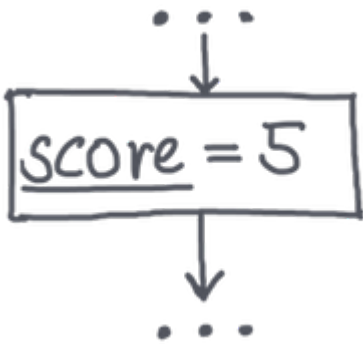
3. Verktygslådan

3.1. Verktyg för ändring av State

3.1.1. 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**.

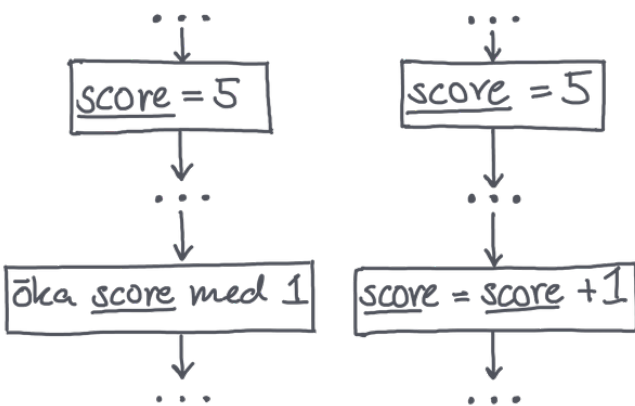
Flödesschema	Kod
	<pre>score = 5</pre>

3.1.2. Inkrementering / Dekrementering

Inkrementering

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

Oftast används inkrementeringsverktyget i samband med loopar.

Flödesschema	Kod
	<pre>score = 5 score = score + 1</pre>

Dekrementering

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

Flödesschema	Kod
<pre> graph TD In1[...] --> S1[score = 5] S1 --> D1[...] D1 --> M1[minska score med 1] M1 --> Out1[...] In2[...] --> S2[score = 5] S2 --> D2[...] D2 --> M2[score = score - 1] M2 --> Out2[...]</pre>	<pre> score = 5 score = score - 1</pre>

Oftast används dekrementeringsverktyget i samband med loopar.

3.2. Verktyg för uppdelning av flödet (Villkor)

3.2.1. 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.

Flödesschema	Kod
<pre> graph TD In[] --> Cond{condition} Cond -- false --> Join((+)) Cond -- true --> DoStuff[do stuff] DoStuff --> Join Join --> Out[]</pre>	<pre> if condition # True-delen end # Sammanfogningen</pre>

3.2.2. 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**.

Flödesschema	Kod
<pre> graph TD Start(()) --> Condition{condition} Condition -- false --> DoStuff1[do stuff] Condition -- true --> DoStuff2[do stuff] DoStuff1 --> Join(()) DoStuff2 --> Join Join -- sammanfogning --> Exit(()) </pre>	<pre> if condition # True-delen else # False-delen end #Sammanfogning </pre>

3.2.3. If-Elseif-Else

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

Flödesschema	Kod
<pre> graph TD Start(()) --> C1{condition1} C1 -- true --> DS1[do stuff] C1 -- false --> C2{condition2} C2 -- true --> DS2[do stuff] C2 -- false --> C3{condition3} C3 -- true --> DS3[do stuff] C3 -- false --> DS4[do stuff] DS1 --> Join(()) DS2 --> Join DS3 --> Join DS4 --> Join Join -- sammanfogning --> Exit(()) </pre>	<pre> if condition1 # True-delen för condition1 elseif condition2 # True-delen för condition2 elseif condition3 # True-delen för condition3 else # False-delen för alla conditions end #Sammanfogning </pre>

3.3. Verktyg för upprepning av flödet (Loopar)

3.3.1. Räknande loop

Inkrementerande

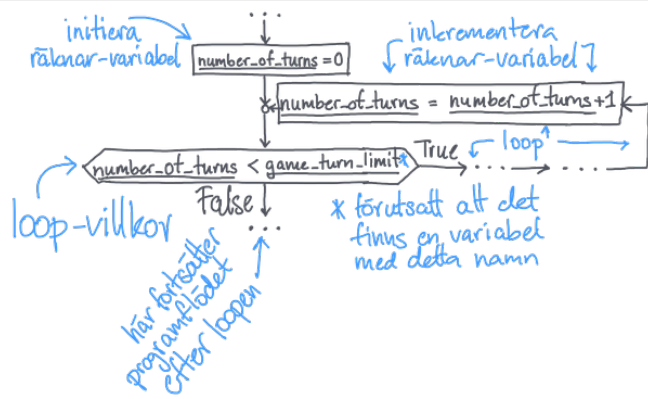
En inkrementerande loop passar bra att använda om du i förväg vet (eller kan räkna ut) hur många gånger programflödet behöver upprepas, t.ex. om programmet alltid ska göra något 5 gånger, eller om vet att du ska göra något lika många gånger som du har tal i en lista.

Inkrementerande loopar använder sig av en *räknar-variabel*. En räknar-variabel håller koll på hur

många gånger loopen upprepas. Vanliga namn på räknar-variabler är **counter**, **iterations** eller **i**, men en räknar-variabel kan heta precis vad som helst.

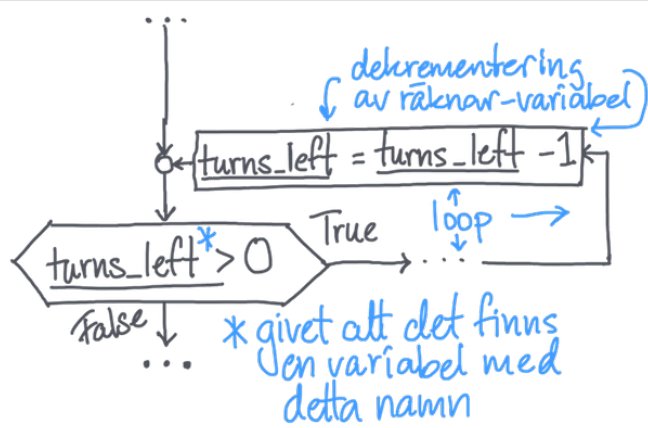
För att en räknande loop ska fungera behöver följande steg ske:

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 loopens fortsätta i all evighet, eftersom loop-villkoret aldrig blir falskt.

Flödesschema	Kod
	<pre>iterations = 0 number_of_iterations = 10 while iterations < number_of_iterations # Do stuff iterations = iterations + 1 end</pre>

Dekrementerande

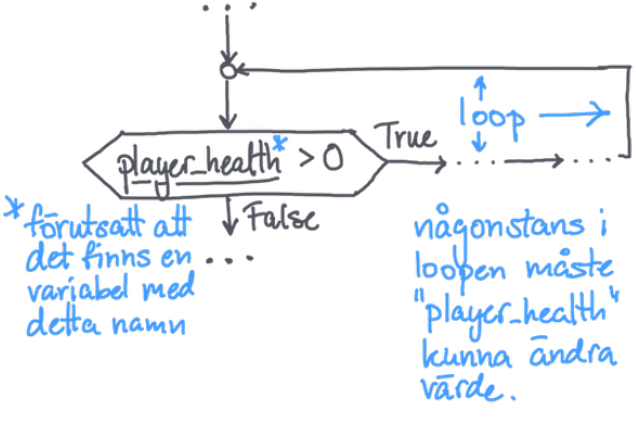
En dekrementerande loop har i stort sett samma användningsområde som en inkrementerande loop, det vill säga, du vet (eller kan räkna ut) hur många gånger du behöver upprepa programflödet.

Flödesschema	Kod
	<pre>iterations_left = 10 while iterations_left > 0 # Do stuff iterations = iterations - 1 end</pre>

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

3.3.2. 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.

Flödesschema	Kod
	<pre>while player_health > 0 # Do stuff end</pre>

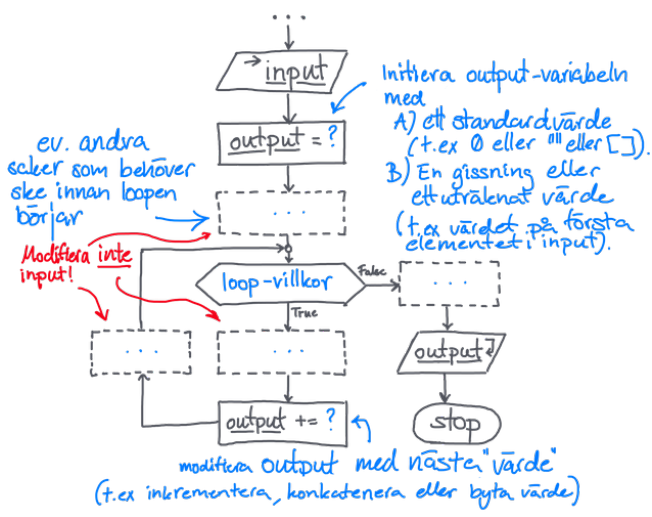
3.4. Verktyg för funktioner

3.4.1. Outputvariabel

När en funktion ska behandla **input** på något vis, och returnera någon **output** beroende på beräkningen så är en outputvariabel väldigt användbar. Outputvariabeln håller värdet på det som ska ges som output när funktionen har körts klart. Det krävs att outputvariabeln **initieras** med något startvärde. Startvärdet kan vara vilket värde som helst eller att man kopierar input.

3.4.2. Iterativt uppbyggd output

Ett vanligt användningsområde för en **outputvariabel** är när man iterativt bygger upp en **output**. Detta verktyg är särskilt användbart när varje iteration beror på resultatet av iterationen innan.

Flödesschema	Kod
	<pre>def function(input) output = ? # ... while condition # ... output = output + ? end return output end</pre>