Utvecklarblogg - Seminarium 1

Grupp B6 - guser766 | hadan326

Att lära sig Ruby (nya saker)

För att lära oss Ruby försökte vi så gott som möjligt att använda den officiella dokumentationen på https://docs.ruby-lang.org/en/2.7.0/.
Själva dokumentationen på denna sida var rätt bra, men vi tyckte det var svårt att få en överblick eftersom startsidan inte presenterade olika områden så bra, mer än en spalt på vänstersidan med diverse sidor.
Vi föredrog att istället göra en websökning på det vi var ute efter, och när vi hittade ett svar som gav förslag på en metod, kunde vi gå tillbaka och läsa den officiella dokumentationen för att få en utökad förståelse.

Ordbaserad syntax

Något som liknar Python, men som skiljer sig från C++, är Rubys ordbaserade syntax. T ex:

```
def n_times (n, &block)
  for i in 1..n do
        block.call
  end
end
```

For-loopens block startar här med *do* och avslutas med *end.* Dock verkar det som att *do* här är frivilligt. Överlag föredrar vi nog "måsvingarna" från C++, eftersom de ger koden ett mer standardiserat utseende som gör det lättare att läsa koden, och framförallt att läsa andras kod.
Det kan dock vara lättare att läsa logiska uttryck med ordbaserad syntax, beroende på hur van man är att läsa kodbaserad syntax. T ex:

```
if i > 2 then f = 2 else puts "Hello" end
```

Jämför detta med:

```
if (i > 2) {
        f = 2
}
else {
        puts "Hello"
}
```

Vi var förvirrade över att funktioner kan ha block som en inparameter, men att de inte skrivs

Tillägg i klasser

Det var förvånande lätt att lägga till funktioner till en klass. Det enda man behöver är att göra en ny definition av en befintlig klass, och sedan lägga till en ny funktion i den. T ex:

```
class Integer
  def fib
```

```
        if self == 1 or self == 2
        1
        else
        (self-1).fib + (self-2).fib
        end
    end
end
```

Det som också var spännande med Ruby var att man kunde ta sig friheter med de klasser som i C++ hade klassats som primitiver. Hade vi velat göra samma fib-funktion i C++ hade man istället fått göra en fristående funktion som tog in siffran som parameter.

Setters och getters

Vi gillar strukturen som Ruby har för variabelåtkomst, med attr_accessors och att skapa egna setters/getters med

def foo=(rhs) och *def foo*

Eftersom alla variabler är privata som standard, är det svårare att göra misstag när det kommer till inkapsling, då man explicit måste ge tillgång till variabler. Att flagga enkla medlemsvariabler med attr_[ ] var också väldigt smidigt, då det i C++ var jobbigt att behöva skriva funktioner vars enda uppdrag var att returnera en variabel. Det är även snyggt att både

def foo=(rhs) och attr_writer :foo

kallas med

class.foo = rhs

då detta förenklar abstraktionen för slutanvändaren, då man alltid använder samma syntax.

For Loop

För att iterera något n gånger skriver man:

for i in 1..n do

Detta liknade vad vi sett tidigare i Python:

for i in range (1, n)

Skillnaden är dock att iteratorn i Ruby går från 1 till n, men i Python går den från 1 till n-1.

Ruby: n = 3 -> 3 upprepningar

Python: n = 3 -> 2 upprepningar

Dock upptäckte vi att man kan få samma funktionalitet som i Python, alltså att sista talet ignoreras, om man använder tre punkter:

for i in 1...n

Det är bra att Ruby har den valfriheten, men det känns som att man lätt kan missa den extra punkten, vilket kan göra det svårare att felsöka sin kod.

Tolkning av uppgifter och förklaring av lösningar

Uppgift 5

I vår initiala lösning för den här uppgiften gjorde vi egna getters och setters på formatet set_name(string), eftersom vi alltid vill hantera hela namn, både för get och set, och bara hade de uppdelade medlemsvariablerna name och surname. Vi antog alltså att man inte kunde applicera attr_accessors i det här fallet.
Man kan i och för sig inte använda just attr_accessor för mer komplicerad variabelhantering, men t ex en setter kan skrivas på formatet

```
def fullname=(rhs) {kodblock}
```

och kan sedan då sättas med

```
person.fullname = rhs_string
```

Vår getter returnerade först "@surname + " " + @name", men vi insåg sen att det var bättre med "#{@surname} #{@name}" då det är enklare att formatera texten på det viset.

Uppgift 6

I denna uppgift var vi kluvna kring om vi skulle använda *default* eller *optional* parameters. Default liknade det vi sett tidigare i C++, men formuleringen "valfri" fick oss att tro att man menade just optional parameters.
Eftersom vi inte anger ett exakt födelsedatum förutsätter vi att man matar in den ålder man kommer bli samma år som man matar in ålder.

Uppgift 7

Vår tolkning av uppgiften är att *n.fib* ska returnera tal nummer n i fibonacci-serien. Alltså:

```
1.fib = 1
2.fib = 1
3.fib = 2
n.fib = (n-1).fib + (n-2).fib
```

Det finns inget "nollte" tal i serien, utan det första talet är 1.
Vi löste uppgiften med en rekursiv funktion, som returnerar 1 om n = 1 eller 2, annars returnerar den (n-1).fib + (n-2).fib

Uppgift 8

För att lägga till en funktion i den befintliga klassen String gjorde vi en extra klass-definition med samma namn som endast innehöll den nya funktionen.
Funktionen kallar på sig själv med *self*, splittar upp strängen med *split* vid varje mellanrum, loopar över det splittade resultatet och sparar ner den första bokstaven i varje ord i en variabel som slutligen returneras.
Som vi tolkade uppgiften behöver man inte göra några ytterligare kontroller av strängen, utan alla tecken är godkända och sätts endast till versaler.

Uppgift 10

Som vi tolkar uppgiften ska vi göra en funktion som returnerar allt efter

[bokstäver][kolon][mellanslag]

Vi löste detta genom att först hitta en matchning för ovan beskrivna format med regex-uttrycket

[a-zA-z]+:\s+

med funktionen

m = re.match(string)

och sedan returnera allt som kommer efter den matchningen med

m.post_match

Uppgift 12

I enlighet med reglerna för registreringsskyltar ska strängen innehålla ett uttryck med 3 stora bokstäver, 2 siffror och avslutas med antingen en stor bokstav eller siffra.
I, Q och V är förbjudna, och ett regnummer får dessutom inte sluta på bokstaven O.
Vi tänkte först att man kunde använda ^ för att exkludera de förbjudna bokstäverna, men det finns inget OCH i regex, så det gick inte att säga “3 stora bokstäver OCH de är inte IVQO”. Istället skrev vi ut exakt vilka intervall som var godkända. Regex-uttrycket blev då:

[A-HJ-PR-UW-Z]{3}\d{2}([A-HJ-NPR-UW-Z]|\d)

Vi använde

re.match(string)

för att hitta eventuella träffar. Om resultatet var skiljt från *nil* returnerade vi den första träffen, annars *false*.