Εντολές ελέγχου και συναρτήσεις στη Ruby

Εισαγωγή

Στο προηγούμενο μάθημα παρουσιάσαμε τα γενικά χαρακτηριστικά της Ruby και ασχοληθήκαμε με την irb (ακόμα και για την online έκδοση). Είδαμε το πόσο εύκολα εγκαθίσταται η Ruby, μιλήσαμε για τους τελεστές της, τα ruby gems καθώς επίσης για πως κάνουμε απλά loops και για την εντολή if.

Αυτή τη φορά θα ασχοληθούμε με άλλες εντολές ελέγχου και βρόγχου καθώς και με το πως γράφουμε δικές μας συναρτήσεις. Εάν γνωρίζετε κάποια άλλη γλώσσα προγραμματισμού δε θα δυσκολευτείτε καθόλου από όσα θα δούμε σήμερα.

Εντολές ελέγχου και βρόχων επανάληψης

Τελεστής ? :

Έχουμε ήδη μιλήσει για την εντολή if στη Ruby, πριν όμως προχωρήσουμε αξίζει να αναφέρουμε ότι στη Ruby θα βρείτε τον τελεστή που συναντάμε και στη C/C++ "συνθήκη **?** εντολή εάν ισχύει η συνθήκη **:** εντολή εάν ΔΕΝ ισχύει η συνθήκη". Δηλαδή εάν ισχύει η συνθήκη εκτελείται η πρώτη εντολή, αλλιώς η δεύτερη.

Δοκιμάστε το εξής αρχείο (if_operator.rb) που περιέχει κώδικα από το επίσημο wiki books της Ruby (http://en.wikibooks.org/wiki/Ruby_Programming/Syntax/Control_Structures , στο οποίο μπορείτε να βρείτε πρόσθετο υλικό για μελέτη):

```
a = 5
plus_or_minus = '+'
print "The number #{a}#{plus_or_minus}1 is: " + (plus_or_minus == '+' ? (a+1).to_s : (a-1).to_s) + "."
=>The number 5+1 is: 6.
```

Το #{} και τη μέθοδο to_s τα αναφέραμε την προηγούμενη φορά. Η print χρησιμοποιείται εδώ στη θέση της puts. Εν συντομία, ο τελεστής ελέγχει εάν θέλουμε να κάνουμε πρόσθεση ή αφαίρεση και εκτελεί την αντίστοιχη πράξη. Δοκιμάστε να αλλάξετε το plus_or_minus σε '-'. Επίσης σημειώστε ότι τα + που είναι υπογραμμισμένα χρησιμοποιούνται για να προσθέσουν Strings , πρακτικά "κολλάνε" το κείμενο με τις επεξηγήσεις με το αποτέλεσμα του τελεστή ? :.

.times

Έχουμε ήδη δει την 1.upto(5). Στη Ruby υπάρχει ένας ακόμα απλός τρόπος για τη δημιουργία loops. Δείτε τον παρακάτω κώδικα (αρχείο times.rb):

```
6.times{puts "This is a simple loop"}
puts ""
5.times do
        puts "And this is a way to write a loop with more than one instructions"
end
=>
This is a simple loop
This is a simple loop
This is a simple loop
This is a simple loop
This is a simple loop
This is a simple loop

And this is a way to write a loop with more than one instructions
And this is a way to write a loop with more than one instructions
And this is a way to write a loop with more than one instructions
And this is a way to write a loop with more than one instructions
And this is a way to write a loop with more than one instructions
```

Είναι εύκολο να καταλάβουμε ότι η times είναι ιδιαίτερα βολική για απλά loops στα οποία ξέρουμε πόσες επαναλήψεις θέλουμε. Επίσης, σημειώστε ότι γενικά στη Ruby μπορούμε να έχουμε {} με μία εντολή μέσα τους ή να χρησιμοποιήσουμε τη δομή do-end ώστε να ορίσουμε ένα block εντολών.

unless

Συνεχίζουμε με την εντολή unless. Από το όνομα και μόνο μπορείτε να καταλάβετε γιατί μιλάμε. Δοκιμάστε να τρέξετε τον παρακάτω κώδικα (unless.rb)

```
1.upto(5) do |x|
        unless x!=3#αντί δηλαδή του if x==3
                x=-3
        end
        puts x
end
=>
1
2
-3
4
5
```

Όπως παρατηρείτε, πρόκειται για την αντίθετη ή καλύτερα δυϊκή της if. Δηλαδή

η εντολή ή οι εντολές μέσα στο block unless-end εκτελείται/εκτελούνται όταν δεν ισχύει η συνθήκη ελέγχου. Δοκιμάστε για παράδειγμα να αλλάξετε τον έλεγχο από x!=3 σε x==3. Τι παρατηρείτε;

case

Όπως σχεδόν σε όλες τις γλώσσες έτσι και εδώ με την case ελέγχουμε την τιμή μίας μεταβλητής χωρίς να βάλουμε πολλαπλά if ή if-else. Ας δούμε ένα παράδειγμα όπου ελέγχουμε την τιμή ενός αριθμού (αρχείο case.rb):

```
number = 20
puts case number
        when 0..100 then "#{number} is less than 100"
        when 100..200 then "#{number} is greater than 100 and less than 200"
        when 200..300 then "#{number} is greater than 200 and less than 300"
        else "#{number} out of range.Please try again with different number."
end
=>20 is less than 100
```

Όπως είναι φανερό η δομή της εντολής είναι:

```
case τιμή για έλεγχο
        when τιμή then …
        when τιμή then …
        else #προληπτικά, εκτελείται εφόσον δεν ισχύει καμία από τις παραπάνω
συνθήκες
end
```

Με τη βοήθεια της κλάσης Range, που είδαμε και την προηγούμενη φορά, μπορούμε να ελέγξουμε όχι μόνο εάν η μεταβλητή number έχει μία συγκεκριμένη τιμή, όπως π.χ. σε άλλες γλώσσες, αλλά ακόμα και αν ανήκει σε συγκεκριμένο εύρος τιμών.

Επίσης, μπορούμε να βάλουμε την εντολή puts πριν την case. Σε αυτή την περίπτωση η case δίνει ως έξοδο το κείμενο που είναι να εμφανιστεί κατά περίπτωση και η puts το δέχεται ως παράμετρο. Δηλαδή το αποτέλεσμα της μίας εντολής (εδώ της case) είναι είσοδος της άλλης (εν προκειμένω της puts). Εναλλακτικά μπορούμε να γράψουμε την εντολή puts σε κάθε μία περίπτωση (when then) του case. Δηλαδή ο παραπάνω κώδικας θα μπορούσε να γραφεί και ως :

```
number = 20
case number
        when 0..100 then puts "#{number} is less than 100"
        when 100..200 then puts "#{number} is greater than 100 and less than 200"
        when 200..300 then puts "#{number} is greater than 200 and less than 300"
        else puts "#{number} out of range.Please try again with different number."
end
=>20 is less than 100
```

Εννοείται ότι μπορείτε να χρησιμοποιείτε όποιον τρόπο θέλετε καθώς δεν έχουν κάποια ιδιαίτερη διαφορά.

while

Δε θα μπορούσε φυσικά να λείπει η while. Η μορφή της είναι :

```
while συνθήκη
...
end
```

Ένα απλό παράδειγμα χρήσης της μπορείτε να βρείτε στο αρχείο while.rb :

```
number = 5#πρέπει να είναι >=1
result = 1
while number!=1
        result*=number
        number-=1
end
puts result
=>120
```

Η λογική είναι πολύ απλή. Στην πράξη με αυτό το while υπολογίζουμε το παραγοντικό του αριθμού number. Να σημειώσουμε βέβαια ότι όπως είναι γραμμένος ο κώδικας πρέπει number>=1 διαφορετικά θα λαμβάνουμε ως αποτέλεσμα το 1-αρχική τιμή της result- επ' αόριστον. Για αυτό, όπως σε όλες τις γλώσσες προγραμματισμού πρέπει να φροντίσουμε να ισχύσει κάποτε η συνθήκη τερματισμού του βρόχου.

until

Πρόκειται για άλλον έναν τρόπο δημιουργίας loops. Η γενική μορφή σύνταξης είναι:

```
until  συνθήκη
        ....
end
```

Δηλαδή έως ότου να ικανοποιηθεί η συνθήκη ο κώδικας μέσα στο block (until – end) θα εκτελείται. Φυσικά πρέπει να φροντίσουμε ώστε η συνθήκη κάποτε να ικανοποιηθεί για να μην έχουμε ατέρμονα βρόχο.

Δείτε για παράδειγμα τον παρακάτω κώδικα (αρχείο until.rb):

```
number = 5
result = 1
until number==1
        result*=number
        number-=1
end
puts result
=>120
```

Όπως είναι φανερό μπορούμε να αντικαταστήσουμε το while number!=1 με το until number==1 χωρίς να αλλάξει το αποτέλεσμα. Το ποια εντολή θα χρησιμοποιήσουμε είναι καθαρά θέμα του τι θέλουμε να κάνουμε αλλά και θέμα

ευκολίας.

Θα ήθελα να σας μιλήσω και για την εντολή yield αλλά πριν θα ήταν καλό να μιλήσουμε για το πως φτιάχνουμε συναρτήσεις στη Ruby.

Δημιουργία συναρτήσεων

Για όσους έχετε ασχοληθεί με Python, μία γλώσσα που προσωπικά θεωρώ αρκετά αξιόλογη, θα έχετε ήδη διαπιστώσει πως η Ruby έχει αρκετά κοινά στοιχεία με την Python (ευτυχώς για μένα δεν κράτησε τον πονοκέφαλο των κενών). Άλλωστε το είχαμε πει και στο εισαγωγικό μάθημα η Ruby κράτησε τα καλύτερα στοιχεία πολλών γλωσσών όπως π.χ. Python, Small Talk κ.ά.. Έτσι, λοιπόν, εάν γνωρίζετε πως να ορίζετε συναρτήσεις στην Python δε θα δείτε κάποια διαφορά., καθώς χρησιμοποιείται και εδώ η δομή def-end.

Π.χ. έστω ότι θέλουμε να γράψουμε τη συνάρτηση min. Βέβαια, θα μπορούσαμε να το κάνουμε με τη βοήθεια του τελεστή ?: σε μία γραμμή (αρχείο min.rb):

```
a=2
b=3
puts (a<b) ? a : b
=>2
```

Για να πειραματιστούμε όμως, θα την κάνουμε και συνάρτηση. Έτσι έχουμε (αρχείο min_fun):

```
def minimum a,b
        (a<b) ? a : b# μπορούμε να γράψουμε και return (a<b) ? a : b
end

puts minimum(2,3)
=>2
```

Καταρχήν παρατηρείστε ότι στο ίδιο αρχείο που ορίζουμε τη συνάρτηση μπορούμε να κάνουμε και την κλήση της συνάρτησης. Ακόμη, δεν δηλώσαμε τον επιστρεφόμενο τύπο (π.χ. void, int, string) όπως γίνεται σε άλλες γλώσσες. Επίσης, παρατηρήστε ότι στη Ruby υπάρχει η εντολή return την οποία μπορείτε να χρησιμοποιήσετε κατά τα γνωστά για να σας επιστρέψει το αποτέλεσμα τις συνάρτησης · αλλά στην προκειμένη περίπτωση εσκεμμένα δε χρησιμοποιήθηκε. Και αυτό έγινε για να έχετε υπόψη σας ότι η Ruby εάν δεν δει return μέσα σε μία συνάρτηση επιστρέφει το αποτέλεσμα της τελευταίας εντολής εντός της συνάρτησης που εκτελέστηκε. Και σε περίπτωση που δεν υπάρχει κάποιο αποτέλεσμα να επιστρέψει τότε θα λάβετε nil. Αν το ξεχάσετε μπορεί να βρεθείτε σε δυσάρεστες καταστάσεις ή να ξοδέψετε πολύτιμο χρόνο προσπαθώντας να καταλάβετε γιατί συμβαίνει αυτό.

yield

Μία εντολή της Ruby που ξενίζει και παίδεψε ομολογουμένως και τον γράφων μέχρι να εξοικειωθεί μαζί της, είναι η yield. Αξίζει να ασχοληθείτε μαζί της καθώς κατανοώντας την μπορείτε να κάνετε σύντομα και κομψά αυτό που θέλετε. Για να το

πετύχετε αυτό θα πρέπει να πειραματιστείτε.

Για τη yield έχετε υπόψη σας πως όταν τη χρησιμοποιείτε λαμβάνει ως είσοδο ένα κομμάτι κώδικα και το εκτελεί. Το πιο χαρακτηριστικό παράδειγμα που δίνεται συχνά για να γίνει αυτό κατανοητό είναι το παρακάτω (αρχείο yield.rb):

```
def do3times
  yield
  yield
  yield
 end

do3times { puts "yielding..." }#δοκιμάστε να αλλάξετε το κείμενο αυτό
yield puts 4+5#δοκιμάστε να αλλάξετε το κείμενο αυτό
=>
yielding...
yielding...
yielding...
9
```

Δηλαδή, στέλνοντας ως είσοδο στη συνάρτηση do3times το κομμάτι κώδικα «puts "yielding..."» αυτό το δέχονται οι 3 yield εντός της do3times ως είσοδο και το εκτελούν. Ομοίως και στην τελευταία εντολή η yield δέχεται ως είσοδο το κομμάτι κώδικα «puts 4+5» και το εκτελεί.

Στο βιβλίο http://pragprog.com/book/ruby/programming-ruby θα βρείτε ένα πολύ όμορφο παράδειγμα για τη yield που επίσης δίνεται συχνά για να εξηγηθεί η yield. Πρόκειται για τον υπολογισμό της ακολουθίας Fibonacci. Η ακολουθία δηλαδή όπου κάθε όρος είναι το άθροισμα των δύο προηγούμενων ξεκινώντας από το ένα. Δηλαδή οι αριθμοί 1,1,2,3,5,8,13,21... Χάρη στη yield αυτή η ακολουθία υπολογίζεται πολύ εύκολα (αρχείο yield_fibonacci):

```
def fibonacci(till_number)
        i1, i2 = 1,1 #παράλληλη ανάθεση τιμών (i1=1 και i2=1)
        while i1<till_number
                yield i1
                i1, i2 = i2, i1+i2
        end
end
fibonacci(1000) {|f| print f, " "}#εάν δοκιμάστε να καλέστε απλώς fibonacci(1000)
                                  # θα πάρετε μήνυμα λάθους
                                  #yield_fibonacci.rb:5:in `fibonacci': no block given #(yield)
                                  (LocalJumpError)
                                  #from yield_fibonacci.rb:9:in `<main>'
=>1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Καταρχήν να εξηγήσουμε τα απλά, δηλαδή την παράλληλη ανάθεση, η οποία μας βοηθάει να αρχικοποιήσουμε τα i1 , i2=1,1 με μία εντολή αλλά και να υπολογίσουμε τον επόμενο όρο με την εντολή i1 , i2 = i2, i1+i2. Ένα άλλο απλό πραγματάκι που πρέπει να επισημάνουμε είναι ότι λόγω της yield εάν γράφαμε απλώς fibonacci(1000) θα παίρναμε το μήνυμα λάθους που αναγράφετε στα σχόλια, δηλαδή η yield περιμένει κάτι

ως είσοδο και δεν το λαμβάνει. Στη συγκεκριμένη περίπτωση η εντολή yield i1 επειδή είπαμε {|f| print f, " "} εμφανίζει στην κονσόλα τα αποτελέσματα, δηλαδή την ακολουθία Fibonacci μέχρι τον αριθμό 1000.

Επίλογος

Κάπου εδώ θα σας αφήσω και πάλι για να πειραματιστείτε. Εάν ξέρετε ήδη κάποια γλώσσα προγραμματισμού θα είδατε πως οι βασικές εντολές είναι ίδιες, όπως άλλωστε είναι αναμενόμενο. Από εκεί και πέρα υπάρχουν και κάποια πρόσθετα στοιχεία όπως η πολύ ενδιαφέρουσα yield.

Μέχρι την επόμενη φορά καλό πειραματισμό!