Ruby

Ruby weboldalak

- http://www.ruby-lang.org
  - a nyelv honlapja angol és japán nyelven
- http://www.rubycentral.com
- http://www.ruby-doc.org
  - dokumentációs központ
- http://raa.ruby-lang.org
  - Ruby Application Archive
  - scriptek és könyvtárak

Ruby történelem

- Yukihiro “matz” Matsumoto
  - Japán
- Matz 1993 szeptemberében kezdett el foglalkozni a Ruby-val
- 1995 decemberében került kiadásra a 0.95 verzió
  - számos levelezőlista és weboldal foglalkozik vele azóta
- 2000: 1.6 verzió
  - ehhez készült el a Programming Ruby
- 2003: 1.8 verzió
  - legfrissebb: 1.8.2 (2004. augusztus 29.)

Ruby tulajdonságai

- megfontoltan tervezett nyelv
  - Perl, Smalltalk
  - Eiffel
  - CLU
- scriptnyelv
  - dinamikus típusrendszer
- automatikus memóriakezelés (szemétgyűjtés)
- sztringek és reguláris kifejezések rugalmas kezelése
- a változókat nem kell deklarálni
- kivételek
- szálak

Ruby tulajdonságai

- objektum-orientált
  - minden objektum
  - öröklődés, metódusok
  - hozzáférésszabályozás
    - public, protected, private
  - mixin modulok
  - iterátorok, closures
- alapvetően objektum-orientált, de használható úgy is, mint strukturális nyelv
  - Perl, Python: alapvetően strukturális, de használható objektum-orientáltan is
  - Java: csak OO
- a nem definiált érték neve 'nil'

Ruby példák

- print “Hello world!\n”
  - scriptben
  - ruby -e
- puts “Hello world”
- #!/usr/bin/ruby -w

  puts “Hello world”
- hello.rb

Ruby példák

- n!
- def fact(n)
  ```
  if n == 0
      return 1
  else
      return n * fact(n - 1)
  end
  ```
  end
- puts fact(10)
- érdekessége, hogy csak a memória korlátozza azt, hogy minek a faktoriálisa számolható ki

Lexikális elemek

- kis és nagybetűket megkülönbözteti
- megjegyzéseket a # jellel tehetek
  - a sor hátralevő része megjegyzés

Lexikális elemek

- az utasításokat (kifejezéseket) egymástól ; vagy újsor karakterek választják el
  - ; -t nem kötelező kitenni
  - újsor karakter csak akkor nem választ el, ha a kifejezésnek még nyilván nincs vége
    - "Current time: "
      + Time.now.to_s
      - hibás!
    - "Current time: "\
      + Time.now.to_s
      - helyes
    - "Current time: " +
      Time.now.to_s
      - helyes

Azonosítók

- betűvel, vagy aláhúzásjellel kezdődnek
- betűvel, számmal vagy aláhúzásjellel folytatódik
- az elején szerepelhet
  - @
  - @@
  - $
  - csak változók esetében
- végén szerepelhet
  - ?
  - !
  - csak metódusnevek esetében

Azonosítók

- $
  - globális változók
  - $stderr, $stdout
- @@
  - egy osztály változói
- @
  - példányváltozók
- egyéb változók lokális változók
- ?
  - tipikusan logikai függvények
  - stack.empty?

Azonosítók

- !
  - veszélyes függvények jelölésére használható figyelemfelkeltésként
  - stack.empty!
  - string.strip
    - nem változtatja meg a stringet
  - string.strip!
    - megváltoztatja a stringet
- nagybetűvel kezdődő azonosítók konstansok
  - osztályok nevét mindig nagybetűvel kezdjük

Ruby változók

- deklaráció nem szükséges
  - az első rájuk vonatkozó értékadással jönnek létre
  - még nem létező változókra való hivatkozás hibaüzenethez vezet
- változóknak nincs típusa
- Ruby-ban minden objektum
  - a nyelv beépített típusai is osztály-hierarchiába szerveződnek
    - Numeric
      - Integer
      - Float
  - osztályok is objektumok (Class osztály példányai)

Ruby változók

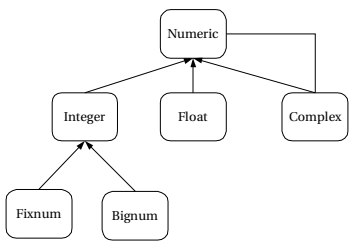
- a változók referenciákat tartalmaznak
  - immediate típusokból egy példány van
    - numerikus típusok
  - más nyelvekben esetleg immutable
- a = b = []
  - a és b változók ugyanarra a tömbre hivatkoznak
- a = []
  b = []
  - a és b változók nem ugyanarra a tömbre hivatkoznak
- a = 1
  b = 1
  - a és b változók ugyanarra az egészre hivatkoznak

Ruby változók

- a paraméterül átadott nem immediate típusok megváltoztathatók
- def modify(aList)
  aList << 1
  end
  list = []
  modify(list) # list megváltozik

Alapvető típusok

Numerikus típusok



Numerikus típusok

- Numeric: a numerikus típusok alaptípusa
- Float: natív dupla pontosságú lebegőpontos számok
- Integer: egész típusok bázisosztálya
- Fixnum: konkrét értékhatárral rendelkező egész típus
- Bignum: tetszőlegesen nagy egész tárolására használható
- Complex: komplex számok

Numerikus típusok

- tetszőlegesen nagy egész számok kezelhetők
- deklaráláskor nem kell megadnunk a típust
  - nem is adhatjuk meg, nincs new metódusa a Bignum és Fixnum osztályoknak
- a numerikus érték nagyságától függ a típus
  - platformfüggő, hogy mi a határ
- nincsenek előjel nélküli egész típusok
- a konverzió Fixnum és Bignum között automatikus
  - mindkét irányban

Numerikus típusok

- Float
  - ceil
  - floor
  - round
  - finite?
  - infinite?
- natív dupla pontosságú

Numerikus literálok

- 231847
- 12381.123
- .42E+2
- 1_000_000
- 0xffff
- 0xdead_beef
- 0b10001001
- 0644
- 0b110_100_100
- ezek mind objektumok már
  - 0.succ # 1

Sztringek

- String osztály példányai
- jelen van a Perl-ből ismert változóhelyettesítés
  - double-quoted strings
  - single-quoted strings
- str = '12'
  - str.class: String
  - str + 3
    - hibás, mert különböző típusúak
    - nincs automatikus konverzió számok és sztringek között
    - str.to_i + 3 # 15
    - str + 3.to_s # “123”
- kiíratás
  - print
  - puts

Sztringliterálok

- hasonló lehetőségek, mint Perl-ben
- ''
  - single-quoted, azaz nincs helyettesítés
  - \', \\
  - megadható több soros sztring
- “”
  - double-quoted
  - megadható több soros sztring
  - \n, \t, stb.
  - helyettesítés #{} konstrukcióval
    - “Hello #{name}”
  - tetszőleges kifejezés lehet
    - “Egy nap #{24 * 60 * 60} másodperc.”

Sztringliterálok

- a {} elhagyható globális-, osztály- és példányváltozók esetében
  - a # továbbra is kötelező!
  - “Script name: #$0”
  - “Value: #@@class_var”
  - “Value: #@instance_var”
- %q{}
  - single-quoted
  - elhatároló megválasztható
    - pl: !, #, |, /, {}, [], (), <>
- %Q{}, %{}
  - double-quoted
  - elhatároló megválasztható
    - pl: !, #, |, /, {}, [], (), <>

Sztring műveletek

- “hello”.length
  - 5
- “9z”.succ
  - “10a”
- “ hello “.strip
  - “hello”
- “0x0f”.hex
  - 15
- “011”.oct
  - 9

Sztring műveletek

- “árvíztűrő tükörfúrógép”.tr(“áéíóöőúüű“, “aeioooouuu”)
  - “arvizturo tukorfurogep”
- “password”.crypt(“sa”)
  - "sa3tHJ3/KuYvI"

Sztringliterálok

- HERE dokumentumok
- s = <<EOS
  ...
  EOS
  - double-quoted
- s = <<'EOS'
  ...
  EOS
  - single-quoted
- s = <<”EOS”
  ...
  EOS
  - double-quoted

Sztringliterálok

- s = <<-EOS
  ...
  EOS
  - - jel miatt az EOS behúzható (indentálható)

Sztringliterálok

- sztring literálok minden egyes használata külön objektum létrehozását eredményezi
  - nemcsak lexikálisan, hanem dinamikusan is
  - ciklusokban
  - paraméterátadáskor
  - értékadáskor
- for 1..3 do
  puts 'hello'.object_id
  end
  - 538380596
    538380576
    538380556

Sztring műveletek

- “foo”.upcase
  - “FOO”
- “foo”.capitalize
  - “Foo”
- “BAR”.downcase
  - “bar”
- “Bar”.swapcase
  - “bAR”

Sztring műveletek

- “foo bar”.split
  - [“foo”, “bar”]
- “1974. 10. 21.”.scan(/\d+/)
  - [“1974”, “10”, “21”]
- “1974. 10. 21.”.scan(/\d+/).collect { |s| s.to_i }
  - [1974, 10, 21]
- “1974. 10. 21.”.scan(/(\d+)/)
  - [[“1974”], [“10”], [“21”]]
- “Year: 1974. Month: 10. Day: 21.”.
  scan(/(\w+):\s*(\d+)/)
  - [[“Year”, “1974”], [“Month”, “10”], [“Day”, “21”]]

Sztring műveletek

- További műveletek a dokumentációban
- a fenti műveletek egy új sztringet adnak vissza
  - nem módosítanak
- a hasonló nevű !-re végződők helyben módosítják a sztringet
  - upcase -> upcase!
  - downcase -> downcase!
  - tr -> tr!
- ez mind külön metódus
  - nem automatizált mechanizmus
- nem minden metódusnak van ilyen megfelelője
  - pl. crypt! nem létezik

Range

- Range osztály
- egy intervallumot reprezentál
  - ..
  - ...
- nem tömb, mint Perl-ben
  - (0..5).to_a
    - [0, 1, 2, 3, 4, 5]
  - (0...5).to_a
    - [0, 1, 2, 3, 4]
- 'a'..'z'
- tetszőleges osztály példányai lehetnek egy Range határai
  - legyen egy succ metódus
  - legyen <=> metódus

Range

- Feltételként is használhatóak
- while gets
    print if /start/../end/
  end
  - Ruby 1.8 ezt már nem támogatja
    - warning
    - nem működik!
  - while gets
      print if $_ =~ /start/ .. $_ =~ /end/
    end
- a feltétel igaz lesz, ha a Range kezdeti kifejezése igazat ad
- a feltétel addig marad igaz, amíg a második hamis

Range

- Intervallumként is használhatóak
- eldönthető, hogy egy elemet tartalmaz-e az intervallum
  - nincs köze a succ függvényhez
  - <=>
- ===
  - case összehasonlító operátor
- (1..10) === 15 # false
- (1..10) === 3   # true
- (1..10) === 3.14159 # true

Tömbök

- Array osztály példányai
- [] operátor
  a = ['foo', 'bar']

  a = [
    'foo',
    'bar',
    'baz',
  ]
- utolsó vessző nem hiba
- mérete nem fix, változhat

Tömbök

- Array.new
  - konstruál egy tömböt
  - []
- Array.new(3)
  - [nil, nil, nil]
- Array.new(3, 0)
  - [0, 0, 0]
- a = Array.new(3, [])
  - ugyanarra az objektumra fognak mutatni
  - a[0] << 0
  - [[0], [0], [0]]

Tömbök

- a = Array.new(3, 'foo')
  - ['foo', 'foo', 'foo']
  - mindhárom referencia ugyanarra a String objektumra mutat
- a[0].upcase!
  - ['FOO', 'FOO', 'FOO']
- DE:
  - a[0] += ' bar'
    - ['foo bar', 'foo', 'foo']
    - String#+ az összefűzés során új objektumot hoz létre

Tömbök

- szintén van szavakból tömböt létrehozó operátor
- %w(foo bar)
  - ['foo', 'bar']
- minden whitespace elválasztja a szavakat
- szónak minősül minden, ami nem-whitespace karakterek sorozata
  - \S+

Tömbök

- ez elemek indexelése 0-val kezdődik
- az adott elem elérésére a [] operátor használható
- rengeteg művelettel rendelkezik
  - ezek közül néhány a Perl-ből gyakran hiányzik
  - halmazként is használható

Tömbelemek elérése

- a = %w(a b c d e f)
  - [“a”, “b”, “c”, “d”, “e”, “f”]
  - a[0] # “a”
  - a[-1] # “f”
  - a[1, 3] # [“b”, “c”, “d”]
    - kezdet és hossz
  - a[0..3] # [“a”, “b”, “c”, “d”]
    - Range objektum a paraméter
- a[0, 2, 5]
  - nem támogatott
- a.at(1) # “b”
  - az elem gyorsabb elérése
  - nincs lehetőség Range, stb. paraméterek megadására

Tömbelemek módosítása

- a = %w(a b c d e f)
  - a[0] = “A”
  - a[-2] = “E”
  - a[8] = “???”
    - [“a”, “b”, “c”, “d”, “e”, “f”, nil, nil, “???”]
    - a tömb automatikusan növekszik
  - a[-10]
    - IndexError
- a[start, length] = ...
  - elemek cseréje
  - a[1, 2] = [“B”, “E”, “C”, “E”]
    - [“a”, “B”, “E”, “C”, “E”, “d”, “e”, “f”]
  - a[1, 0] = [“B”, “E”, “C”, “E”]
    - [“a”, “B”, “E”, “C”, “E”, “b”, “c”, “d”, “e”, “f”]
    - a hossz 0 => beszúrás

Tömbelemek módosítása

- Range esetében hasonló módon a meghatározott részlista módosul
- a = %w(a b c d e f)
  - a[1..3] = []
    - [“a”, “e”, “f”]
  - a[1..0] = [“X”, “Y”]
    - [“a”, “X”, “Y”, “b”, “c”, “d”, “e”, “f”]
    - 0 hosszúságú => beszúrás

Műveletek tömbökkel

- veremként kezelve
  - a << 1
  - a << 2 << 3 << 5
  - a.push(1)
  - a.push(2, 3, 5)
  - a.pop
    - visszatér a verem tetején (tömb végén) levő elemmel
    - a tömb módosul
  - a.last
    - a verem teteje (a tömb utolsó eleme)

Műveletek tömbökkel

- sorként kezelve
- push/shift
  - a << 1
    - a.push(1)
  - a.shift
    - kiveszi a sor elejéről az elemet
    - a tömb módosul
- unshift/pop
  - a.unshift(1)
  - a.pop

Műveletek tömbökkel

- halmaz jellegű műveletek
- a = [1, 2, 3, 3, 3, 4, 7, 2]
  - a.uniq # [1, 2, 3, 4, 7]
    - a.uniq!
  - a.include?
    - a.include?(5) # false
    - a.include?(3) # true
- metszet: &
  - [1, 2, 2, 3] & [1, 1, 2, 4]
    - [1, 2]
    - a duplikációkat kiszűri

Műveletek tömbökkel

- únió: |
  - [1, 2, 2, 3] | [1, 1, 2, 4]
    - [1, 2, 3, 4]
    - a duplikációkat kiszűri
- különbség
  - [1, 2, 2, 3] - [1, 1, 2, 4]
    - [3]
    - a duplikációkat kiszűri

Műveletek tömbökkel

- [1, 2, nil, 4, nil].compact
  - [1, 2, 4]
  - kiszűri a nil elemeket
- [4, 2, 3].sort
  - [2, 3, 4]
  - <=> operátort használ az elemek összehasonlítására
  - [“b”, “a”, 1]
    - hiba!
- %w(foo bar baz).join(', ')
  - “foo, bar, baz”
- hossz
  - a.length
  - a.size

Műveletek tömbökkel

- [1, 2] * 3
  - multiplikáció
  - [1, 2, 1, 2, 1, 2]
- [1, 2, 3] + [4, 5, 6]
  - új tömböt hoz létre
  - összefűz
  - [1, 2, 3, 4, 5, 6]
- [1, 2, 3].concat([4, 5, 6])
  - nem hoz létre új tömböt
- lapítás
  - [1, 2, [3, 4, 5], [6, 7, [8, 9]]].flatten
    - [1, 2, 3, 4, 5, 6, 7, 8, 9]
  - rekurzívan

Műveletek tömbökkel

- a.replace([10, 20, 30])
  - megváltoztatja az 'a' által hivatkozott tömböt
  - a = [10, 20, 30]
    - új tömböt hoz létre, és az 'a' erre fog mutatni
- a.clear()
  - a tömb tartalmának törlése
  - a = []
    - új tömböt hoz létre, és az 'a' erre fog mutatni

Hash

- Hash osztály példányai
- új hash létrehozása
  - {}
  - h = {
    'red' => 0xff0000,
    'green' => 0x00ff00,
    'blue' => 0x0000ff,
    }
- utolsó vessző itt sem okoz gondot

Hash

- Hash.new(anObject)
  - default érték, amennyiben egy lookup során a keresett kulcs nem szerepel a hash-ben
- Hash.new(0)
  - számlálás esetén hasznos
  - hash[key] += 1
    - nem jelez hibát, ha a key-t még nem számoltuk
- Hash.new([])
  - minden esetben ugyanazt a tömböt fogja adni!

Hash műveletek

- hash mérete
  - a benne levő párok száma
  - h.size
  - h.length
- h.empty?
- h.keys
  - egy tömb az összes kulccsal
- h.values
  - egy tömb az összes értékkel
- h.has_key?(key)
  - h.keys.include?(key)
- h.has_value?(value)
  - h.values.include?(value)

Hash műveletek

- h = { 'a' => 10, 'b' => 11 }
- h.invert
  - {10 => 'a', 11 => 'b'}
- ha több azonos érték szerepel, akkor csak az egyik marad
  - a belső tárolás alapján meghatározott
- { 'a' => 10, 'b' => 10 }.invert
  - {10 => 'b' }

Hash műveletek

- h = { 'a' => 10, 'b' => 11 }
- h.fetch
  - megadható egy default érték, ha a kulcsot nem találja a hash-ben
  - h.fetch(“no_such_key”, 0)
- h.index(10)
  - 'a'
  - megadja az értékhez tartozó kulcsot
  - ha nincs ilyen érték, akkor nil
- h.indexes
  - a felsorolt kulcsokhoz tartozó értékek listája
  - ha egy kulcs nincs a hash-ben, akkor a default

Hash műveletek

- update
  - hash módosítása
  - h1 = { “a” => 1, “b” => 2, “c” => 3 }
    h2 = { “b” => 4, “d” => 5 }
    h1.update(h2)
    - { “a” => 1, “b” => 4, “c” => 3, “d” => 5 }
- h.replace(aHash)
  - a h objektum módosítása
- h.sort
  - a kulcsok alapján rendezi a hash-t, és párok tömbjét adja vissza
  - h = {“a” => 1, “b” => 2, “c” => 3}
    h.sort
    - [[“a”, 1], [“b”, 2], [“c”, 3]]

Halmazok

- nem alaptípus
  - require 'set'
- Set osztály példányai
  - rendezetlen értékek duplikációk nélkül
  - Hash osztály használja belső tárolásra
- bármely Enumerable objektum használható konstruálásra
  - iterálhatóak legyenek az elemek
- minden Enumerable objektum halmazzá alakítható
  - to_set metódus
  - [1, 2, 3].to_set

Halmazok

- s << element
  - új elem hozzáadása a halmazhoz
- s.size
  - a halmaz elemszáma
- s.empty?
  - igaz, ha s üres
- s.include?
  - tartalmazásvizsgálat
- s1.subset?(s2)
  - igaz, ha s1 részhalmaza s2-nek
- s1.proper_subset?(s2)
  - igaz, ha s1 valódi részhalmaza s2-nek
- s1.superset?(s2)
- s1.proper_superset?(s2)

Halmazok

- s1 & s2
  - halmazok metszete
  - s1.intersection(s2)
  - új halmazt hoz létre
  - [1, 2, 3].to_set & [2, 3, 4].to_set
    - #<Set: {2, 3}>
- s1 | s2
  - halmazok uniója
  - s1 + s2
  - s1.union(s2)
  - visszatérési értéke egy új halmaz
  - [1,2].to_set | [2, 3].to_set
    - #<Set: {1,2,3}>

Halmazok

- s1.merge(s2)
  - únió
  - nem hoz létre új halmazt
- s1.delete(o)
  - elem kivétele a halmazból
- s1.subtract(enum)
  - elemek kivétele a halmazból
- s1 ^ s2
  - kizáró vagy
  - (s1 | s2) – (s1 & s2)
  - (1..3).to_set ^ (2..5).to_set
    - #<Set: {5, 1, 4}>

Blokkok

- alapvető nyelvi konstrukció a Ruby-ban
- a szokásostól eltérő fogalom
  - elágazások, ciklusok, metódusok esetén másról van szó
- csoportosíthatók vele utasítások
- iterátorok implementálhatók ezzel az eszközzel
  - nem kell külön iterátor osztályt definiálni
  - egyszerűen kezelhető
  - iterátor: olyan metódus, amely fel van készülve arra, hogy blokkot kap
- számos beépített osztálynak van iterátor metódusa
  - gyakran 'each'

Blokkok

- object.iterator_method { |param|
  ...
  }
- object.iterator_method do |param|
  ...
  end
- az iterátor metódus adhat paramétereket a blokknak
- anArray.each { |element|
  puts element
  }
  - egy tömb elemein iterál

Blokkok

- anArray.reverse_each { |element| ... }
  - egy tömb elemei visszafele
- aHash.each_key { |key| ... }
  - egy hash kulcsain iterál
- anInteger.times { ... }
  - adott számszor meghívja a blokkot
- anInteger.step(10, 2) { |i| ... }
  - léptetés anInteger-től 10-ig 2-es lépésközzel
- aRange.each { |i| ... }
  - az intervallum elemein iterál
- stb.

Blokkok

- Set#classify { |element| ... }
- egy halmaz elemei osztályozhatók vele
- minden elemre kiértékeli a blokkot
- a blokk visszatérési értéke alapján osztályoz
- az eredmény egy hash
  - kulcsok a blokk visszatérési értékei
  - értékek halmazok, melyek az eredeti részhalmazai

Kifejezések

Kifejezések

- Ruby-ban szinte minden kifejezés
  - értékadás
  - elágazások
- tiszta szemantika
- TIMTOWTDI

Értékadás

- =
  - értékadás operátor
- kifejezés, értéke az értékül adott érték
- a = b
  - változó értékadás
  - referencia másolás
- hifi.volume = 10
  - metódushívás
  - Hifi#volume=
    - ha hifi.is_a?(Hifi)

Értékadás

- >1 balérték
- a, b = b, a
  - egyszerű csere
  - tömbök segítségével
- a, b, c = d, e
  - a == d, b == e
  - c == nil
- a, b = c, d, e
  - a == c, b == d
- szimultán értékadás esetén
  - a kifejezés visszatérési értéke egy tömb az értékadás jobb oldalán álló kifejezésekkel
  - a = (b, c = 1, 2)
    - [1, 2]

Értékadás

- az értékadás jobb oldalán tömb is állhat
  - ha a jobb oldalon csak ez van, de a bal oldalon több változó áll, akkor a tömb objektumok sorozatára bomlik
    - expanding
  - ugyanez történik, ha egy '*' -t kerül a tömb elé
- a, b, c = [1, 2, 3]
  - a: 1, b: 2, c: 3
- array = [1, 2, 3]
  a, b, c = array
- a, b, c = (1..3).to_a
  - kifejezésekre is igaz
  - a, b, c = 1..3
    - a: 1..3, b: nil, c: nil

Értékadás

- array = %w(B C D E)
- a, b, c, d, e = “A”, array
  - a: “A”, b: array
  - c: nil, d: nil, e: nil
- a, b, c, d, e = “A”, *array
  - a: “A”, b: “B”, c: “C”, d: “D”, e: “E”
- a, b, c, d, e = “A”, *(“B”..”E”).to_a
  - kifejezés is expandolható
  - metódus visszatérési értéke is

Értékadás

- értékadás baloldán levő tömb előtt a * összegyűjti
  - csak az értékadás baloldalának utolsó változója előtt lehet
    - egyéb esetben szintaktikai hiba
- n = (1..5).to_a
- a, b = n
  - a: 1, b: 2
- a, *b = n
  - a: 1
  - b: [2, 3, 4, 5]
- a, *b = *n
  - ugyanaz, mint fent

Értékadás

- a, *b = 1, n
  - a: 1, b: [[1, 2, 3, 4, 5]]
- a, *b = 1, *n
  - a: 1, b: [1, 2, 3, 4, 5]
- a, aa, *b = 1, *n
  - a: 1, aa: 1, b: [2, 3, 4, 5]
- a, b = 1, (x, y = 0, 0)
  - a: 1, b: [0, 0]
- a, *b = 1, 2, (x, y = 0, 0)
  - a: 1, b: [2, [0, 0]]
- a, b = 1, *(x, y = 0, 0)
  - a: 1, b: 0

Értékadás

- beágyazott értékadások
  - értékadás baloldalán
  - a soron következő elemet használja a belső értékadásban
- a, (b, c), d = 1, 2, 3, 4
  - a: 1, b: 2, c: nil, d: 3
- a, (b, c), d = 1, [2, 3], 4
  - a: 1, b: 2, c: 3, d: 4
- a, (b, c), d = 1, [2, 3, 4], 5
  - a: 1, b: 2, c: 3, d: 5
- a, (b, *c), d = 1, [2, 3, 4], 5
  - a: 1, b: 2, c: [3, 4], d: 5

Operátorok

- Ruby-ban majdnem minden operátor egy metódushívást indukál
- a + b * c
  - a.+(b.*(c))
- a * b + c
  - figyelembe veszi a precedencia szabályokat
  - (a.*(b)).+(c)
- a = a + b
  - a += b
  - belül a fenti formára alakul
  - 'a' osztályában egy + metódust kell definiálni
- operátorok felüldefiniálhatóak

Operátorok

- ::
- []
- **
- -(unary) +(unary) ! ~
- * / %
- + -
- << >>
- &
- | ^
- > >= < <=
- <=> == === != =~ !~

Operátorok

- &&
- ||
- .. ...
- = += -=...
- not
- and or
- nem definiálható felül
  = ... ! not && and || or != !~
  rövidített operátorok
  += -=

Logikai kifejezések

- minden kifejezésnek van igazságértéke
  - nil: hamis
  - false: hamis
  - minden más: igaz
- nil
  - NilClass egyetlen példánya
- false
  - FalseClass egyetlen példánya
- true
  - TrueClass egyetlen példánya
- '' igaz
  - String osztály hétköznapi példánya
- 0 igaz
  - Fixnum osztály hétköznapi példánya

Logikai kifejezések

- and, or, not
  - szokásos operátorok
  - Perlben megszokott szintaxis és szemantika él
  - lusta kiértékelés
  - gets or puts “No more lines”
- &&, ||, !
  - ekvivalens az and, or, not operátorokkal
  - ezek precedenciájuk magasabb

Logikai kifejezések

- ==
  - szokásos egyenlőségvizsgálat operátor
- ===
  - case egyenlőségvizsgálat
- <=>
  - “spaceship operator”
  - általános összehasonlító operátor
- <, <=, >, >=
  - összehasonlító operátorok
- =~
  - mintaillesztés operátora

Logikai kifejezések

- eql?
  - igazat ad, ha a két operandus (fogadó és argumentum) típusa megegyezik, és az értékük egyenlő
  - fogadó (receiver): aki az üzenetet kapja
  - 1.eql?(1.0)
    - false
  - 1 == 1.0
    - true
- equal?
  - igazat ad, ha a fogadó és az argumentum objektum azonosítója (object id) megegyezik

Logikai kifejezések

- !=
  - == operátorból származik
  - Ruby parse-olás során cserél:
    - a != b kifejezéseket !(a==b) alakúra
- !~
  - =~
  - Ruby parse-olás során cserél:
    - a !~ b kifejezéseket !(a=~b) alakúra
- ==, =~ felüldefiniálható
  - azonnal használhatóak a !=, !~ operátorok
  - nem függetlenek az operátorok (!= nem definiálható)

Logikai kifejezések

- if expr_1 .. expr_2
  ...
  end
  - flip/flop
  - igazzá válik a feltétel, amikor az első kifejezés igaz lesz
  - addig marad igaz, amíg a második feltétel igaz nem lesz
- if ($s =~ /begin/) .. ($s =~ /end/)
  ...
  end

defined?

- defined? operátor
  - nil, ha az argumentum nem definiált
  - egyébként az argumentum leírása
    - defined? no_such_method# nil
    - defined? 1 # “expression”
    - defined? gets # “method”
- különbözik a Perl defined operátorától
  - defined? gets
    - nem hívja meg a gets metódust!
- ha azt akarjuk vizsgálni, hogy egy kifejezés eredménye nil-e, vagy sem
  - nil isa NilClass
    - Object#nil?
    - nil.nil? # true
    - 0.nil? # false

``

- backtick operátor
- ahogy Perlben, itt is az operációs rendszerbe hív
  - visszatérési értéke a program standard kimenete
    - String típusú
- `date`.match(/^(Sat|Sun)/) ?
  "Weekend" : "Weekday"
  - “Weekend”
- $? változóban a program exit kódja
  - $CHILD_STATUS in English
  - $CHILD_STATUS isa Process::Status
    - Ruby 1.8 óta

Process::Status

- $CHILD_STATUS.pid
  - process ID
- $CHILD_STATUS.exited?
  - igaz, ha a program már kilépett
- $CHILD_STATUS.exitcode
  - a program kilépési kódja
- stb.
- `date`
  p $CHILD_STATUS
  - #<Process::Status: pid=16598,exited(0)>

Vezérlési szerkezetek

Elágazások

- minden elágazás kifejezés, és van értéke
  - utolsó kifejezés értéke
- if expr then
    ...
  elsif expr then
    ...
  else
    ...
  end
- then elhagyható
  - ha egy sorba kerül a feltétel és az utasítás, akkor kötelező
  - if Date.today.leap? then print “Szökőév” end

Elágazások

- unless expr then
    ...
  else
    ...
  end
- elsif
  - nincs rá lehetőség: szintaktikai hiba
  - nem lenne sok értelme
- then elhagyható
  - ha egy sorba kerül a feltétel és az utasítás, akkor kötelező
  - unless Time.now==Time.now then puts “SLOW” end

Elágazások

- case expr
    when case_1 then ...
    when case_2 then ...
    else ...
  end
- then elhagyható
  - akkor kötelező, ha az utasítások a when feltételekkel egy sorba kerülnek
- expr tetszőleges kifejezés lehet
  - case_n is tetszőleges kifejezés lehet
- ha case_n megfelel a kifejezésnek, akkor nem hajtódik végre a többi ág
- 'case_n' megfelel az 'expr' kifejezésnek, ha
  - case_n === expr

Elágazások

- Object#==
  - a Ruby egyenlőségvizsgálat operátora
- Object#===
  - a Ruby case egyenlőségvizsgálat operátora
  - szinonímája az Object#== operátornak
- bizonyos gyerekosztályok felüldefiniálhatják ezt
  - Regexp
  - Range
- case is kifejezés Ruby-ban
  - utolsó kifejezés értéke
- ha egyik when ágra sem illeszkedik, és nincs else ág
  - kifejezés értéke nil
  - nem probléma (nem kell lefedni az állapotteret)

?:

- cond ? expr_1 : expr_2
  - cond feltétel alapján elágaz
  - ha cond igaz, expr_1
  - ha cond hamis, expr_2
- kifejezés
  - a = Time.now.hour < 10 ? “sleep” : “get up”
- nem balérték
  - ellentétben a Perllel

Utasításmódosítók

- ahogy Perlben is
  - expr if condition
  - expr unless condition
- condition értékelődik ki először
  - puts “Label: $1” if /^(\w+):/
- mivel az elágazás is kifejezés, ezért akár:

```
unless total == 0
    $log.puts(“Total: #{total}”)
    $stderr.puts(“Total: #{total}”)
end if options[“verbose”]
```


Ciklusok

- Ruby-nak csak nagyon egyszerű ciklus konstrukciói vannak
- a blokkok miatt nincs szükség kifinomultabb konstrukciókra
  - The Ruby Way
- ciklusok visszatérési értéke mindig nil
  - nem kifejezés

Ciklusok

- while cond do
  ...
  end
- until cond do
  ...
  end
- do elhagyható
  - csak akkor kell, ha egy sorban van a feltétel és a ciklusmag
- while gets
  print “Read: #$_”
  end

Ciklusok

- while, until ciklusoknak van hátultesztelő változata is
- begin
  ...
  end while expr
- begin
  ...
  end until expr

for ciklusok

- Ruby-ban igazából nincs
- for (i=0; i<n; ++i) {}
  - n.times {}
- for (i=0; i<n; ++i) { ...i... }
  - 0.upto(n-1) { |i| ...i... }
- for (i=0; i<n; i += 2) { ... i ... }
  - 0.step(n-1, 2) { |i| ...i...}
- foreach my $element (@array) { .. $element ... }
  - array.each { |element| ... element.... }
  - array.each do |element| ... element ... end
- each: iterátor

for.rb

for ciklusok

- for element in enumerable
  ... element ...
  end
- array.each do |element|
  ... element ...
  end
- mindkettő Ruby utasítás
- szinte ugyanaz a kettő
  - az első esetben a lokális változók láthatósága másképp alakul

loop

- beépített függvény
  - egy egyszerű iterátor
- loop do
  ...
  end
- a blokkot hívja folyamatosan

next, break

- next
  - egy ciklus következő iterációját kezdi meg
  - újra kiértékeli a ciklusfeltételt
  - while line = gets
    next if line =~ /^#/
    ...
    end
- break
  - kiugrás a ciklusból
  - while line = gets
    next if line =~ /^#/
    break if line.chomp == “quit”
    ...
    end

redo.rb

redo

- az iteráció újrakezdése a feltétel újbóli kiértékelése nélkül
- while line = gets
  line.chomp!
  next unless line.length > 1

  puts line.chop!
  redo
  end
- nincs probléma a line változó láthatóságával és élettartamával

Iterátor blokkok

- blokkokban is használható
  - next, break, redo
- first_prime = nil
  (100..200).each do |i|
  if isPrime(i)
  first_prime = i
  break
  end
  end
  - iterátor blokkban nem lehet return

retry.rb

retry

- az egész ciklust újra kezdi
- valódi ciklusokra nem használható
- iterátor blokkokon belül érvényes
- természetesen nem tekeri vissza az időt
  - a módosított változók módosulva maradnak

Változók láthatósága

- elágazások nem nyitnak új scope-ot
  - if, unless, case
  - utasításmódosítók sem
  - val = 1
    if true then val = 2 end
    p val
  - ha a változó még nem létezik, akkor itt jön létre
- ciklusok nem nyitnak új scope-ot
  - while, unless, for
  - val = 1
    while gets do val = 2 end
    p val
  - hátultesztelő ciklusok sem

block-scope.rb
loop-scope.rb

Változók láthatósága

- iterátor blokkokon belül egy változó lokális
- ugyanakkor a blokk környezete is lényeges a futtatás során
  - ha egy változó nem létezik a környezetben, akkor lokális lesz
  - ha létezik, akkor a létező változót használja a Ruby