Bevezetés a Ruby nyelv használatába

Avagy minden, amit tudnunk kell ahhoz, hogy elkezdhessünk programozni Ruby nyelven...

Mi programozók igazán meg lehetünk elégedve azzal a világgal, amelyben ma élhetünk. Ezt azért mondom, mert rengeteg kiváló programozási nyelv közül válogathatunk, és ez az állítás különösen igaz a nyílt forrású szoftverek világára.
Manapság az egyik legtöbbet emlegetett nyelv a ***Ruby*** annak ellenére, hogy igazából nem is annyira új. Kitalálója, ***Yukihiro „Matz" Matsumoto*** az első változatot valójában 1995-ben tette közzé. A nyelv azóta jelentős fejlődésen ment keresztül, és a népszerűsége is fokozatosan nőtt a mai szintre.
Az egyik legnagyobb lökést ennek a folyamatnak kétségkívül a ***Ruby on Rails*** rendszer jelentette, ami egymaga jelentősen megnövelte a ***Ruby*** elterjedtségét és népszerűségét.
A ***Ruby***-t gyakran a ***Perl*** és a ***Smalltalk*** keresztezésének tekintik, és én magam is úgy gondolom, hogy ez a hasonlat nem is olyan rossz. Ennek pedig az az oka, hogy aki sokat programozott ***Perlben***, és ismeri az objektumközpontú fejlesztést is, annak – hiába kezdő – igen könnyű dolga lesz a ***Ruby***-val.
Ebben a cikkben a ***Ruby*** alapjait szeretném bemutatni egyrészt összevetve a többi, magas szintű programnyelvvel, másrészt a hasonlóságok mellett rámutatva azokra a pontokra is, ahol valami különlegeset hozzátesz az eddigi paradigmákhoz. Őszintén remélem, hogy a cikk végére az olvasó rendelkezni fog annyi ismerettel, amely alapján már megpróbálhat megírni egy kisebb alkalmazást ***Ruby***-ban. Ha az Olvasó programozással kapcsolatos beállítottsága hasonló az enyémhez, bizonyára gyorsan fel fogja fedezni, milyen elegáns és kompakt módon lehet könnyen karbantartható kódokat írni ezen a nyelven.

Az alapok

A ***Ruby*** letöltése és telepítése egészen egyszerű, különösen, hogy a legtöbb ***Linux*** terjesztés eleve tartalmazza egy viszonylag friss (1.8.2-es) változatát. Aki nem elégszik meg ezzel, az a nyelv webhelyéről letöltheti a valóban legfrissebb (1.8.4-es) kódot is. Tekintettel arra, hogy nyílt forrású projektről van szó, azon gondolom senki nem fog meglepődni, hogy ezen a webhelyen (➲ ***http://www.ruby-lang.org***) megtalálja a teljes forráskódot is ***.tar.gz*** formátumban. A szokásoknak megfelelően vannak persze közvetlenül telepíthető ***DEB*** és ***RPM*** csomagok is.
Aki forrásból szeretné a ***Ruby***-t telepíteni, annak sincs nehéz dolga. Töltsük le a kódot tartalmazó csomagot, majd bontsuk ki a ***.tar.gz*** állományt:

```
$ cd Downloads
$ tar -zxvf ruby-1.8.4.tar.gz
```

Ezután a szabványos `configure` programmal beállíthatjuk a rendszer felszereltségének megfelelően a fordítási paramétereket, majd a `make` parancs kiadásával elvégezhetjük a fordítást. A ***Ruby*** helyes működését a `make test` paranccsal lehet ellenőrizni:

```
$ ./configure && make && make
➥test
```

Ha minden jól ment, akkor a fenti parancsok kimenetének utolsó sora a `test succeeded` szöveg lesz. Ha valóban ez a helyzet, akkor lépjünk be root-ként, és telepítsük a ***Ruby***-t a rendszerre:

```
$ su
# make install
```

Talán érdemes megjegyezni, hogy ezzel nem csak maga a nyelv, hanem számos apróbb könyvtár és segédprogram is fölkerül a rendszerre.

Az interaktív Ruby: irb

Maga a ***Ruby*** nyelv tulajdonképpen nem több egyetlen `ruby` nevű végrehajtható állománynál, amit szükség esetén természetesen kézzel is futtathatunk a parancssorból:

```
$ ruby
```

Ugyanakkor a ***Ruby***-nak ez a változata csak nem interaktív üzemmódban képes működni. Ez utóbbi pedig nem föltétlen megfelelő, ha mondjuk tesztelni akarunk egy kódot, vagy egyszerűen csak kísérletezünk a rendszerrel.
Nos, erre találták ki az ***irb*** programot, ami egy interaktív ***Ruby*** héj (shell).
Az `irb` gyakorlatilag olyasmi, mint egy nyomkövetőprogram, hiszen meglehetősen hasonlóan működik: felhasználói bemenetre vár (amit az ***Enter*** lenütése zár), majd végrehajtja a begépelt parancsokat. Próbáljuk meg elindítani:

```
$ irb
```

Erre az `irb` a következőt válaszolja:

```
irb(main):001:0>
```

Na, akkor most gépeljünk be valamit ékes ***Ruby*** nyelven:

```
irb(main):001:0> print "Hello,
➥world"
```

Erre a következő választ kapjuk az ***irb***-től:

```
Hello, world=> nil
```

A fenti kimenet azt mutatja, hogy a `print` utasítás kiírja a kimenetre a neki átadott `"Hello, world"` karakterláncot, majd `nil` értékkel tér vissza. A `nil` a ***Ruby***-nál a más nyelvekben szokásos null értéknek felel meg. Ez az, amit ***Perl***-ben `undef`-nek, ***Pythonban*** `None`-nak, ***SQL***-ben `NULL`-nak hívnak.
Mint megannyi más nyelv, a ***Ruby*** is lehetővé teszi, hogy előzetes deklarálás nélkül adjunk értéket egy változónak. Ennek megfelelően a következő forma ***Ruby***-ban helyes:

```
greeting = "Hello, world"
print greeting
```

A ***Ruby*** – nem túl meglepő módon – matematikai műveletekre is képes, vagyis ismeri a `+`, `-`, `*` és `/` műveleti jeleket:

```
5 + 3
60 - 23
60 * 23
10 / 2
```

Ebből a kódból most teljesen kihagytam azt a lépést, ami a fenti műveletek eredményét megjeleníti, mivel a tárgyalás szempontjából lényegtelen. Ugyanakkor azt fontos megjegyezni, hogy a ***Ruby*** magától semmit nem ír a képernyőre vagy a szabványos kimenetre, hacsak nem utasítjuk erre a megfelelő `print` paranccsal.
Aki sokat dolgozott a ***Perl*** nyelvvel, azt valószínűleg meg fogja lepni a következő művelet végeredménye:

```
5 / 2
```

Az eredmény itt ugyanis 2, mivel az 5 és a 2 egyaránt egészek, így a ***Ruby*** feltételezte, hogy egészosztást szándékozunk végrehajtani. Ha nem ez volt a szándékunk, és lebegőpontos műveletet akarunk végezni, akkor gondoskodnunk kell róla, hogy az operandusok közül legalább az egyik lebegőpontos szám legyen:

```
5 / 2.0
```

A nulla megfelelő ponton való kiírása már elég ahhoz, hogy az eredmény a várt 2,5 legyen. Más nyelvekhez képest szintén lényeges eltérés, hogy a ***Ruby*** a tizedestörteknél is megköveteli a tizedespont előtti nullát, vagy a `.5` alak nem helyes, csak a `0.5`.
Ha egy karakterláncot akarunk egésszé vagy lebegőpontos értékké konvertálni, azt az `_i` illet `_f` metódusokkal tehetjük meg:

```
"5".to_i
"5".to_f
```

A ***Ruby*** valamennyi objektuma rendelkezik emellett egy `_s` nevű metódussal is, amely az objektum tartalmát karakterláncá alakítja.
A ***Ruby*** egyik szintén meglepő adattípusa az úgynevezett szimbólum, ami számos kezdőnek okoz majd némi fejtörést, pedig nem különösebben bonyolult. A szimbólum olyasmi, mint egyfajta speciális karakterlánc, ami sokkal kisebb helyet foglal a memóriában, különösen ha számos helyen kell használni. A szimbólumok neve mindig kettősponttal kezdődik (például `:reader`), és valójában nem mindig használhatók a karakterláncok helyett. Ugyanakkor használatukkal a programok könnyebben áttekinthetők, olvashatóbbak lesznek. Egyes esetekben objektumokra és metódusokra való hivatkozásként is használjuk őket, de erről a cikk későbbi részében még lesz szó.

Interpoláció és metódusok

Megannyi más programozási nyelvhez hasonlóan a ***Ruby*** is lehetővé teszi a kettős idézőjelek közé zárt karakterláncok és értékek interpolációját. (Az egyszeres idézőjelek közé írt karakterláncokat a ***Ruby*** betű szerint kezeli, amiben szintén egyezik a legtöbb magas szintű nyelvvel.) Ennek megfelelően használható a következő nyelvi fordulat:

```
name = "Reuven"
"Hello, #{name}"
```

A fenti kifejezés eredménye valójában a következő:

```
Hello, Reuven
```

A `#{ }` jelek között nem csak egy változó, hanem bármilyen ***Ruby*** kifejezés is szerepelhet:

```
name = "Reuven"
print "Hello, #{name}. Your
➥name is #{name.length}
➥letters long."
print "Backwards, your name is
➥'#{name.reverse}'."
print "Capitalized, your
➥backwards name is '#
➥{name.reverse.capitalize}'."
```

Amint látható, az interpoláció segítségével gyakorlatilag tetszőleges összetett kifejezéseket megalkothatunk darabokból, egyetlen kettős idézőjel segítségével. De várjunk csak egy pillanatot. Mit is jelen pontosan a fenti kifejezésekben a `name.length`, `name.reverse` és a `name.reverse.capitalize`?
A válasz egyszerű. ***Ruby***-ban a karakterláncok, mint minden egyéb nyelvi konstrukció, objektumok. Ennek megfelelően gyakorlatilag mindent, amit egy karakterlánccal egyáltalán művelni lehet metódusokkal, és nem önálló függvények segítségével hajthatunk végre. Ha tehát meg akarjuk fordítani egy karakterláncban a betűk sorrendjét, kíváncsiak vagyunk a hosszára, csupa nagybetűssé akarjuk alakítani, vagy fel szeretnénk darabolni, a ***Ruby*** egy-egy a karakterlánc objektumhoz rendelt metódusát fogjuk meghívni, mégpedig az `objektum.üzenet` szintaxist használva. Lássunk egy példát:

```
name.reverse
```

Ez a kódrészlet megfordítja egy karakterlánc objektumban a betűk sorrendjét úgy, hogy az eredménye egy olyan új karakterlánc-objektum lesz, amiben fordítva van a `name`-ben megnevezett objektum tartalma. Mivel az a visszaadott karakterlánc is objektum, ezzel kapcsolatban is használhatjuk a karakterlánc típushoz tartozó valamennyi metódust, vagyis szükség esetén tovább alakítgathatjuk. A megfordítás után például csupa nagybetűssé alakíthatjuk pontosan úgy, ahogy az előbbi példában láttuk. A ***Ruby*** programozók gyakran folyamodnak a metódusok ilyen összefűzéséhez, ha valamilyen összetettebb műveletsort akarnak végrehajtani.
Ügyeljünk rá, hogy a ***Ruby*** nyelv dokumentációjában az egy adott objektumpéldánnyal kapcsolatos metódushívásokat az `Objektumtípus#metódus` jelöléssel illetik, vagyis a fenti művelet ott a következőképpen festene: `String#reverse`.

Na de honnan tudhatjuk, hogy egy adott objektumtípushoz milyen metódusok tartoznak? Ennek az egyik módja az, ha megkérdezzük magát az objektumot, hogy melyik osztálynak a tagja. Ezt a következőképpen tehetjük meg:

```
name.class
```

Kérdezhetünk aztán célirányosabban is, nevezetesen megtudhatjuk magától az objektumtól, hogy egy adott osztálynak tagja-e:

```
name.is_a?(String)
```

Ez a kódrészlet a közönséges földi halandók számára bizony elsőre kicsit furcsán fest részben a metódus nevében szereplő kérdőjel miatt, részben meg azért, mert a metódusnak paraméterként egy típus megnevezését adjuk át, de amúgy minden a legnagyobb rendben, a ***Ruby***-nál ez működik ugyanúgy, mint azok közönséges metódusok, amelyekről eddig volt szó. Amikor egy `name` nevű objektumnak elküldjük az `is_a?` üzenetet, akkor az egy ***Boolean*** (igaz vagy hamis) értékkel válaszol. Az `is_a?` metódus argumentuma egy osztály neve, ami esetünkben a `String`.
Mármost ha nincs kedvünk böngészni a ***Ruby*** programozási felületének terjedelmes dokumentációját, akkor megtehetjük azt is, hogy egyszerűen megkérdezzük magát az objektumot, hogy milyen metódusokkal rendelkezik, vagyis milyen üzenetekre képes reagálni. Ehhez a következőt kell tennünk:

```
name.methods
```

Ez a metódushívás egy tömböt (vagyis egy listát) ad vissza, amely tartalmazza mindazoknak a metódusoknak a felsorolását, amelyekre a kérdéses objektum válaszolni tud. A tömbökről néhány pillanat múlva még bővebben is lesz szó, most azonban a legfontosabb annak a ténynek a felismerése, hogy a válaszul visszakapott objektum nem egy egyszerű karakterlánc, hanem olyan tömb, amelynek az elemei történetesen karakterláncok. Mármost a tömböknek van egy beépített rendező metódusa (sort), ami egy újabb, immár rendezett tartalmú tömböt ad vissza:

```
name.methods.sort
```

Jómagam legalább naponta egyszer leírok egy `Objektum.methods.sort` típusú hívást ahelyett, hogy kinyitnék egy könyvet vagy megnézném a ***Ruby*** online dokumentációját.

Tömbök és hash-ek

Aki korábban dolgozott már a ***Perl*** vagy ***Python*** nyelvekkel, azt valószínűleg nem fogja különösebben meglepni, hogy a ***Ruby*** is rendelkezik beépített tömbökkel és ***hash***-ekkel. Tömböt úgy hozhatunk létre, hogy szögletes zárójelek között megadjuk a tartalmát:

```
an_array = [1, "two", true]
```

Egy tömb tetszőleges számú objektumot tartalmazhat, és minden ilyen objektum tetszőleges típusú lehet, akár egy másik tömb is. A fent megadott tömb például három objektumot tartalmaz amelyek típusa `Fixnum`, `String` és `Boolean` ebben a sorrendben. Egy töm valamennyi eleméhez egyedi index tartozik – elvégre attól tömb – és az első elemnek mindig 0 az indexe. Egy kifejezésben egy tömbelemre a következőképpen hivatkozhatunk:

```
an_array[1]
```

A fenti kifejezés visszatérési értéke `"TWO"` lesz, vagyis az `an_array` nevű tömb 1-es sorszámú (valójában második) eleme. A tömbök módosíthatók (***mutable***), vagyis egy tömbelemnek bármikor új értéket adhatunk a következő módon:

```
an_array[1] = "TWO"
```

Érdekes módon a címzésnél használhatunk negatív számokat is. Ilyenkor a ***Ruby*** a tömb végéről kezd számlálni visszafele, vagyis esetünkben például az `an_array[-1]` hívás eredménye egy ***Boolean*** igaz érték lenne. Lehetőségünk van a tömb egy teljes összefüggő szakaszának megcímzésére is úgy, hogy a nyitó és a záró indexet vesszővel elválasztva adjuk meg a szögletes zárójelek között. Ilyenkor a címzett területhez a két korlát is hozzátartozik.

```
an_array[0,1]
```

Ha egy töm összes elemét összefűzve egyetlen karakterláncot szeretnénk belőlük létrehozni, használhatjuk a `join` metódust a következőképpen:

```
an_array.join(", ")
```

A fenti metódushívás eredménye egyetlen karakterlánc lesz, amely a korábbi tömb (`an_array`) miden elemét tartalmazza vesszővel elválasztva.
A ***hash***-ek meglehetősen hasonlóak a tömbökhöz. Az egyetlen lényeges eltérés az, hogy a tárolt elemeket itt nem rendezett, numerikus értékekkel (indexekkel) lehet azonosítani, hanem kulcsokkal. Lássunk rögtön egy példát:

```
my_hash = {'a' => 1, 'b' => 2}
```

Az így eltárolt két elemhez a következőképpen juthatunk hozzá a hozzájuk tartozó kulcs megadásával:

```
my_hash['a']
my_hash['b']
```

A fenti két sor tehát az 1 és 2 értéket fogja visszaadni, ebben a sorrendben. Akárcsak a tömböknél, az elemek itt is tetszőleges objektumok lehetnek, tehát nem csak egész számok, mint a fenti példában.
A kulcsokat és a hozzájuk tartozó értékeket a `Hash#kulcs` illetve `Hash#érték` metódusokkal kérhetjük le. (Később azt is be fogom mutatni, hogyan haladhatunk végig egy ***hash*** valamennyi kulcs-érték párján.) Egyes esetekben ugyanakkor mindössze arra vagyunk kíváncsiak, hogy egy adott kulcs létezik-e a ***hash***-ben. Nos, ezt is könnyen megtudhatjuk a `Hash#has_key?` metódus segítségével, ami paraméterként egy karakterláncot vár, visszatérési értéke pedig egy ***Boolean*** érték.
A következő kódnak tehát igaz értékkel kell visszatérnie:

```
my_hash.has_key?("a")
```

Feltételes szerkezetek

Minden programozási nyelvben lehetőségünk van arra, hogy egyes

kódrészletek végrehajtását bizonyos feltételek teljesülésétől tegyük függővé. ***Ruby***-ban ezt – nem különösebben meglepő módon – az `if` paranccsal tehetjük meg. Nézzük például a következő – kissé talán kitekert – kódot:

```
if server_status == 0
print "Server is in single-user
➥mode"
elsif server_status == 1
print "Server is being fixed "
elsif network_response == 3
print "Server is available"
else
print "Network response was
➥unexpected value '#
➥{network_response}'"
end
```

Figyeljük meg, hogy ***Ruby***-ban a feltételt magát nem kell zárójelek közé tennünk. És bár a feltételként megadott kifejezésnek nem kell feltétlenül ***Boolean*** értékkel visszatérnie, azért a ***Ruby*** a biztonság kedvéért figyelmeztetést küld milyen olyan esetben, amikor ilyen környezetben az = (vagyis az értékadás) operátort használjuk az == (vagyis az egyenlőség logikai vizsgálata) operátor helyett. Az == vagy összehasonlító operátor valamennyi objektumtípussal kapcsolatban működik, vagyis nincs külön ilyen operátora például a szöveges és numerikus értékeknek, mint a ***Perl*** nyelvben. Ugyanez igaz a kisebb/nagyobb (< és >) jellegű vizsgálatokra is, amelyekkel számokat és karakterláncokat egyaránt összehasonlíthatunk. Végezetül a feltételes szerkezetek tagolására a ***Ruby*** nem használ kapcsos zárójeleket. Az ilyen kódblokk végét az `end` utasítás jelzi. Akárcsak a ***Perl***-ben, a ***Ruby*** esetében is használhatjuk az `if` és `unless` kulcsszavakat arra, hogy segítségükkel feltételes szerkezetté alakítsunk egy kifejezést:

```
print "We won!" if our_score >
➥their_score
print "Here is your change of
➥#{amount_paid - price}!"
    unless amount_paid <= price
```

Hasonlóan akár efféle dolgokat is művelhetünk:

```
if inputs.length < 4
    print "Not enough
➥inputs!\n"
end
```

Sőt, ha már itt tartunk, még a következő forma is helyes:

```
if not my_hash.has_key?
➥("debug")
    print "Debugging is
➥inactive.\n"
end
```

Ciklusok

A ***Ruby*** – mint a többi nyelv – többféle ciklusszervező utasítást tartalmaz. Ilyen például a máshonnan is jól ismert `for` és `while`. Ez eddig természetes. Na de az igazi élvezetet az olyan szerkezetek használata jelenti, mint például ez itt:

```
5.times {print "hello\n"}
```

Lássuk csak mi is történt itt... Van ugye egy szám, és mi a szokásos ***Ruby*** szintaxis szerint meghívtunk egy ehhez tartozó metódust. Az egészekhez – mint objektumokhoz – tartozó `times` metódus egy adott kódrészletet annyiszor hajt végre, amennyi maga a kérdéses szám. A fenti sorban tehát az történik, hogy ötször végrehajtjuk a kapcsos zárójelek között megadott utasítást, ami ennek megfelelően ötször fogja kiírni a képernyőre a `"hello"` szöveget (amit minden esetben egy újsor karakter követ). Az egyes kódblokkok egymás között is átadhatnak paramétereket a csövek (|) segítségével:

```
5.times {|iteration| print
➥"Hello, iteration number
➥#{iteration}.\n"}
```

Hasonlóan érdekes nyelvi szolgáltatás az `each` metódus, amivel egy tömb elemein mehetünk végig:

```
an_array = ['Reuven', 'Shira',
➥'Atara', 'Shikma', 'Amotz']
an_array.each {|name| print
➥"#{name}\n"}
```

Az `each` metódus egy továbbfejlesztett változata az `each_with_index` nevű metódus, amely bemenetként egy két paraméterből álló blokkot vár. Az első paraméter az elem, míg a második az index:

```
an_array = ['Reuven', 'Shira',
➥'Atara', 'Shikma', 'Amotz']
an_array.each_with_index
➥{|name, index| print
➥"#{index}: #{name}\n"}
```

Van egy pont az összetettségben, ahol aztán a blokkok a fenti szintaxist alkalmazva nehezen követhetővé válnak a kódban. Éppen ezért a ***Ruby*** rendelkezésünkre bocsát egy másik szintaxist is, amelyben a kapcsos zárójelek helyett a `do` és az `end` kulcsszavak használhatók, valahogy így:

```
an_array = ['Reuven', 'Shira',
➥'Atara', 'Shikma', 'Amotz']
an_array.each_with_index do
➥|name, index|
    print "#{index}: #{name}\n"
end
```

Egy ***hash*** elemein többféle módszerrel mehetünk végig. Az egyik megoldás annak a módszernek a használata, amit a Perl és a Python felhasználók bizonyára már évek óta ismernek: lekérdezzük a ***hash*** kulcsot (a `Hash#keys` konstrukcióval, ami egy tömböt ad vissza), majd a tömbön iterálva lekérdezzük az egyes kulcsokhoz tartozó értékeket.

```
state_codes = {'Illinois' =>
➥'IL', 'New York' => 'NY',
                'New Jersey' =>
➥'NJ', 'Massachusetts' => 'MA',
                'California' =>
➥'CA'}
state_codes.keys.each do
➥|state|
    print "State code for
➥#{state} is #{state_codes
➥[state]}.\n"
end
```

Persze a kulcsokat nem árt rendezni, mielőtt megkezdenénk az iterációt:

```
state_codes.keys.sort.each do
➥|state|
    print "State code for
➥#{state} is #{state_codes
➥[state]}.\n"
end
```

Bár ez a módszer tökéletesen működik, a ***Ruby*** mindazonáltal rendelkezésünkre bocsát egy egyszerűbb módszert is ennek a feladatnak

a megoldására. Ez a módszer pedig nem más, mint az `each_pair` metódus, amit a következőképpen használhatunk:

```
state_codes.each_pair do
➥|state, code|
    print "State code for
➥#{state} is #{code}.\n"
end
```

Osztályok és metódusok

Az alapokon ezzel szerencsésen túl is jutottunk. Igazából nincs már hátra, mint összegzésképpen létrehozni egy saját osztályt, és néhány hozzá tartozó metódust. Az ***irb***-ben, vagy a ***Ruby*** kódban akárhol létrehozhatunk egy saját osztályt, ha a következő utasítást adjuk ki:

```
class Simple
end
```

Ugye nem bonyolult. Két sor az egész, és máris van egy új osztályunk.
Na de elég lesz ez ahhoz, hogy létrehozzunk egy `Simple` nevű, ebben az osztályba tartozó objektumot? Lássuk:

```
foo = Simple.new
foo.class
```

Nos, a leghatározottabban úgy fest, hogy a `foo` nevű változó úgy tudja magáról, ő a `Simple` osztály eleme. Mivel a deklaráció során még csak azt sem mondtuk meg, hogy a `Simple` osztály mely osztály metódusait örökölje, automatikus öröklés lépett életbe, melynek során `Simple` a nyelv szintjén megadott `Object` osztály leszármazottja lett. A ***Ruby*** csupán egyszeres öröklődést engedélyez, amit a deklarációban a következőképpen fejezhetünk ki:

```
class SimpleArray < Array
end
```

Most tehát már van két saját osztályunk, ami szép és jó, de saját metódus megadására még nem láttunk példát. A ***Ruby*** lehetővé teszi, hogy egy magunk által deklarált osztályt bármikor felnyissunk, és új metódusokat rendeljünk hozzá, vagy módosítsuk a már meglevőket.
Új metódust a `def` paranccsal hozhatunk létre. Itt természetesen jeleznünk kell azt is, hogy a metódus vár-e paramétereket. Lássunk rögtön egy példát:

```
class Simple
    def id_squared
        return self.object_id
➥* self.object_id
    end
end
```

Az általunk imént létrehozott metódus meglehetősen egyszerű, sőt, ha jobban megnézzük olyasvalamit csinál, amire épeszű programban nem nagyon lesz soha szükségünk: veszi a kérdéses objektum egyedi azonosítóját (ez az objektumtulajdonság az automatikus öröklés során keletkezett és az `object_id` nevű változó tartalmazza), majd négyzetre emeli és visszaadja a hívónak. (Ez utóbbi bizonyára egy `Bignum` típusú objektum lesz már csak a mérete miatt is.)
Ha a fenti definíciót begépeljük az ***irb***-ben, valami csodálatos dolog fog történni: a korábban létrehozott `foo` nevű, amúgy a `Simple` osztályba tartozó objektumunk immár válaszolni fog a `Simple#id_square` hívásra, pedig amikor létrehoztuk, még nem is volt az osztálynak ilyen metódusa! Nos igen, a ***Ruby*** lehetővé teszi, hogy a metódusokat röptében módosítsuk, és hogy menet közben felnyissunk és átírjunk már létező osztályokat. Akár azt is megtehetjük például, hogy a beépített `Array` és `String` osztályokat átszabjuk egy kicsit a programunkban úgy, hogy egyes metódusaikat lecseréljük a saját változatunkra.
Végezetül fejlesztéseink során nyilván szükségünk lesz majd arra, hogy az objektumokban valamiféle állapotot tároljunk. Erre a ***Ruby*** példányosított változói (***instance variable***) adnak lehetőséget. Az ilyen speciális változók nevét mindig egy `@` karakter előzi meg, ami kissé zavaró lehet mindazoknak, akik írtak már ***Perl*** programot:

```
class Simple
    def initialize
        @simple_data = [ ]
    end
end
```

A speciális `initialize` metódus mindannyiszor lefut, valahányszor létrehozunk egy új, a `Simple` osztályba tartozó objektumot. Tegyünk most így, és adjuk meg újra a `foo` nevű változót, mint a `Simple` osztály egy példányát:

```
foo = Simple.new
```

Ennek az lesz a következménye, hogy a `foo` változónak keletkezik egy példányosított változata, amelynek létezéséről a következő metódushívással szerezhetünk tudomást:

```
foo.instance_variables
```

Ez a metódushívás egy tömböt ad vissza:

```
["@simple_data"]
```

De hogyan adhatunk értéket a `@simple_data` elemnek? És hogyan kérdezhetjük le a tartalmát? Ennek az egyik módja az, ha létrehozunk néhány megfelelően kialakított metódust: az egyikkel írni, a másikkal olvasni lehet majd ezt a példányosított változót. De nem kell föltétlen ilyen bonyolult módon eljárnunk, létezik ugyanis erre a célra egy `attr_reader` és egy `attr_writer` metódus:

```
class Simple
    attr_reader :simple_data
    attr_writer :simple_data
end
```

A fenti kód azt mondja a ***Ruby*** rendszernek, hogy van nekünk egy `@simple_data` nevű példányosított változónk, és azt szeretnénk, hogy ehhez tartozzon két olyan metódus, amelyekkel a tartalmát írni illetve olvasni lehet. És ezzel el is érkeztünk arra a pontra, amikor láthatjuk, miként lehetnek segítségünkre a szimbólumok – amelyek nem igazán karakterláncok, de nem is literálisan értelmezendő adattípusok – a példányosított változókra való hivatkozásban. Mindezekkel a nyelvi szolgáltatásokkal a kezünkben a következőkhöz hasonló dolgokat művelhetünk:

```
foo = Simple.new
foo.simple_data = 'abc'
foo.simple_data = [1, 2, 3]
print foo.simple_data.join(', ')
```

Összefoglalás

Az elmúlt egy vagy néhány évben a ***Ruby*** nyelv bámulatos népszerűségre tett szert, ami nem kis részben annak köszönhető, hogy a webes alkalmazások fejlesztői között egyre többen használják a ***Ruby on Rails*** keretrendszert. Ugyanakkor a ***Ruby*** mint új nyelv a ***Rails*** rendszer nélkül is kifejezetten figyelemre méltó jelenség. Az a tény, hogy ***Ruby***-ban mind tárolt adat egyben objektum is, kompakt és kifejezetten elegáns programok írását teszi lehetővé mind a metódusok, mind a blokkszerkezetek szintjén. Ráadásul a nyelvhez tartozó szabványos könyvtár rengeteg előre definiált objektumtípust és ezekhez tartozó metódust tartalmaz, ami határozottan lenyűgöző képességekkel ruházza fel a nyelvet.

Ebben a cikkben természetesen nem mehettem bele számos olyan részletnek a tárgyalásába, amelyek ugyanakkor a ***Ruby*** fejlesztők számára érdekesek lehetnének. Ilyen például a modulok, vagy az osztályváltozók témaköre, a fájlokkal kapcsolatos bemeneti és kimeneti megoldások, a hálózati szolgáltatások, az ***XML*** adatszerkezetek feldolgozása, az interneten szabadon hozzáférhető ***RubyGems*** könyvtár, vagy a szabályos kifejezések használatának lehetősége. A ***Ruby*** egyrészt szolgáltatásokban rendkívül gazdag nyelv, ugyanakkor teljes szerkezetét tekintve konzisztens és határozottam könnyen tanulható mindazok számára, akik rendelkeznek valamelyes jártassággal az objektum-orientált programozás területén. Jómagam úgy gondolom, hogy a ***Ruby*** logikájának megértéséhez ez az utóbbi a legfontosabb.

A ***Ruby*** ugyanakkor fejlődőben lévő nyelv, és vannak még megoldásra váró problémái. Ilyen például a viszonylag lassú működés vagy a Unicode támogatás teljes hiánya, de ezeknek a megoldásán a fejlesztők már dolgoznak, és a nem túl távoli jövőben minden bizonnyal sikerrel is járnak majd. Ami pedig a fejlesztői csapat attitűdjét illeti, nos én nem sok hozzájuk hasonlóan jól szervezett és összetartó társaságot láttam.

Jómagam az elmúlt egy év során egyre több és több feladat megoldására használtam a ***Ruby*** nyelvet, és azt kell mondjam, minél jobban megismertem, annál inkább lenyűgözött. Éppen ezért minden olvasómnak csak javasolni tudom, hogy tegyen vele egy próbát. Szerintem megéri. Még ha nem is válik egyhamar az elsődlegesen használt programozási nyelvünkké, akkor is elültetheti bennünk egy új gondolkodásmód csíráit, sőt az sem kizárt, hogy e megismerés által egy másik nyelven fogunk szebb vagy jobb programokat írni, vagy egyszerűen csak jobban fogjuk élvezni a programfejlesztést.

Linux Journal 2006., 147. szám

Reuven M. Lerner hosszú ideje dolgozik web és adatbázis szakértőként, jelenleg pedig PhD-hallgató Evanstonban a Northwestern Univerty-n, Illinois államban. Ő és felesége nemrég ünnepelték fiuk Amotz David születését.

KAPCSOLÓDÓ CÍMEK

➲ www.linuxjournal.com/article/901