13

String

13.1 String İşlemleri

Bilgisayara giriş ve çıkışlar, bizim anlayacağımız karakterlerle yapılır; yani kullandığımız alfabenin karakterlerinden oluşan array'lerdir. Onlara *string* diyoruz. Farklı dillerde ve farklı alfabelerle yazılmış stringleri kullanabilen tek bir editör yoktur. Ama bütün editörler sayıları okuyup yazabilir. Çünkü sayılar binary sistemle yazılabilir. Öyleyse, editörlerin, karakterleri önce sayılara dönüştürerek kaydetmesi ve gerektiğinde onları kayıt ortamından tekrar çağrırıp bizim anlayacağımız karakterlere dönüştürmesi mümkün olan akılcı bir yoldur. Aslında, başka yol olmadığı için, bütün editörler bir dosyayı kaydederken ve kaydedilmiş dosyayı çağırıp okurken bu yolu izler. Farklı alfabaleri (karakterleri) sayılara dönüştüren sistemlere `kodlama` sistemleri denilir. Yazık ki, tek bir kodlama sistemi yoktur. Dünyada kullanılan çok sayıda kodlama sistemleri vardır. Mevcut editörlerin hiçbirisi bütün kodlama sistemlerini kullanamıyor. Dolayısıyla, kaynak programı yazarken kullandığımız editörün hangi kodlama sistemini kullandığı önem taşır.

Karakterlerin bileşimleri birer stringdir (metin). Bilgisayara giren her şeyi string olarak yazarız. Bilgisayardan gelen çıktılar da stringe dönüşmüş olarak bize ulaşır. Biz yalnızca onları yazar, okur ve anlarız. Kullandığımız editör, bilgisayara girecek olan stringleri bir `kodlama` sistemine göre sayılara dönüştürür. Bilgisayara girişler, o kodlama sisteminden binary sisteme dönüşür. Çıktılar için de bu yolun tersi izlenir. Bilgisayarın anladığı `binary` kodlar önce bir kodlama sistemine göre sayılara dönüşür. Sonra o sayılar ilgili alfabenin karakterlerine dönüşür. Biz çıktı olarak onları görür

ve anlarız.

Görülüyor ki, bilgisayara giriş ve çıkışlar için kullanılan kodlama sistemi, kullanıcı ile etkileşimi sağlayan tek araçtır. Bilgisayarın kullandığı binary sayılar farklı kodlama sistemlerinde farklı karakterlere dönüştürülebilir. Girdiğimiz her şey *string* tipindendir (input). Onlar editör tarafından bir kodlama sisteminin sayılarına dönüşür. En sonunda da bilgisyara girerken o sayılar, bilgisayarın anladığı makine diline dönüşür. Bilgisayar çıktıları için de bu yolun tersi izlenir. İçeride makine dili ile yazılan veriler dışarıya çıkarken, bir kodlama sisteminin sayılarına, oradan da ilgili kodlama sisteminin karakterlerine dönüşür. Başka bir deyişle, içeride makine diliyle yazılan veriler ekran, printer vb. gibi bir çıkış ortamına giderken string tipine dönüşür.

Tabii, kolay algılanırlığı sağlamak için, çıktıya özel biçemler vermek mümkündür. Örneğin çıkan veriyi metin olarak yazmak, sayı olarak yazmak, sayıları binliklerine ayırmak, kesir hanelerini azaltıp çoğaltmak, çıktıyı sağa ya da sola yanaşık yazmak vb. gibi eylemler *çıktının biçemlenmesi* eylemleridir. Bunları yapmak, çıktının kolay algılanmasını sağlar. Bütün çıktılar string olduğuna göre, string biçemleyen metotların olması gerekir.

String biçemleme metotları bir çok dilde ortaktır. Ruby'de çok sayıda string metodu vardır. Burada başlıcalarına örnekler vermekle yetineceğiz.

Kaynak programın hangi kodlama sistemiyle yazıldığını belirten bir açıklama satırı koymak, çoğu sisteme yapacağı dönüşüm için bilgi verir. Kaynak programın başına yazılacak coding: kodlama_adı ifadesi yeterlidir. Bunu çok değişik biçimlerde yazabiliriz. Örneğin, kodlama isteminin UTF-8 olduğunu belirtmek için, aşağıdakilere benzer açıklamalar konulabilir Bunlara, unix ve türevleri olan işletim sistemlerinde *sihirli açıklamalar* (magic commments) denilir.

Liste 13.1.

```
# coding: UTF-8
# encoding: UTF-8
# zencoding: UTF-8
# vocoding: UTF-8
# fun coding: UTF-8
# decoding: UTF-8
```

yazmak,

13.2 Ruby'de String

Ruby'de *String* bir sınıftır. Byte (8-bit) dizimlerinden oluşur. Tipik olarak byte alanlarında karakterler yer alır. Karakterler okunabilir metinlerdir (text). Ruby, string nesnelerini yaratmak, onlara erişmek ve onlar üzerinde işlem yapmak için üst-düzeyde ve alt-düzeyde çalışan metotlara sahiptir.

Ruby'de *string literal*'i belirli tipten nesne yaratan sözdizimidir. Örneğin 12 bir *Fixnum* nesnesi yaratan bir literal'dir. String literalleri bir kaç biçimdedir.

String sınıfı çok sayıda metoda sahiptir. Onlar, metinlerle yapılabilecek hemen her işlemi yapmaya yeterlidir.

Text şablonları Ruby'de düzgün ifade *(Regexp)* nesneleri olarak işlenir. Ruby, düzgün ifadeler için gerekli sözdizimini sunar.

Ruby'de tek tırnak ya da çift tırnak içine alınan karakter dizileri birer string oluşturur. Ama tek tırnak (') ile çift tırnak (" ") davranışları farklıdır. Bu farklılığı biraz sonra ele alacağız. Ruby'de string yapısı karakter arrayi olma özelikleri yanısıra başka özeliklere de sahiptir.

13.3 Ruby'de Karekterler

Ruby'de *karakter (char, character)* veri tipi yoktur. Uzunluğu 1 olan String bir karakter yerine geçer. Karakterler String'in altstringi olarak elde edilebilir

13.4 Ruby'de String Yaratma

Ruby'de stringler (metinler) `String` sınıfından türetilen nesnelerdir. Onlara, *String* sınıfının bütün metotları uygulanabilir.

Yeni bir metin (string literal) yaratmak için farklı yöntemler kullanılabilir:

Liste 13.2.

1. String sınıfı kullanılarak yeni bir string yaratmak için *new* operatörü kullanılabilir.

```
str1 = String.new  #=> ""
```

   Bu yöntem çift tırnak içinde ("") ile gösterilen *boş* bir string yaratır.

2. Yaratılacak metin, yukarıdaki *new* operatörünün argümanı gibi yazılabilir:

```
str2 = String.new("Bu bir stringdir.")
```

3. Çekirdekte var olan şu yöntem de string nesnesi yaratır:

```
str3 = String("Bu da bir string\'dir")
```

4. Belki de string nesnesi yaratmanın en kolay yolu, istenen metni (nesne) bir değişkene atamaktır. Gerisini Ruby yapacaktır.

```
str4 = "String nesnesini kolay yaratma!"
```

Tek Tırnak ve Çift Tırnak Kullanımı

Ruby'de metinler (string literals) tek ya da çift tırnak içinde yazılabilir.

Liste 13.3.

```
str1 = 'Bu basit bir Ruby string literalidir.'
str2 = "Merhaba Ruby!"
```

Ruby'de *str* nesneleri tek tırnak (' ') içinde, iki tırnak (" ") içinde, üç tane tek tırnak (''' ''') ya da üç tane çift tırnak (""" """) içinde oluşturulabilir. Tırnak simgeleri stringin başlangıç ve bitiş yerlerini gösteren sınırlayıcılardır. İçiçe ikiden çok sınırlayıcı olduğunda, Ruby en dıştaki hariç, içte kalan sınırlayıcıları da stringin içinde sayar. Aşağıdaki bildirimler geçerlidir; yani her birisi bir *String* nesnesi yaratır:

Liste 13.4.

```
str1 = 'Uzun ince bir yoldayım '
str2 = "Gidiyorum gündüz gece "

str4 = '''Gidiyorum gündüz gece'''
str4 = """Gidiyorum gündüz gece"""
```

Gerçekten bunların string tipinden olup olmadıklarını denetlemek için, Ruby'un `class` metodunu kullanabiliriz.

Liste 13.5.

```
str1.class    # => String

str2.class    # => String

str3.class    # => String

str4.class    # => String
```

`class` metodunun verdiği yanıtlardan çıkan sonuç şudur: yukarıdaki deyimlerle tanımlanan dört değişkenin her birisi `String` sınıfına ait bir nesnenin referansıdır; yani herbirisi `String` tipi bir nesne işaret eden pointerdir.

Son iki bildirim açıklama (döküman) hazırlamaya da yaradığı için, işlevleri ilk ikisinden farklıdır. Çoğunlukla, *String* tipinden değişken tanımlarken, ilk iki yöntemi kullanacağız; yani stringi ya tek tırnak ya da çift tırnak içinde vereceğiz.

Aşağıdaki deyimler *metin1* ve *metin2* adlı iki string yaratır.

Liste 13.6.

```
metin1 = 'Merhaba Ruby!'
metin2 = "Ruby nesne tabanlı bir programlama dilidir."
```

Ruby'de String Bildirimi

Ruby'de string tipi değişken bildirimi, genel değişken bildirimi kuralına uyar. Bir değişken adına string tipinden bir değer atanınca, bildirim tamamlanmış olur. Tabii, Ruby'de değişkenlerin birer *pointer* olduğunu unutmuyoruz. Değişken, kendisine atanan nesnenin ana bellekteki adresini gösterir. Ona *işaretçi, etiket, referans* gibi adlar da verilir.

Alıntılar

Bir alıntı yaparken tek ve çift tırnak karakterleri ayrı işlevler için kullanılabilir.

Liste 13.7.

```
'Yunus der ki "Beğler azdı yolundan, bilmez yoksul halinden" '
  'Yunus der ki "Beğler azdı yolundan, bilmez yoksul halinden" '

"Yunus der ki 'Beğler azdı yolundan, bilmez yoksul halinden' "
  "Yunus der ki 'Beğler azdı yolundan, bilmez yoksul halinden' "
```

(\) operatörü Kaçış (escape) Karakteri

Tek tırnak içinde kaçış (\) karakterinin oynadığı roller alıştıklarımızdan biraz farklıdır. O farklılıkları bilmekte yarar vardır. Bu konuyu örneklerle açıklamak daha kolay olacaktır

Liste 13.8.

```
str3 = 'Ankara\'da yaşıyorum.'
str4 = 'Bir kaçış karakteri : (\) kendini çift yapar.'
'a\b' == 'a\\b' #=> true
str5 = 'İki kaçış karakteri : \\abc olduğu gibi yazılır'
str6 = 'Üç kaçış karakteri  : \\\abc olduğu gibi yazılır'
```

(\) karakterini tek tırnak (') karakteri izliyorsa, (\) karakteri (') için *escape* rolünü oynar; yani onun Ruby'deki işlevini askıya alır, işletim sistemindeki rolünü oynamasını sağlar. Dolayısıyla (') karakterini yazar (str3). Tek tırnak (') içinde bir tek (\) karakteri var ama onu izleyen (') karakteri yoksa, Ruby (\) karakterini kendiliğinden çift yapar; birincisi ikinci için *escape* rolünü oynar (str4 ve sonraki satıra bakınız).

(\) karakterini (\) karakter(leri) izliyorsa; normal olarak, soldakinin kendi sağındaki (\) için kaçış karakteri rolünü oynaması beklenir. Ancak bazı sistemlerde böyle olmaz. Başka bir deyişle (') karakterinin izlemediği (\) karakteri normal bir karakter gibi davranır; *escape* rolünü oynamaz.

Tek ve çift tırnak kullanımı arasında bilinmesi gereken en önemli fark, tek tırnak içinde Ruby string *yerleştirme* (interpolation) yapamaz, ama çift tırnak içinde yapabilir. Tek tırnak kullanımına *hard code*, çift tırnak kullanımına da *soft code* denilir. Örneğin, *soft code*'da bir metin içine bir değişken değerini #{...} içine yerleştirebilir; ama *hard code*'da yapamaz. Aşağıdaki deyimler kullanıcıdan adını sorup, ona selam veriyor:

Liste 13.9.

```
print "Adınız nedir? "
ad = gets.chomp
puts "Merhaba, #{ad}"
```

Bunu bir de *hard code* olarak deneyiniz. *Hard code* string literalleri *soft code* literallerine göre çok daha kısıtlıdır.

\n operatörü Bir çok dilde olduğu gibi, Rubyda da satırbaşı (newline) eylemi \n operatörü ile yapılır.

Liste 13.10.

```
s = "Ala gözlerini sevdiğim dilber \nGöster cemalini görmeye
 geldim \n–Karacaoğlan–"

print(s)
```

```
Ala gözlerini sevdiğim dilber
Göster cemalini görmeye geldim
–Karacaoğlan–
```

\t operatörü String içinde tab yapmak için kullanılır.

```
print("Deniz,\t,1972")
Deniz	1972
```

13.5 Stringleri Birleştirme

İki metni (string) birleştirmek için Ruby'de farklı yöntemler izlenebilir. Bunlardan bazılarını şimdi ele alabiliriz.

+ operatörü: İki stringi birleştirmek (concatenate) için (+) operatörü kullanılabilir. Birleşen stringlerin arasına boşluk koymak için " " boş string konulabilir. Örneğin,

```
a = "Bilmek istersen seni"
b = 'Sen seni bil sen seni'
c = "\n—Hacı Bayramı Veli--\n"
d = a + " " + b + " " + c
print d
```

```
/**
  'Bilmek istersen seni
   Sen seni bil sen seni'
   --Hacı Bayramı Veli—
*/
```

13.6 Altstring

13.6.1 Ruby'de Altstring İşlemleri

String tipi *Array* tipi gibidir. Byte'lardan oluşan dizimdir. O nedenle *array* yapısına benzer işlevlere sahiptir.

Bir string içinde yer alan karakterler bir karakter dizimi oluşturur. Stringin ilk karakteri 0-ıncı öğe, ikinci karakteri 1-inci öğe, . . . vb. dir. Başka bir deyişle, Ruby *Array*'in öğelerini saymaya 0'dan başlar. n tane öğesi olan bir *Array*'in son öğesi (n-1)-inci öğedir. *Array*'in öğelerinin sıra numarası, array yapısında olduğu gibi ([ ]) köşeli parantez içine yazılır. Örneğin,

Liste 13.11.

```
a = "Bilmek istersen seni"
b = "Sen seni bil sen seni"
```

stringinleri verilsin. İşlemlerde yazma kolaylığı için stringlere `a` ve `b` gibi kısa adlar verdik. Ancak, uzun programlarda böyle kısa adlar yerine, değişken ve metotlara işlevlerini çağrıştıracak kadar uzun adlar verilmelidir.

class, length, a[0], a[3], a[21] deyimlerine ruby aşağıdaki yanıtları verecektir.

Liste 13.12.

```
a = "Hayat beklentilerle doludur!"
a.class    # => String
a.length   # => 28
a[0]       # => "H"

a[3]       # => "a"
a[21]      # => "o"
```

x bir değişken ise *class* metodu x'in veri tipini, yani hangi sınıftan türediğini; *length* metodu ise x listesinin uzunluğunu verir. Örnekteki a stringinin ilk karakteri $a[0] ==' H'$, dördüncü karakteri $a[3] ==' a'$, ..., yirmi ikinci karakteri ise $a[21] ==' o'$ olur.

String İçinden Karekter Seçme

Genel olarak, bir *str* stringinden indisi r olan karakteri seçmek için aşağıdaki deyimi kullanırız.

Liste 13.13.

```
    str[r]
```

Uzunluklar `length` ile `size` denk iş yaparlar. `bytesize` string'deki byte sayısını verir. `empty?` metodu string boş ise `true` verir.

Liste 13.14.

```
str = "Yunus sordu girdi yola ..."
=> "Yunus sordu girdi yola ..."
str.length      # => 26
str.size        # => 26
str.bytesize    # => 26
str.empty?      # => false
"".empty?       # => true
```

split metodu *split* metodu, sınırlayıcılara göre bir metni alt metinlere böler. Argümansız olunca, beyaz alanlar öntanımlı (default) sınırlayıcıdır.

Liste 13.15.

```
# encoding utf-8

str ="Ruby dersi var"
arr = str.split(//)
puts str
print arr
```

```
\**
Ruby dersi var
["R", "u", "b", "y", " ", "d", "e", "r", "s", "i", " ", "v", "a", "r"
 ]
*/
```

Liste 13.16.

```
s = "Uzun ince bir yol" # => "Uzun ince bir yol"

s.split          # => ["Uzun", "ince", "bir", "yol"]

s.split('i')       # => ["Uzun ", "nce b", "r yol"]

"1, 2,3".split(/\d+/) # => ["", ", ", ",", ","]

"1, 2,3".split(/\s*/) # => ["1", ",", "2", ",", "3"]
```

chars.to_a

Liste 13.17.

```
s ="Trenle geldi"   # => "Trenle geldi"
s.chars.to_a
=> ["T", "r", "e", "n", "l", "e", " ", "g", "e", "l", "d", "i"]
```

scan metodu

Liste 13.18.

```
"ARILARIN KANATLARI".scan /\w/
=> ["A", "R", "I", "L", "A", "R", "I", "N", "K", "A", "N", "A", "T",
 "L", "A", "R", "I"]
```

class metodu

Liste 13.19.

```
123.class        # => Fixnum
12345678790123456789.class  # => Bignum
0b010101.class     # => Fixnum (Binary)
0x2a4b.class     # => Fixnum (hex)
0123.class       # => Fixnum (Octal)
'A'.class      # => String
'abc'.class      # => String
"abc".class      # => String
true.class       # TrueClass
false.class      # FalseClass
nil.class      # NilClass


o123.class
# => NameError: undefined local variable or method `o123' for
0b123.class
# => SyntaxError: (irb):17: syntax error, unexpected tINTEGER,
```

Altstring Seçme

Liste 13.20.

```
s = "abcdefghi"
s[2..5]                  # => "cdef"
puts s[0, s.length - 3]  # => abcdef
puts s[0..-4]            # => abcdef
```

Açıklama:

Bir stringin bir altstringini seçmek için `[alt_indis : üst_indis]` operatörünü kullanırız. Bu işlem, stringden bir dilim seçme gibidir.

sub Bir stringde bir karakterin ya da bir altstringin yerine başkasını koymak için `sub` metodu kullanılır. *RegEx* ile çok sayıda argüman alabilir.

Liste 13.21.

```
s = "abcdefghi"
s.sub('b','B')          # => "aBcdefghi"
s.sub('bcd','BCD')      # => "aBCDefghi"
```

times Bazen bir stringi bir kaç kez yazmak isteyebiliriz. O zaman (*) operatörünü ya da times metodunu kullanabiliriz:

Liste 13.22.

```
  s = "Ruby "
2 s * 3 # => Ruby Ruby Ruby
3.times do print("Ruby ") end # => Ruby Ruby Ruby => 3
```

include Bir karakterin ya da bir alt stringin bir string içinde olup olmadığını anlamak için *include* metodunu kullanırız:

Liste 13.23.

```
s = "Ruby"        # => "Ruby"
2
s.include?('b')   # => true
s.include?('uby') # => true
7 s.include?('abc') # => false
```

Büyük-küçük Harfe Dönüştürme

Ruby'de büyük-küçük harf dönüşümü yapan üç metot vardır.

- *downcase* metodu stringi küçük harfe dönüştürür.
- *upcase* metodu stringi büyük harfe dönüştürür.
- *capitalize* metodu, string içindeki ilk harfi büyük harfe dönüştürür.
- *swapcase* metodu büyük harfleri küçük, küçük harfleri büyük yapar

```
"Ruby".upcase      # => "RUBY"
3 "PYTHON".downcase   # => "python"
h = "abCDefGH"       # => "abCDefGH"
h.swapcase         # => "ABcdEFgh"
```

13.7 Stringlerin Sınırları

Stringlerin tek ya da çift tırnak içine alındığını söylemiştik. Ama Ruby daha fazlasını yapar. Herhangi bir karakterin önüne % konularak o karakter string sınırlayıcısı yapılabilir:

Örnekler:

Liste 13.24.

```
str1 = 'Bu bir stringdir'
str2 = "Bu bir stringdir"
str3 = %&Bu bir stringdir&
str4 = %KBu bir stringdir K
str5 = %\Bu bir stringdir\
str6 = %=Bu bir stringdir=
str7 = %%Bu bir stringdir%
str8 = %?Bu "da" bir stringdir?
```

Sınırlayıcı olarak parantezler de kullanılabilir. Algılanırlığı artırdığı için, parantezler tercih edilebilir.

Liste 13.25.

```
str1 = %(Bu yaz ayları çok sıcak oldu.)
str2 = %[Bu yaz ayları çok sıcak oldu.]
str1 = %{Bu yaz ayları çok sıcak oldu.}
```

Ruby %q simgesini tek tırnak, %Q simgesini çift tırnak %x simgesini ters kesme (back quote) olarak algılar. Bazı klavyelerde tırnak karakterleri olmayabilir ya da yerleri hemen bilinmeyebilir. O durumlarda, Ruby'nin programcıya sunduğu bu kolaylıklardan yararlanabiliriz.

Bazan string içine alıntıların yazılması gerekir. O durumlarda (\) karakteri işe yarar. Tabii, tek tırnak içindeki alıntıyı çift tırnak içine almak ya da bunun tersini yapmak iyi bir yöntemdir

Liste 13.26.

```
str1 = "Bu \"benim\" yazdığım string\'dir"
str2 = 'Bu \"senin\" yazdığın string\'dir'
str3 = "Bu \'onun \' yazdığı string\'dir"
```

Satırbaşı Bir çok dilde metin içinde satırbaşı (yeni satır, new line) yapmak için (\n) kontrol karakteri kullanılır. *n* karakterinin önündeki (\) karakterine *kaçış (escape)* karakteri demiştik.

Yeni satır komutu, bir satırın bittiğini ve yeni bir satıra başlanacağını bildiren bir karakter ya da karakterler dizisidir. Bazı dillerde CR (carriage return) ve LF (line feed) karakterlerinden oluşur. Örneğin Microsoft Windows CR 'yi izleyen LF karakterleri kullanılır. Bazı sistemlerde yalnızca LF karakteri kullanılır.

Ruby'de tek tırnak içinde satırbaşı (\n) kaçış karakteri işlevsel değildir. Satırbaşı için (\n) kaçış karakterini çift tırnak içinde yazmalıyız.

13.7.1 Değişebilir String

Teknik açıdan bakınca, bir *string* nesnesi yaratıldığında, onun karakterleri ana bellekte ardışık hücrelere yerleşir. Çünkü String nesnesi bir karakter array'idir. O nedenle, hemen her dilde *string* nesneleri ana bellekte yerleştikleri yerde uzayıp kısalamazlar. Başka bir deyişle, string nesneleri değiştirilemez *(immutable)*. Değiştirilmeleri için, ana bellekte değişmiş biçemlerine (format) uyan yeni bir yer ayrılır ve oraya yeni biçemleriyle yazılırlar.

Oysa Ruby dökümanı, Array nesnelerinde olduğu gibi, *String* nesnelerinin de değişebilir *(mutable)* olduğunu söyler. Oysa ana bellekte, String için öteki dillerde yapılanlardan farklı bir eylem düşünülemez. Ama, Ruby, string'de istenen değişikliği kendiliğinden yapıyor; ayrı bir metot çağırmaya ya da yazmaya gerek bırakmıyor; String metotlar bu işi otomatik yapıyor.

Procedural dillerden gelen alışkanlığa uyarak, string'lerle yapılan işlemlere *String işlemleri* diyeceğiz. String çok kullanılan bir veri tipi olduğu için, onunla ilgili işlemler de çoktur. Bu bölümde *Ruby*'de string işlemlerinin başlıcalarını; yani String nesnelerine uygulanan başlıca operatörleri ve metotları inceleyeceiiz. Metotların tam listesi için *Ruby* web sayfasına bakılabilir [12].

Ruby 1.8 sürümünde stringler ascii karakterleriyle yazılırdı. Ruby 1.9 sürümünden sonra String nesneleri *unicode* karakterlerinden oluşur. Bu önemli bir gelişmedir. ASCII ya da genişlemiş ASCII kodlama sisteminin içermediği dillerde string yazmak için *unicode* dönüştürücüsünü kullanmak gerekiyordu.

Tasarruf sağlamak için, çoğu sistem UTF-8 kodlama sistemini kullanıyor. Bu kodlama sisteminde 7 ve 8 bit uzunluğa sığan ascii ve genişlemiş ascii karakterleri aynen korunurken, öteki karakterler unicode olarak yazılıyor (bkz. [8]).

Ruby 1.9 ve sonraki sürümlerinde String'leri utf-8 kodlama sistemiyle yazmak için kaynak programın başına aşağıdaki kodu yazmak yeterlidir

```
# encoding: utf-8
```

Ruby 1.9 sürümü 95'ten çok karakter kodlama sistemi kullanır. utf-8 bunlara dahildir. Bu kodlama sistemleri, dünyadaki bütün alfabeleri yazmaya yeterlidir:

Başlıca Veri Tipleri

Bir nesnenin hangi sınıftan türediğini bilmek için class metodu kullanılır. Örneğin, aşağıdaki nesnelerin tipleri karşılarında yazılı olduğu gibi çıkar.

Liste 13.27.

```
123                         # Fixnum
-123                        # Fixnum (işaretli)
1_123                       # Fixnum (\_ simgesi gözetilmez)
-543                        # Fixnum (negatif)
123_456_789_123_456_789     # Bignum (\_ simgeleri gözetilmez)
123.45                      # Float
1.2e-3                      # Float
0xaabb                      # (Hexadecimal) Fixnum
0377                        # (Octal) Fixnum
-0b1010                     # (Binary [negatif]) Fixnum
0b001_001                   # (Binary) Fixnum
?a                          # ASCII character code for 'a' (97)
?\C-a                       # Control-a (1)
?\M-a                       # Meta-a (225)
?\M-\C-a                    # Meta-Control-a (129)
```

13.8 Program içinde Açıklamalar

Program yazılırken değişkenlerin ve metotların ne iş yaptıklarını açıklamak yararlıdır. Programda güncelleme ya da değişiklik yapılacağı zaman bu açıklama satırları çok işe yarar. O nedenle, yazılım şirketleri büyük programlar için geniş açıklamalar isterler. Böylece, programın ilerideki zamanlarda kolayca güncellenebilmesini güvenceye almış olurlar.

Açıklamalar tek satır ya da çok satırlı olabilir. Bazı dillerde tek satırlı açıklamalar (//) simgeleri ile yapılır. (//) simgelerinden sonra satır sonuna kadar yazılanları derleyiciler görmez.

Ruby'de tek satırlık açıklamalar için (#) simgesi kullanılır. Örneğin,

```
# s değişkeni personelin sicil numarasını tutacaktır
s = 123
```

ya da aynı işi yapmak üzere

```
s = 123   # s değişkeni personelin sicil numarasıdır
```

yazabiliriz. Bu durumda, s= 123 yorumlanacak, ama (#) simgesinden başlayarak satır sonuna kadar yazılanlar yorumlanmayacaktır.

Bazen program içinde yapılan açıklamalar bir satıra sığmayabilir. O zaman çok satırlı açıklama yapılabilir. C, java ve başka bazı dillerde çok

satırlı açıklamalar /* ... */ içine yazılır. Java'da dökümana girmesi istenen açıklamalar /** ... */ içine yazılır.

Ruby'de çok satırlı açıklama öz olarak yoktur. Açıklama satırlarının hepsinin önüne (#) simgesi konulur. Böyle olması belki programdaki açıklamaların kolay algılanmasını sağlıyor.

Ancak, çok satırlı açıklamalar da gerekebilir. Ardışık üç tane tek ya da ardışık üç tane çift kesme simgeleri arasında yer alan satırları Ruby derleyicisi yorumlamaz. Dolayısıyla onlar çok satırlı açıklama olarak kullanılabilirler. Örneğin,

```
''' Bu pythonda
   iki satırlı bir açıklamadır '''
```

ya da

```
""" Bu metin python'da
     üç satırlı bir
     açıklamadır """
```

Ayrıca bu biçimde yazılan açıklamalar döküman olarak da yazdırılabilir. Bu durumuyla python'daki üç tırnak, java'daki /** ... */ açıklamasına benzer.

Stringleri birleştirmek için kullandığımız + operatörü sayılarda kullandığımız + operatöründen farklıdır. Onun adaşlanmışıdır (overloaded).

Aşağıdaki betik, stringleri (+) operatörü ile birleştiriyor.

Liste 13.28.

```
str1 = "Neler yapmadık şu vatan için!"

str2 = "Kimimiz öldük,"

str3 = "Kimimiz nutuk söyledik."

str4 = str1 + "\n" +  str2 + "\n" + str3

print(str4)
```

```
/**
  Neler yapmadık şu vatan için!
  Kimimiz öldük,
  Kimimiz nutuk söyledik
*/
```

Kural 13.1. *Stringlere sayısal işlemlerde kullanılan çıkarma, çarpma ve bölme işlemleri uygulanamaz. (+) işlemi string birleştirme (concat) operatörüdür. Sayılardaki (+) işleminden farklıdır.*

Sözkonusu işlemlerin uygulanamayacağını aşağıdaki örneklerden görebiliriz. Her işlem için python yorumlayıcısının verdiği hata uyarısına dikkat ediniz.

Liste 13.29.

```
"ankara" - "izmir"
# => NoMethodError: undefined method `-' for "ankara":String

"ankara" * "izmir"
# => TypeError: no implicit conversion of String into Integer

"ankara" / "izmir"
 => NoMethodError: undefined method `/' for "ankara":String
```

Kural 13.2. *Stringlere sayı eklenemez, çıkarılamaz. Stringler sayılara bölünemez. (*) operatörü ise stringin sayaç kadar tekrar etmesini sağlar*

Sözkonusu işlemlerin uygulanamayacağını aşağıdaki örneklerden görebiliriz. Her işlem için python yorumlayıcısının verdiği hata uyarısına dikkat ediniz.

Liste 13.30.

```
"abcd" + 123
 => TypeError: no implicit conversion of Fixnum into String

"abcd" - 123
# => NoMethodError: undefined method `-' for "abcd":String

"abcd" / 123
# => NoMethodError: undefined method `/' for "abcd":String\noindent

"Ruby"*3
# => "RubyRubyRuby"
```

Kural 13.3. *Stringlerle sayılar çarpılabilir, ama bu işlem sayılardaki çarpma işlemi değildir; onun adaşlanmışıdır.*

13.9 *Integer* <==> *String* Dönüşümleri

Integer tipi *String* tipe Dönüştürme

Integer tipi bir sayı tırnak içine alınınca *String* tipe dönüşür. Aynı işi yapan *to_s* metodu da kullanılabilir. .

Örnekler:

Liste 13.31.

```
'123'.class    # => String
"123".class    # => String
123.to_s     # => "123"
```

Liste 13.32.

```
123.class      # => Fixnum
"123".class      # => String
'123'.class      # => String
123.to_s       # => String
'123'.to_i       # => Fixnum
"123.456".class    # => String
"123.456".to_f     # => Float
```

Stringi *Integer* tipe dönüştürme

Stringe dönüşmüş Integer tipi bir sayıyı tekrar *Integer* tipe dönüştürmek için *to_i* metodu kullanılır. String'den sayıya dönüşüm için *to_i* ve *to_f* metotları kullanılır

Liste 13.33.

```
"123".to_i     # => 123
"-123".to_i    # => -123
"123".to_i     # => 123
"123abc".to_i # => 123
"ten".to_i     # => 0 (hata uyarısı vermez)
```

Float tipi *String* tipe dönüştürme

Float tipi bir sayı tırnak içine alınınca *String* tipe dönüşür. Aynı işi yapan *to_s* metodu da kullanılabilir. Tersine dönüşüm için *to_f* metodu kullanılır.

```
"123.456".class      # => String
123.456.to_s       # => String
"123.456".to_f       # => Float
```

Uyarı 13.1. *Sayının String tipe dönüşmesi demek, o sayının " " ya da ' ' kesmeleri arasına alınması demektir. Ruby, herhangi bir dilde sayıyı ifade eden terimleri dönüştüremez. Uyarı vermeden* 0 *gönderir.*

Liste 13.34.

```
"oniki".to_i     # => 0
"twelve".to_i    # => 0
```

concat

Liste 13.35.

```
#encoding utf-8
# concat örneği

dil = "Ruby" + " programlama" + " diluge"
puts dil

dil = "Python" " programlama" " dili"
puts dil

dil = "Perl" << " programlama" << " dili"
puts dil

dil = "Java".concat(" programlama").concat(" dili")
puts dil
```

Liste 13.36.

```
#convert örneği
  "123.678".to_i     # => 123
  '123.678'.to_i     # => 0
```

gets örneği

Liste 13.37.

```
#encoding: UTF-8
#downcase örneği
print "Dosyayı internetten indirmek istiyor musunuz? (Evet/Hayır) "

response = gets

if (response.downcase == "Evet")
    puts "indir"
else
    puts "indirme"
end

puts response.inspect
```