PM1/PT Ruby

Zeichenketten

Die Klasse String

Inhalt

- Zeichenketten und Zeichen
- Klasse *String*
- Stringliterale
- Auswerten von Zeichenketten
- Destruktive versus nicht destruktive Methoden: Zeichenketten sind veränderbar
- Methodenkategorien von *String*

Einführung

- **Zeichenketten** sind beliebige Folgen von Bytes, die typischerweise darstellbaren Zeichen entsprechen.

- Jedem Zeichen entspricht ein positiver ganzzahliger Wert. Zeichen-Codetabellen ordnen Zeichen diesen ganzzahligen Werten zu. Bekannte Tabellen sind die ASCII Tabelle (siehe nächte Seite), UTF-8 und cp1252 (Windows).

- **Zeichenketten** sind in Ruby Objekte der Klasse *String*.

- Für einzelne **Zeichen** gibt es in Ruby **keine** eigene Klasse.

- Hingegen gibt es die Möglichkeit Zeichen zu benennen, indem dem Zeichen ein *?* vorangestellt wird: *?x* benennt das Zeichen **x**. Der Wert von *?x* ist *120.*

Ascii Tabelle

000	NUL	033	!	066	B	099	c	132	ä	165	Ñ	198	ã	231	þ
001	Start Of Header (SOH)	034	"	067	C	100	d	133	à	166	ª	199	Ã	232	Þ
002	Start Of Text (STX)	035	#	068	D	101	e	134	å	167	º	200	╚	233	Ú
003	End Of Text (ETX)	036	$	069	E	102	f	135	ç	168	¿	201	╔	234	Û
004	End Of Transmission (EOT)	037	%	070	F	103	g	136	ê	169	®	202	╩	235	Ù
005	Enquiry	038	&	071	G	104	h	137	ë	170	¬	203	╦	236	ý
006	Acknowledge (ACK)	039		072	H	105	i	138	è	171	½	204	╠	237	Ý
007	Bell	040	(	073	I	106	j	139	ï	172	¼	205	═	238	¯
008	Backspace (BS)	041	)	074	J	107	k	140	î	173	¡	206	╬	239	´
009	Horizontal Tab	042	*	075	K	108	l	141	ì	174	«	207	¤	240	-
010	Line Feed **(LF)**	043	+	076	L	109	m	142	Ä	175	»	208	ð	241	±
011	Vertical Tab	044	,	077	M	110	n	143	Å	176	░	209	Ð	242	‗
012	Form Feed **(FF)**	045	-	078	N	111	o	144	É	177	▒	210	Ê	243	¾
013	Carriage Return **(CR)**	046	.	079	O	112	p	145	æ	178	▓	211	Ë	244	¶
014	Shift Out	047	/	080	P	113	q	146	Æ	179	│	212	È	245	§
015	Shift In	048	0	081	Q	114	r	147	ô	180	┤	213	ı	246	÷
016	Dataline Escape (DLE)	049	1	082	R	115	s	148	ö	181	Á	214	Í	247	¸
017	DC 1 (XON)	050	2	083	S	116	t	149	ò	182	Â	215	Î	248	°
018	DC 2	051	3	084	T	117	u	150	û	183	À	216	Ï	249	¨
019	DC 3 (XOFF)	052	4	085	U	118	v	151	ù	184	©	217	┘	250	·
020	DC 4	053	5	086	V	119	w	152	ÿ	185	╣	218	┌	251	¹
021	Negative Acknowledge (NAK)	054	6	087	W	120	x	153	Ö	186	║	219	█	252	³
022	Synchronous Idle	055	7	088	X	121	y	154	Ü	187	╗	220	▄	253	²
023	End Of Transmission Block	056	8	089	Y	122	z	155	ø	188	╝	221	¦	254	■
024	Cancel	057	9	090	Z	123	{	156	£	189	¢	222	Ì	255	
025	End Of Medium	058	:	091	[	124	|	157	Ø	190	¥	223	▀		
026	Substitude	059	;	092	\	125	}	158	×	191	┐	224	Ó		
027	Escape **(ESC)**	060	<	093	]	126	~	159	ƒ	192	└	225	ß		
028	File Seperator	061	=	094	^	127 (DEL)	⌂	160	á	193	┴	226	Ô		
029	Group Seperator	062	>	095	_	128	Ç	161	í	194	┬	227	Ò		
030	Record Seperator	063	?	096	`	129	ü	162	ó	195	├	228	ő		
031	Unit Seperator	064	@	097	a	130	é	163	ú	196	─	229	Õ		
032	*SPACE* (SP)	065	A	098	b	131	â	164	ñ	197	┼	230	µ		

Zeichenketten erzeugen

- Zeichenketten werden mit *String.new()* erzeugt. Das kennen wir bereits für andere Objekte.

```
String.new()   #=> ""
String.new("Good morning")
```

- Da Strings eine sehr häufig auftretender Datentyp in Programmen ist, kann man sie auch erzeugen, indem man sie einfach „hinschreibt“. Wir sprechen dann von **Stringliteralen**.

- Es gibt zwei Arten von Stringliteralen, die mit einfachen und die mit doppelten Anführungszeichen. (Die „Magie“ hinter den doppelten Anführungszeichen lernen wir gleich kennen).

```
"Good morning"
'Good morning'
```

Stringbegrenzer als Zeichen in Strings

- Sie wollen die Zeichenkette ***„Geht's noch?“*** ausgeben.

```
'Geht's noch'   #Fehler
```

- Begrenzungszeichen für Stringliterale

  ' und " müssen in Zeichenketten besonders markiert werden, damit sie nicht als Begrenzungszeichen sondern **wörtlich** interpretiert werden. Man sagt auch, die Zeichen müssen ***escaped*** werden.

- Das Escapezeichen für Zeichen in Strings ist der Backslash „\".

- Die richtige Schreibweise:

```
'Geht\'s noch?'   #ok
```

Auswerten von Zeichenketten

Single vs. Double Quoted

- Ruby unterscheidet **Strings**
  - in einfachen Anführungsstrichen (**single-quoted Strings**)
  - in doppelten Anführungsstrichen (**double-quoted Strings**)
- **Single-quoted Strings** werden verwendet wie hingeschrieben und nicht ausgewertet.
- **Double-quoted Strings** werden ausgewertet.

Funktionsweise der Auswertung

- ein double-quoted String *ein_string* wird ausgewertet.
  - Eine Reihe von Sonderzeichen wird interpretiert.
  - Jeder in *#{....}* eingeschlossene gültige Rubyquelltext wird ausgewertet und ausgeführt.
- Auf dem Ergebnis der Auswertung der Sonderzeichen und der Ausdrücke in *#{... }* wird *to_s()* aufgerufen.
- Diese Ergebnisstrings werden zu *ein_string* zusammengehängt.

Auswerten von Sonderzeichen

- Ergebnis der Auswertung von

```
str = "\123 \124 \125"
puts str
str = "Good \n \t\t morning"
puts str
```

123,124,125 zur Basis 8
entspricht 83,84,85 zur Basis 10
entspricht den Zeichen S,T,U in der ASCII Tabelle

```
S T U
Good
        morning
```

Auswerten von Zeichenketten

Quelltext in #{...}

```
class Kreis
  …
  def to_s
    kreis mp = #{@mittelpunkt}
          r=#{@radius}"
    end
end

class Point
  …
  def to_s()
    return "Point(#@x,#@y)"
  end
end
```

Ergebnis *kreis1.to_s()*

```
kreis1 = Kreis.new()
puts kreis1
```

➔ **kreis mp = Point(20,60) r=15**

Ü3-1-c: Erklären Sie bitte dieses Ergebnis!

Auswerten von Zeichenketten

- In *#{...}* ist beliebiger gültiger Ruby Quelltext erlaubt.
- Es ist daher möglich Methodendefinitionen in **double-quoted** Strings einzubetten und diese aufzurufen.
- In *s* wird in *#{...}* die Methode *the(...)* definiert, die anschließend mit dem String Parameter „*Zeit*“ aufgerufen wird.
- Die Auswertung des Quelltextes in *#{...}* ergibt:

```
s = "Jetzt ist #{def the(x)
        'es ' + x
      end;
    the('Zeit')} für
  abgefahrene Dinge!"

puts s
```

➔ **jetzt ist es Zeit für abgefahrene Dinge**

Die Addition von Strings

- *String* implementiert die Methode ‚+'.
- \+ hängt zwei Strings zu einem neuen String zusammen.
- Die Addition eines *String* Objektes mit einer Zahl hingegen ist nicht möglich. Der Versuch führt auf einen Fehler, dass Zahlen nicht in Strings umgewandelt werden können.

```
puts 12 + 12          #=> 24
puts '12' + '12'   #=> 1212
puts '12 + 12'      #=> 12 + 12

# Fehler
puts '12' + 12
```

In Ruby sind Zeichenketten veränderbar

- In den nachfolgenden Tabellen werden wir eine Reihe von Methodenpaaren sehen, die es mit und ohne ! gibt.
- Es ist eine **Ruby-Konvention**, dass Methoden, die auf ! enden, das Objekt, auf der die Methode aufgerufen werden verändern. Sie arbeiten **destruktiv**.
- Gibt es Paare von Methoden mit und ohne !, dann erzeugt die Methode **ohne** ! immer eine Kopie des Strings, auf dem die Änderungen durchgeführt werden sollen. Sie sind **nicht-destruktiv.**
- Aber nicht alle Methoden ohne ! sind **nicht-destruktiv**.
- Zum Beispiel sind *insert* und *replace* **destruktiv**. Dies erkennt man in den Methodenbeschreibungen daran, dass das Ergebnis eines Methodenaufrufs auf einem Objekt, das Objekt selber ist:
- *str.insert(index, other_str) => str*
- *str.replace(other_str) => str*

Methoden für **Strings**

Kategorie	Methoden
Zusammenfügen (Konkatenieren)	+, concat, <<
Vergleich und Gleichheit	<=>, casecmp, ==, eql?
Elementzugriff	[…]
Elementzuweisung	[…]=,
Groß/Kleintausch	swapcase(!), capitalize(!), downcase(!),upcase(!)
(Leer)zeichen eliminieren	lstrip(!), rstrip(!), strip(!), chomps(!), chop(!)
Löschen / (Teil) Ersetzen	delete(!), slice(!), insert, tr(!), tr_s(!), replace
Tests / Abfragen	empty?, include?, length, index
Umkehren	reverse(!)
Darstellen	inspect, print_s

Methoden für **Strings**

Kategorie	Methoden
Konvertieren	to_i, to_s, to_str
Iteratoren	*, each, each_line, each_byte, up_to (später)
Reguläre Ausdrücke	split , %, match, scan (evtl. später), =~ , sub(!), gsub(!)
Teilnahme in Intervallen	next(!), succ(!)

Strings konkatenieren

Methode	Beispiel
a_str + a_str2	"Hello" + "world" #=> "Helloworld"
a_str.concat(a_str2) *a_str << a_str2* (Konkateniert Strings)	*a = "hello "* *a << "world" #=> "hello world"*

Tests und Abfragen

Methode	Beispiel
str.empty?() => true or false	**"hello".empty?() #=> false** **"".empty?() #=> true** **" ".empty?() #=>**
str.include?(other_str) => true or false	***"hello".include?("lo") #=> true*** ***"hello".include?("ol") #=> false***
length()	"".length() #=> 0 " ".length() #=> 5
str.index(substring [, offset]) => fixnum or nil Startposition des ersten Vorkommens von substring ab dem offset wenn enthalten sonst nil. str.index(fixnum [, offset]) => fixnum or nil Position des ersten Vorkommens von fixnum ab dem offset wenn enthalten sonst nil.	***"hello".index('e') #=> 1*** ***"hello".index('lo') #=> 3*** ***"hello".index('a') #=> nil*** ***"hello".index(101) #=> 1***

Vergleich und Gleichheit

Methode	Beispiel
<=> str <=> other_str => -1, 0, +1 -1, wenn str < other_str 0, wenn str == other_str +1, wenn str > other_str Strings sind lexikalisch geordnet. Die Ordnung der Zeichen ergibt sich aus deren Integercode.	***"abcdef" <=> "abcde" #=> 1*** ***"abcdef" <=> "abcdef" #=> 0*** ***"abcdef" <=> "abcdefg" #=> -1 "abcdef" <=> "ABCDEF" #=> 1***
casecmp wie <=> aber ignoriert Groß/Kleinschreibung	***"abcdef".casecmp("abcde") #=> 1*** ***"aBcDeF".casecmp("abcdef") #=> 0*** ***"abcdef".casecmp("abcdefg") #=> -1*** ***"abcdef".casecmp("ABCDEF") #=> 0***
str == obj wenn obj kein String, dann false, sonst true wenn (str <=> obj) == 0	
str.eql?(obj) str und obj sind eql? gleich, wenn sie gleiche Länge und Inhalt haben	

Elementzugriff

Methode	Beispiel
1.) str[fixnum] => fixnum or nil (Ruby 1.8) => String (Ruby 1.9) ***2.) str[fix1, fix2] => new_str or nil*** ***3.) str[interval] => new_str or nil*** ***4.) str[other_str] => new_str or nil*** 1.) liefert den Code des Zeichens / einelementigen String an der Position fixnum 2.) liefert einen Substring, der an Position fix1 beginnt (Offset) und eine Länge von fix2 hat. 3.) ein Substring, der mit dem Intervalstart beginnt und die Länge der Intervalls hat. 4.) gibt eine Kopie von other_str zurück, wenn other_str in str enthalten ist. Wenn der Offset negativ ist, wird vom Ende des Strings gezählt. Ergebnis ist nil, wenn (a) offset größer Länge von str ist, (b) Länge negativ (c) Start des Intervalls > Ende des Intervalls ist.	***a = "hello there"*** ***a[1] #=> 101*** ***a[1,3] #=> "ell"*** ***a[1..3] #=> "ell"*** ***a[-3,2] #=> "er"*** ***a[-4..-2] #=> "her"*** ***a[12..-1] #=> nil*** ***a[-2..-4] #=> "“*** ***a["lo"] #=> "lo“*** ***a["bye"] #=> nil***

Elementzuweisung

Methode	Aufgabe: Verwandle hello there in Moin folks.
str[fixnum] = fixnum, Zeichenliteral (Ruby 1.8) ***= String (> Ruby 1.9)*** ***str[fixnum] = new_str*** ***str[fixnum, fixnum] = new_str*** ***str[interval] = aString*** ***str[other_str] = new_str*** Ersetzt Teile von ***str*** oder ***str*** gesamt Wenn die Länge des ersetzenden Strings nicht gleich der Länge des zu ersetzenden ist, wird die Länge von ***str*** angepasst. Gibt einen IndexError, wenn ***fixnum*** > Länge von str-1, eine RangeError für interval.	***a = "hello there"***

Groß/Kleintausch

Methode	Beispiel
str.swapcase => new_str (new_str: Groß und Kleinbuchstaben getauschen) ***str.swapcase!***	***"Hello".swapcase #=> "hELLO"*** ***"cYbEr_PuNk11".swapcase #=> "CyBeR_pUnK11"***
str.capitalize => new_str (in new_str ist der erste Buchstabe ein Großbuchstabe) ***str. capitalize! => str or nil***	***"hello".capitalize #=> "Hello"*** ***"HELLO".capitalize #=> "Hello"*** ***"123ABC".capitalize #=> "123abc"***
str. downcase => new_str (neuer String new_str mit Kleinbuchstaben) str. downcase! => str or nil	***"hEllO".downcase #=> "hello"***
str. upcase=> new_str (neuer String new_str mit Großbuchstaben) ***str. upcase! => str or nil***	***"hEllO".upcase#=> "HELLO"***

(Leer)zeichen eliminieren

Methode	Beispiel
str.lstrip => new_str (in new_str sind alle Leerzeichen am Anfang von str entfernt) ***str.lstrip! => self or nil*** (wie str.lstrip. nil, wenn str keine Leerzeichen am Anfang enthält)	***" hello ".lstrip #=> "hello "*** ***"hello".lstrip #=> "hello"*** ***" hello ".lstrip #=> "hello "*** ***"hello".lstrip! #=> nil***
rstrip, rstrip! analog ***lstrip, lstrip!*** nur für die rechte Seite	
strip,strip! kombiniert lstrip / lstrip! und rstrip / rstrip!	
str.chomp(separator=$/) => new_str liefert new_str, in dem der separator am Ende von str gelöscht ist. str.chomp!(separator=$/) => str or nil (wie str.chomp. nil, wenn str kein separator Zeichen enthält.)	***"hello".chomp #=> "hello"*** ***"hello\n".chomp #=> "hello"*** ***"hello\r\n".chomp #=> "hello"*** ***"hello\n\r".chomp #=> "hello\n"*** ***"hello\r".chomp #=> "hello" "hello \n there".chomp #=> "hello \n there"*** ***"hello".chomp("llo") #=> "he"***
chop(!) Entfernt das letzte Zeichen eines Strings	

Löschen / Teilersetzen

Methode	Beispiel
str.delete([other_str]+) => new_str new_str: eine Kopie von str, in der alle Zeichen, die sich aus dem Schnitt der Zeichen in [other_str]+ ergeben, gelöscht sind. ***str.delete!([other_str]+>) => str or nil***	***"hello".delete "l","lo" #=> "heo"*** ***"hello".delete "lo" #=> "he“ "hello".delete***
str.slice(fixnum) => fixnum or nil ***str.slice(fixnum, fixnum) => new_str or nil*** ***str.slice(range) => new_str or nil*** ***str.slice(other_str) => new_str or nil*** Löscht die angegebenen Bereiche in str und liefert den gelöschten Bereich. analog ***slice!***	***string = "this is a string" string.slice!(2) #=> 105*** ***string.slice!(3..6) #=> " is "*** ***string.slice!(/s.*t/) #=> "sa st"*** ***string.slice!("r") #=> "r"*** ***string #=> "thing"***
str.insert(index, other_str) => str Fügt other_str an der Position index in str ein. **Destruktiv**	***"abcd".insert(0, 'X') #=> "Xabcd"*** ***"abcd".insert(4, 'X') #=> "abcdX"*** ***"abcd".insert(-3, 'X') #=> "abXcd"*** ***"abcd".insert(-1, 'X') #=> "abcdX"***
str.replace(other_str) => str Ersetzt den Inhalt str von durch den Inhalt von other_str	s = "hello" #=> "hello“ s.replace "world" #=> "world"

Umkehren

Methode	Beispiel
str.reverse => new_str *str.reverse! => str*	*"stressed".reverse #=> "desserts"*

Konvertieren

Methode	Beispiel
str.to_s => str ***str.to_str => str***	
str.to_i() => integer Parsed einen Integer aus den führenden Zeichen von str. Nicht passende Zeichen nach der gültigen Zahl in str werden ignoriert. Liefert 0, wenn keine Zahl geparsed werden konnte.	***"12345".to_i #=> 12345*** ***"99 red balloons".to_i #=> 99*** ***"0a".to_i #=> 0*** "hello".to_i #=> 0
str.to_f => float Parsed einen Float aus den führenden Zeichen von str. Liefert 0.0, wenn keine Zahl geparsed werden konnte.	"123.45e1".to_f #=> 1234.5 "45.67 degrees".to_f #=> 45.67 "thx1138".to_f #=> 0.0

Teilnahme in Intervallen

Methode	Beispiel
str.succ => new_str ***str.next => new_str*** Liefert den Nachfolger von str, indem die Zeichen, beginnend mit dem letzten alphanumerischen Zeichen (oder beginnend mit dem ersten Zeichen, wenn keine alphanumerischen Zeichen enthalten sind). Ziffern werden immer in Ziffern gewandelt, Buchstaben immer in Buchstaben.	***"abcd".succ #=> "abce"*** ***"THX1138".succ #=> "THX1139"*** ***"<<koala>>".succ #=> "<<koalb>>"*** ***"1999zzz".succ #=> "2000aaa"*** ***"ZZZ9999".succ #=> "AAAA0000"***
str.succ! => str ***str.next! => str***	

Lesen, Konvertieren und Normalisieren von Benutzereingaben

- Ü3-2-c: Schreiben Sie bitte ein Programm, dass solange Benutzereingaben liest, bis der Benutzer „exit“ eingibt! Das Programm liest die Eingaben und konvertiert diese in Zahlen und überprüft, ob die Eingabe eine ganze Zahl oder echte Gleitkommazahl war. Leere Eingaben sollen erkannt werden. Das Programm soll folgenden Ausgaben erzeugen.

```
Willkommen. Sie können das Programm mit exit beenden
Bitte geben Sie eine Zahl ein
12
12 ist eine ganze Zahl
Bitte geben Sie eine Zahl ein

keine Eingabe
Bitte geben Sie eine Zahl ein
13.5
13.5 ist eine Gleitkommazahl
Bitte geben Sie eine Zahl ein

keine Eingabe
Bitte geben Sie eine Zahl ein
EXIT
EXIT ist eine ganze Zahl
exit: Sie haben ein einfaches Programm sehr glücklich gemacht
```

Lesen und Extrahieren von Zeichenketten

- Ü3-3-c: Schreiben Sie bitte ein Programm, das solange Sätze von der Konsole liest und ausgibt, wie viele Wörter eingegeben wurden, bis der Benutzer „exit“ eingibt! Das Programm soll folgende Ausgaben erzeugen.

```
Geben Sie einen Satz ein. Wir zählen die Wörter
dies ist ein satz
Anzahl Wörter = 4
Geben Sie einen Satz ein. Wir zählen die Wörter
 dies ist ein satz
Anzahl Wörter = 4
Geben Sie einen Satz ein. Wir zählen die Wörter
      dies ist          auch ein satz
Anzahl Wörter = 5
Geben Sie einen Satz ein. Wir zählen die Wörter
```

Lesen und Schreiben auf Dateien

```
puts "Wir schreiben 'Hi there 99' in die Datei hi_there"

File.open("hi_there",'w') { |fw|
  fw.puts("Hi there 99")
}

puts "Wir schreiben 'Hi there 00' an das Ende der Datei hi_there"
File.open("hi_there",'a') { |fw|
  fw.puts("Hi there 00")
}

puts "Wir lesen von der Datei hi_there"

File.open("hi_there",'r') { |fw|
  while line = fw.gets()
    puts line
  end
}
```

Zusammenfassung

- Zeichenketten sind Folgen von Einzelzeichen. Zeichen werden intern als ganzzahliger Wert repräsentiert.
- Zeichenketten sind Objekte der Klasse ***String.***
- Stringliterale erzeugen ***String***-Objekte.
- Stringliterale mit doppelten Anführungs-strichen werden ausgewertet.
- Zeichenketten sind veränderbar. Die destruktiven Methoden ändern den Inhalt eines ***String***-Objektes.
- Die Verarbeitung von Zeichenketten ist eine der Stärken von Ruby. Es gibt daher eine große Menge von Methoden zur Manipulation von Strings.