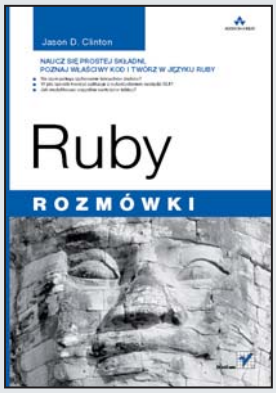
Ruby. Rozmówki



Autor: Jason D. Clinton
Tłumaczenie: Anna Trojan
ISBN: 978-83-246-1053-2
Tytuł oryginału: Ruby Phrasebook
Format: 115x170, stron: 240

Naucz się prostej składni, poznaj właściwy kod i twórz w języku Ruby

- Na czym polega szyfrowanie łańcuchów znaków?
- W jaki sposób tworzyć aplikacje z wykorzystaniem narzędzi GUI?
- Jak modyfikować wszystkie wartości w tablicy?

Ruby — jeden z najszybciej rozwijających się języków programowania — stał się niezwykle popularny dzięki takim technologiom, jak Ruby on Rails. Programy pisane w Ruby są małe i można je łatwo przenosić pomiędzy platformami. Dodatkowo prosta, obiektowa składnia, zwarty kod i elastyczność sprawiają, że możesz szybko nauczyć się pisać aplikacje przy użyciu tego języka, szczególnie jeśli korzystasz z tej książki, stanowiącej podręczny zbiór najbardziej przydatnych jego konstrukcji.

Książka „Ruby. Rozmówki" udostępnia gotowe fragmenty kodu, potrzebne do szybkiego i wydajnego tworzenia projektów programistycznych. Łatwo możesz dopasować je do własnych potrzeb. Zawartość podręcznika została uszeregowana tematycznie, a więc w każdej chwili będziesz mógł dowiedzieć się, na czym polega praca z kolekcjami, obiektami czy potokami. Dzięki temu poradnikowi nauczysz się również przetwarzać tekst, sprawdzać poprawność kodu XML oraz budować aplikacje z wykorzystaniem zestawów narzędzi GUI.

- Konwersja między typami
- Praca z łańcuchami znaków
- Praca z Unicode
- Tworzenie klas wyliczeniowych
- Badanie obiektów i klas
- Praca z zagnieżdżonymi zbiorami
- Ustalanie interaktywnych potoków standardowych
- Przechwytywanie danych wyjściowych procesu potomnego
- Jednowierszowce w Ruby
- Praca z XML
- Gniazda i wątki
- Bazy danych
- Dokumentacja Ruby
- Tworzenie prostego przypadku testowego

Po co wyważać otwarte drzwi? Skorzystaj z gotowych fragmentów kodu Ruby

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

2

Praca z łańcuchami znaków

Łączenie ze sobą aplikacji to to, do czego szczególnie dobrze nadają się takie języki programowania jak Ruby czy Perl — ich możliwości przetwarzania tekstu są doskonałe. Bez względu na to, czy chodzi o przetwarzanie plików konfiguracyjnych, udostępnianie stron internetowych, czy przechwytywanie danych wyjściowych programu, zawsze w grę wchodzi praca z łańcuchami znaków. Ruby i Perl różnią się jednak stopniem zorientowania obiektowego. W Ruby łańcuchy znaków mają własne metody składowe, które można wykorzystać do wykonywania każdej z funkcji przetwarzających tekst. Należy o tym pamiętać.

Poniższy podrozdział oraz dwa kolejne są ze sobą ściśle powiązane. Najpierw krótko przedstawię proste funkcje wyszukujące przyjmujące łańcuchy znaków jako parametry, a następnie przejdę do wyrażeń regularnych.

Wyszukiwanie w łańcuchach znaków

```
'foobar'.include? 'fo'
   #=> true
'foobar'['fo']
   #=> "fo "(true, ponieważ nie jest nil ani false)
'foobar'.count 'ob' # Przyjmuje się, że "ob" jest listą znaków
   #=> 3 (dwa "o" i jedno "b")
'foobar'.count 'ob', 'o'
   #=> 2 (jedynie "o" znajduje się w obu parametrach)
'foobar'.index 'ob'
   #=> 2
'foobar'.index 98 # 98 to kod ASCII dla "b"
   #=> 3
```

W pierwszych dwóch przykładach wykorzystano `#include?` w celu sprawdzenia, czy łańcuch znaków zawiera podłańcuch, oraz zwrócenia wyniku Boolean. Zostało to zaimplementowane w kodzie języka C, przez co jest nieco szybsze od wyszukiwania opartego na wyrażeniach regularnych (niewiele, ale szybsze).

W dwóch środkowych przykładach metodę `#count` wykorzystano do zwrócenia liczby wystąpień określonego łańcucha znaków. Odnaleziona zostaje część wspólna dla dodatkowych parametrów tej metody (czyli powtarzające się znaki). *Część wspólna* to pojęcie z logiki zbiorów. Oznacza, że uwzględnia się jedynie te elementy, które pojawiają się w obu zbiorach. W tym przypadku „elementem" jest znak. Metoda ta nie w pełni działa jeszcze ze znakami Unicode.

Wreszcie w ostatnich dwóch przykładach metoda `#index` zwraca pozycję — zgodnie z notacją powodującą obliczanie od zera — pierwszego wystąpienia łańcucha znaków bądź też liczbowy kod znaku. By odnaleźć pozycję ostatniego wystąpienia, należy użyć metody `#rindex`.

Wyszukując, można również przejść łańcuch znaków, plik lub strumień wejścia-wyjścia (obiekt klasy `IOStream`). Powiedzmy, że chcemy napisać prosty analizator składniowy plików konfiguracyjnych, który umieszcza każdą zmienną konfiguracyjną wraz z jej wartością w tablicy asocjacyjnej.

```
'a = 1\nb = 1\nc = 3\n'.each_line() { |line|
   if line.include?('=')
      # zrób coś z wierszem
   end
}
```

Kod umieszczony po znaku # wykonywany jest tylko wtedy, gdy w wierszu obecny będzie znak =. Bardziej rozbudowany przykład takiego rozwiązania można znaleźć w podrozdziale „Analiza składniowa prostego pliku konfiguracyjnego” w rozdziale 7., „Przetwarzanie tekstu”.

Wyszukiwanie w łańcuchach znaków za pomocą wyrażeń regularnych

```
"Teraz jest: 12:34:54\n".match
/(\d{2}):(\d{2}):(\d{2})/
   #=> #<MatchData:0x402e9548>
```

```
$1
   #=> "12"
$2
   #=> "34"
$3
   #=> "54"
```

Zagadnienie to jest bardzo rozbudowane — wyrażeniom regularnym poświęcono całe książki. Chyba najlepszą z nich jest *Mastering Regular Expressions* autorstwa Jeffreya E. F. Friedla (O'Reilly 2002). Każdej osobie pracującej z czystym tekstem polecam zakup tej pozycji i trzymanie jej zawsze pod ręką.

W niniejszym tekście omówię jedynie wykorzystywanie obiektów `Regexp` do wyszukiwania w tekście za pomocą języka Ruby; nie będzie to omówienie języka wyrażeń regularnych.

Kiedy w Ruby wykonuje się dopasowanie za pomocą obiektu wyrażenia regularnego, dowolna grupa w nawiasach znajdująca się wewnątrz obiektu `Regexp` ustawiana jest na zmienną globalną (lokalną dla wątku) od `$1` do `$9` w takiej kolejności, w jakiej grupy pojawiają się w obiekcie.

Metoda `#match` zwraca obiekt `MatchData` *dla pierwszego dopasowania* w łańcuchu znaków, a także ustawia zmienną globalną `$~` na ten sam obiekt. A oto, co zawiera obiekt `MatchData`.

```
m = "Teraz jest: 12:34:54\n".match
/(\d{2}):(\d{2}):(\d{2})/
m.to_a
   #=> ["12:34:54", "12", "34", "54"]
```

```
m.pre_match
   #=> "Teraz jest: "
m.post_match
   #=> "\n"
```

Dostęp do pełnego dopasowania można uzyskać za pomocą `m[0]`, natomiast do każdej z jego części — za pomocą `m[1]` do `m[9]`.

Jeśli chcemy zwrócić *wszystkie* dopasowania w łańcuchu znaków, należy użyć metody `.scan`. Zwraca ona obiekt klasy `Array` wyglądający następująco:

```
"Teraz jest: 12:34:54\n".scan(/\d{2}/)
   #=> ["12", "34", "54"]
```

Jeśli w wyrażeniu regularnym użyjemy grup w nawiasach, kod ten zwróci zagnieżdżone tablice.

W Ruby możliwe jest również użycie metody `.split` dla łańcuchów znaków w połączeniu z wyrażeniami regularnymi.

```
"Teraz jest: 12:34:54\n".split(/:\s/)
   #=> ["Teraz jest", "12:34:54\n"]
```

Zastępowanie podłańcuchów znaków

```
s = 'foobar'
s[-1] = 'z'
   s #=> "foobaz"
s[0,4] = 'ja'
```

Zastępowanie podłańcuchów znaków

```
   s #=> "jaaz"
s[2] = 122
   s #=> "jazz"

ary = ['jakaś_zmienna','jakaś_wartość']
"W opcji #{ary[0]} jest obecnie ustawiona
➥#{ary[1]}."
   #=> "W opcji jakaś_zmienna jest obecnie ustawiona
   ➥jakaś_wartość"

"To, na co teraz patrzysz, to #{ary[0].tr('_', ' ')}."
   #=>"To, na co teraz patrzysz, to jakaś zmienna."
```

Jak wspomniano w podrozdziale „Z łańcucha znaków na tablicę i z powrotem" w rozdziale 1., „Konwersja między typami", łańcuchy znaków można w wielu aspektach traktować jak tablice znaków. Obejmuje to również zastępowanie, takie jak powyżej.

W dwóch ostatnich przykładach umieszczony w cudzysłowie łańcuch znaków wykorzystany został do analizy znajdujących się wewnątrz niego sekwencji `"#{}"` oraz interpolacji wyników tych obliczeń. Wewnątrz sekwencji `"#{}"` można zrobić prawie wszystko. W powyższych przykładach zaprezentowano dostęp do tablicy, a nawet wywołanie metody `#tr`.

Nieco bardziej niejednoznaczna funkcja `#tr` pozwala wykonywać zastępowanie według znaków.

```
'Teraz patrzysz na łańcuch znaków.'.tr 'aeiou', '_'
   #=> "T_r_z p_trzysz n_ ł_ńc_ch zn_ków."
'Teraz patrzysz na łańcuch znaków.'.tr 'aeiou', 'uoiea'
   #=> "Toruz putrzysz nu łuńcach znuków."
```

Pierwszy przykład zastępuje samogłoski `a`, `e`, `i`, `o` oraz `u` znakiem `_`. W drugim podmienia każdą z tych samogłosek na inną. Kolejny przykład wykorzystywania metody `#tr` wraz z omówieniem działania parametrów tej metody znajduje się w podrozdziale „Oczyszczanie danych wejściowych".

Zastępowanie podłańcuchów znaków za pomocą sprintf

```
'Otrzymałem %25s' % 'łańcuch znaków.'
   #=> "Otrzymałem              łańcuch znaków."
'Otrzymałem %-25s' % 'łańcuch znaków.'
   #=> "Otrzymałem łańcuch znaków. "
'Otrzymałem kilka łańcuchów znaków: %s, %s' %
['jeden', 'dwa']
   #=> "Otrzymałem kilka łańcuchów znaków: jeden, dwa"

'Otrzymałem %25p' % [['jedną', 'tablicę']]
   #=> ""Otrzymałem        [\"jedną\", \"tablicę\"]"
'Otrzymałem %-25p' % [['jedną', 'tablicę']]
   #=> "Otrzymałem [\"jedną\", \"tablicę\"]
```

Tak samo jak w przypadku liczb, można również zastosować formatowanie w stylu `sprintf` z operatorem `%`. W tabeli 2.1 przedstawiono dozwolone argumenty.

W tabeli 2.2 zaprezentowano, co robią te argumenty.

Powinienem również krótko wspomnieć o nieco nietypowej opcji `sprintf` — możliwości użycia notacji pozycyjnej (*`pozycja`*)`$` w celu uzyskania dostępu do określonego

Tabela 2.1. Kody sprintf dla łańcuchów znaków

Argument łańcucha znaków	Dozwolony argument	Wyjaśnienie
`c`	*, -	Oczekuje obiektu klasy `Fixnum` reprezentującego kod znaku.
`s`	*,	Oczekuje obiektu klasy `String`.
`p`	*,	Dowolny obiekt odpowiadający na metodę `.inspect()`.

Tabela 2.2. Argumenty sprintf dla łańcuchów znaków

Argument	Wyjaśnienie
`*d`	`d` musi być liczbą całkowitą. Określa szerokość pola.
-	Wyrównanie do lewej.

wpisu w podanej tablicy. Nieco denerwujące jest jednak to, że notacja ta rozpoczyna odliczanie od `1`, a nie od `0` (tak jak w przypadku obiektów klasy `Array`).

```
'Witaj, %1$s! Dzisiaj kończysz %2$d lat! Wszystkiego
➥najlepszego, %1$s!' % ['Amadeusz', 13]
   #=> "Witaj, Amadeusz! Dzisiaj kończysz 13 lat!
   ➥Wszystkiego najlepszego, Amadeusz!"
```

Zastępowanie podłańcuchów znaków za pomocą wyrażeń regularnych

```
'W tej chwili jest:
➥12:34:21'.sub(/(\d\d):(\d\d):(\d\d)/, '\1\2\3')
   #=> "W tej chwili jest: 123421"
```

I znów wyrażenia regularne mogą nam się bardzo przydać. By dokonać interpolacji wyników dopasowania z grup w nawiasach, należy użyć `\1` do `\9`. By zastąpić wszystkie wystąpienia w łańcuchu znaków, należy użyć `.gsub` zamiast `.sub`.

W powyższym przykładzie całe wyrażenie `12:34:21` zostało dopasowane i zastąpione podgrupami od `1` do `3` bez rozdzielających je dwukropków. Alternatywnie można również po prostu zastąpić każde wystąpienie znaku : pojawiające się między cyframi.

Można by to było zapisać w następujący sposób:

```
'W tej chwili jest: 12:34:21'.gsub(/(\d):
➥(\d)/, '\1\2')
   #=> "W tej chwili jest: 123421"
```

Praca z Unicode

```
#!/usr/bin/ruby -wKu

'      '.scan(/./) { |b| print b, ' ' }
```

Zwraca:

Ruby przyjmuje pliki źródłowe z kodowaniem UTF-8. W celu upewnienia się, że wszystko będzie poprawnie interpretowane w innych systemach operacyjnych oraz przy innych ustawieniach regionalnych, należy dodać opcję wiersza poleceń `-Ku` do wiersza shebang[1]. Zmodyfikowany wiersz shebang to pierwszy wiersz powyższego fragmentu kodu.

Sam Ruby nie jest w pełni zinternacjonalizowany. Ruby nie jest na przykład świadomy wielobajtowej natury UTF-8 poza pierwszymi 255 kodami znaków. Z tego powodu, gdy mamy do czynienia z umiędzynarodowionymi łańcuchami znaków, z metody `#each_byte` należy korzystać z rozwagą. Nie będzie ona na przykład działała na językach niełacińskich.

```
puts "      " # dane wyjściowe bezpośrednio do bufora
"      ".each_byte { |b| print(""<<b, ' ') }
```

[1] Wiersz shebang to inaczej pierwszy wiersz programu rozpoczynający się od znaków `#!` i wskazujący powłoce systemów *nix, jakiego interpretera należy użyć do wykonania poleceń programu — *przyp. tłum.*

Zwraca:

```
? ? ? ? ? ?
```

By przejść takie znaki, należy zamiast tego skorzystać z wyrażeń regularnych (działają one w UTF-8). Kod umożliwiający to znajduje się na początku podrozdziału.

Kiedy pracuje się z Unicode, do tworzenia wycinków oraz dopasowywania należy wykorzystać wyrażenia regularne. Poza tym wyjątkiem, łańcuchy znaków z zawartością o kodowaniu Unicode powinny się zachować dokładnie zgodnie z oczekiwaniami.

Oczyszczanie danych wejściowych

```
new_password = gets
if new_password.count '^A-Za-z._' != 0 then
   puts "Złe hasło"
else
   # zrób coś jak w podrozdziale "Szyfrowanie łańcucha znaków"
end
```

Powiedzmy, że chcemy napisać program zmieniający hasła przeznaczony dla systemu *nix (być może korzystający z serwera LDAP). Po ukazaniu się prośby o zalogowanie się, w haśle można użyć *prawie każdego* znaku, jaki da się wygenerować na klawiaturze. Kilka znaków, których nie można użyć, może jednak przyprawić użytkowników o ból głowy, kiedy odkryją, że po zmianie hasła nie po-

trafią się już zalogować. By ułatwić sobie życie, możemy napisać program zmieniający hasła, który ograniczy hasło do znaków alfanumerycznych i kilku im podobnych. Metoda `String#count`, zastosowana jak powyżej, może nam w tym pomóc.

Rozwiązanie to działa dzięki użyciu specjalnej składni współdzielonej przez `.count`, `.tr`, `delete` oraz `squeeze`. Parametr zaczynający się od znaku `^` powoduje negację listy. Lista składa się z dowolnych poprawnych znaków w bieżącym zbiorze znaków i może zawierać zakresy utworzone za pomocą znaku -. Jeśli do funkcji tych przekazana zostanie więcej niż jedna lista, wykorzystuje się część wspólną list znaków zgodnie z logiką zbiorów — co oznacza, że w filtrowaniu użyte zostaną jedynie znaki występujące w obu listach.

W przypadku innego rodzaju działań oczyszczających można po prostu zastąpić wszystkie niewłaściwe znaki znakiem _ (jak na przykład w formularzu CGI).

```
evil_input = '`cat /etc/passwd`'
evil_input.tr('./\`', '_')
   #=> "_cat _etc_passwd_"
```

Praca z końcami wierszy

Kiedy mamy do czynienia z tekstem z trzech głównych systemów operacyjnych, możemy się spotkać z najstarszym chyba sposobem dzielenia plików (tabela 2.3).

Tabela 2.3. Zakończenia wiersza specyficzne dla systemów operacyjnych

System operacyjny	Zakończenie wiersza
Mac OS 9 i starsze wersje	`\r`
Windows	`\r\n`
*nix	`\n`

Jeśli pracujemy z tekstem w systemach Windows oraz *nix, prawdopodobnie nie musimy się nad tym zastanawiać — są one obsługiwane w niemalże ten sam sposób.

```
"a\r\nb\r\nc\r\n".each_line { |line|
   puts(line.inspect)
}

Zwraca:
"a\r\n"
"b\r\n"
"c\r\n"

"a\r\nb\r\nc\r\n".each_line { |line|
   puts(line.chomp.inspect) # chomp bezpiecznie usuwa oba
}

Zwraca:
"a"
"b"
"c"
```

W systemie Linux wygląda to prawie identycznie.

```
"a\nb\nc\n".each_line { |line|
   puts(line.inspect)
}

Zwraca:
"a\n"
```

```
"h\n"
"c\n"
```

Dla plików systemu Mac OS konieczne jest określenie separatora.

```
"a\rb\rc\r".each_line "\r" { |line|
   puts(line.inspect)
}

Zwraca:
"a\r"
"b\r"
"c\r"
```

Przetwarzanie dużych łańcuchów znaków

```
my_string = ''
(2**21).times{ my_string << rand(256) }
   #=> 2097152 (2 MB losowych danych)

require 'stringio'
string_stream = StringIO.new my_string
string_stream.read 256
   #=> "\351@\300g\251\326\036\314| *\335jJ\017 ...
```

Jeśli tak duży łańcuch znaków znajduje się już w pamięci, najlepszy sposób pracy z nim uzależniony jest od tego, co chcemy z nim zrobić. Jeśli łańcuch ten jest czystym tekstem i chcemy go w jakiś sposób poddać analizie składniowej, należy skorzystać z metody `#each_line`. Przykład utworzenia tablicy asocjacyjnej z takiego łańcucha znaków znajduje się w podrozdziale „Budowanie tablicy asocja-

cyjnej z pliku konfiguracyjnego" w rozdziale 3. Jeśli jednak łańcuch znaków zawiera dane binarne, można tworzyć wycinki o określonej liczbie bajtów za każdym razem, jak widać to wyżej w przyrostach o 256 bajtów. W przykładzie tym za każdym razem, gdy wywoływana jest metoda `#read`, obiekt `Enumerator` (rodzaj znacznika pozycji) przesuwa się, tak by możliwe było śledzenie pozycji w łańcuchu znaków.

Porównywanie łańcuchów znaków

```
"Kto" <=> "kto"
   #=> -1
%w{'kto' 'jest' 'teraz' 'pierwszy?' 'Kto'}.sort
   #=> ["Kto", "jest", "kto", "pierwszy?", "teraz"]
'foobar'.casecmp 'Foobaz'
   #=> -1
'foobar'.casecmp 'FooBar'
   #=> 0
%w{'kto' 'jest' 'teraz' 'pierwszy?' 'Kto'}.sort {
➥|a,b| a.casecmp b }
   #=> ["jest", "kto", "Kto", "pierwszy?", "teraz"]
```

Łańcuchy znaków można porównywać z uwzględnieniem wielkości liter — tak właśnie domyślnie działa metoda `#sort`. Porównania bez uwzględnienia wielkości liter są teraz obsługiwane przez metodę `String#casecmp`. Metoda ta działa w taki sam sposób jak `<=>` (zwracając `-1`, `0` lub `1`), dlatego może być wywoływana jako część wywołań `#sort {}`.

Sprawdzanie sum kontrolnych łańcuchów znaków (MD5 lub inne metody)

```
require 'digest/md5'
digest = Digest::MD5.hexdigest("foobar")
puts digest
   #=> 3858f62230ac3c915f300c664312c63f
digest = Digest::SHA1.hexdigest("foobar")
puts digest
   #=> 8843d7f92416211de9ebb963ff4ce28125932878
```

W powyższym kodzie zaprezentowałem metody sprawdzania sum kontrolnych za pomocą algorytmów MD5 oraz SHA1. Należy pamiętać, że MD5 jest słabym zabezpieczeniem — z pewnym wysiłkiem ktoś może utworzyć plik o tej samej sumie kontrolnej, udający tym samym prawdziwy, poprawny plik. Zamiast tego rozwiązania można skorzystać z zaprezentowanego wyżej algorytmu SHA1.

Jeśli potrzebna jest nam suma kontrolna dla pliku, można wczytać cały plik do łańcucha znaków za pomocą metody `#read`, a następnie skorzystać z powyższych rozwiązań. Jest to jednak niezbyt dobry pomysł, jeśli mamy do czynienia z dużym plikiem. Zamiast tego można rozważyć następujące rozwiązanie:

```
require 'digest/md5'

def md5sum_file path
   d = Digest::MD5.new
   File.open(path, 'r') do |fp|
```

```
    while buf = fp.read(1024*8)
      d << buf
    end
  end
  d.hexdigest
```

Powyższy przykład wykorzystuje metodę `#update` lub `#<<` z MD5 i nie wymaga wczytania całego pliku do pamięci.

Szyfrowanie łańcucha znaków

```
password = 'f00bar'

# wygenerowanie losowej wartości salt
salting_chars = ('A'..'Z').to_a + ('a'..'z').to_a +
['.', '/']
salt = salting_chars[rand(54)] +
salting_chars[rand(54)]
   #=> "jM"
password.crypt(salt)
   #=> "jM7qRC1ulBPhc"
```

Metoda `String#crypt` pozwala na wykonywanie jednostronnej funkcji skrótu na łańcuchu znaków. Można wykorzystać to rozwiązanie do zaimplementowania prostego bezpieczeństwa haseł. By przechować hasło za pierwszym razem, wybiera się losową wartość salt, a następnie wykorzystuje się ją do wykonania funkcji skrótu dla otrzymanego hasła. Warto zauważyć, że modyfikator klucza przechowywany jest na początku wartości skrótu (czyli w ostatnim wierszu powyższego przykładu). Wartość skrótu można wykorzystać do zweryfikowania kogoś, kto

próbuje użyć hasła. Ponieważ jest to szyfrowanie jednostronne, wpisywane hasło należy zaszyfrować z użyciem tej samej wartości salt i porównać oba zaszyfrowane łańcuchy znaków w celu przekonania się, czy są one równe.

```
input_password = 'fOObar'
crypted_password = 'jM7qRC1ulBPhc'

salt = crypted_password[0,2]
   #=> "jM"

# sprawdzanie hasła
input_password.crypt(salt) == crypted_password
```

Gdybyśmy chcieli zaimplementować szyfrowanie dwukierunkowe (wymagające klucza do odszyfrowania tekstu), w standardowej bibliotece języka Ruby nie znajdziemy niestety odpowiedniej klasy czy metody. Można rozważyć użycie modułu Ruby-AES z *Ruby Application Archive* (RAA). W niektórych systemach operacyjnych można również wykorzystać dane wyjściowe z polecenia podprocesu *aesloop*.