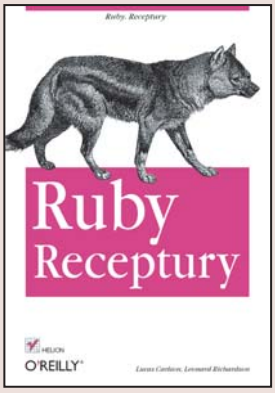
Ruby. Receptury



Autorzy: Lucas Carlson, Leonard Richardson
Tłumaczenie: Andrzej Grażyński, Radosław Meryk
ISBN: 83-246-0768-4
Tytuł oryginału: Ruby Cookbook
Format: B5, stron: 888

Zbiór gotowych rozwiązań dla programistów używających języka Ruby

- Jak przetwarzać pliki XML i HTML?
- Jak wykorzystywać środowisko Ruby on Rails?
- W jaki sposób łączyć Ruby z technologią AJAX?

Korzystasz w pracy z języka Ruby i zastanawiasz się, czy niektóre zadania programistyczne można wykonać szybciej? Chcesz poznać zasady programowania obiektowego w Ruby? A może interesuje Cię framework Ruby on Rails? Język Ruby zdobywa coraz większą popularność, jest wykorzystywany do tworzenia aplikacji sieciowych i stał się podstawą środowiska Ruby on Rails. Jednak nawet najlepszy język programowania nie uwalnia programistów od żmudnego realizowania zadań, które nie mają zbyt wiele wspólnego z tworzeniem aplikacji, czyli usuwania błędów, implementowania typowych algorytmów, poszukiwania rozwiązań mniej lub bardziej typowych problemów i wielu innych.

Książka „Ruby. Receptury" znacznie przyspieszy Twoją pracę. Znajdziesz tu kilkaset praktycznych rozwiązań problemów wraz z przejrzystym komentarzem oraz tysiące wierszy proponowanego kodu, który będziesz mógł wykorzystać w swoich projektach. Przeczytasz o strukturach danych, algorytmach, przetwarzaniu plików XML i HTML, tworzeniu interfejsów użytkownika dla aplikacji i połączeniach z bazami danych. Nauczysz się generować i obrabiać pliki graficzne, korzystać z usług sieciowych, wyszukiwać i usuwać błędy w aplikacjach, a także pisać skrypty niezwykle pomocne w administrowaniu systemem operacyjnym Linux.

- Przetwarzanie danych tekstowych i liczbowych
- Operacje na tablicach
- Praca z systemem plików
- Programowanie obiektowe
- Przetwarzanie dokumentów XML i HTML oraz plików graficznych
- Generowanie plików PDF
- Połączenie z bazami danych
- Korzystanie z poczty elektronicznej, protokołu telnet i połączeń Torrent
- Projektowanie aplikacji internetowych za pomocą Ruby on Rails
- Stosowanie usług sieciowych
- Optymalizacja aplikacji
- Tworzenie wersji dystrybucyjnych
- Automatyzacja zadań z wykorzystaniem języka Rake
- Budowanie interfejsów użytkownika

Jeśli chcesz rozwiązać problem, skorzystaj z gotowej receptury — koło już wynaleziono

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

Spis treści

ROZDZIAŁ 1.

Łańcuchy

Ruby jest językiem przyjaznym programiście. Przed programistami hołdującymi filozofii programowania zorientowanego obiektowo odkryje on drugą jego naturę; programiści stroniący od obiektów nie powinni mieć natomiast większych trudności, bowiem — w odróżnieniu od wielu innych języków — w języku Ruby stosuje się zwięzłe i konsekwentne nazewnictwo metod, które generalnie zachowują się tak, jak (intuicyjnie) można by tego oczekiwać.

Łańcuchy znakomicie nadają się na obszar „pierwszego kontaktu" z językiem Ruby: są użyteczne, łatwo się je tworzy i wykorzystuje, występują w większości języków, a więc służyć mogą zarówno jako materiał porównawczy, jak i okazja do przedstawienia koncepcyjnych nowości języka Ruby w rodzaju *duck typing* (receptura 1.12), otwartych klas (receptura 1.10), symboli (receptura 1.7), a nawet gemów (receptura 1.20).

Omawiane koncepcje ilustrujemy konsekwentnie interaktywnymi sesjami języka Ruby. W środowisku Uniksa i Mac OS X służy do tego program `irb`, uruchamiany z wiersza poleceń. Użytkownicy Windows mogą także wykorzystywać w tym celu program `fxri`, dostępny za pomocą menu Start (po zainstalowaniu środowiska Ruby za pomocą pakietu „one-click installer", który pobrać można spod adresu *http:/rubyforge.org/projects/rubyinstaller*). Program `irb` jest również dostępny w Windows. Wspomniane programy tworzą po uruchomieniu interaktywną powłokę języka Ruby, pod kontrolą której wykonywać można fragmenty przykładowego kodu.

Łańcuchy języka Ruby podobne są do łańcuchów w innych „dynamicznych" językach — Perlu, Pythonie czy PHP. Nie różnią się zbytnio od łańcuchów znanych z języków C i Java. Są dynamiczne, elastyczne i modyfikowalne.

Rozpocznijmy więc naszą sesję, wpisując do wiersza poleceń powłoki następujący tekst:

```
string = "To jest napis"
```

Spowoduje to wyświetlenie rezultatu wykonania polecenia:

```
=> "To jest napis"
```

Polecenie to powoduje utworzenie łańcucha `"To jest napis"` i przypisanie go zmiennej o nazwie `string`. Łańcuch ten staje się więc wartością zmiennej i jednocześnie wartością całego wyrażenia, co uwidocznione zostaje w postaci wyniku wypisywanego (po strzałce `=>`) w ramach interaktywnej sesji. W treści książki zapisywać będziemy ten rodzaj interakcji w postaci

```
string "To jest napis"          => "To jest napis"
```

W języku Ruby wszystko, co można przypisać zmiennej, jest obiektem. W powyższym przykładzie zmienna `string` wskazuje na obiekt klasy `String`. Klasa ta definiuje ponad sto metod

— nazwanych fragmentów kodu służących do wykonywania rozmaitych operacji na łańcuchach. Wiele z tych metod wykorzystywać będziemy w naszej książce, także w niniejszym rozdziale. Jedna z tych metod — `String#length` — zwraca rozmiar łańcucha, czyli liczbę składających się na niego bajtów:

```
string.length                   => 13
```

W wielu językach programowania wymagana (lub dopuszczalna) jest para nawiasów po nazwie wywoływanej metody:

```
string.length()                 => 13
```

W języku Ruby nawiasy te są niemal zawsze nieobowiązkowe, szczególnie w sytuacji, gdy do wywoływanej metody nie są przekazywane żadne parametry (jak w powyższym przykładzie). Gdy parametry takie są przekazywane, użycie nawiasów może uczynić całą konstrukcję bardziej czytelną:

```
string.count 's'                => 2       # s występuje dwukrotnie
string.count('s')               => 2
```

Wartość zwracana przez metodę sama z siebie jest obiektem. W przypadku metody `String#length`, wywoływanej w powyższym przykładzie, obiekt ten jest liczbą `20`, czyli egzemplarzem (instancją) klasy `Fixnum`. Ponieważ jest obiektem, można na jego rzecz także wywoływać metody:

```
string.length.next              => 14
```

Weźmy teraz pod uwagę bardziej ciekawy przypadek — łańcuch zawierający znaki spoza kodu ASCII. Poniższy łańcuch reprezentuje francuskie zdanie „il était une fois” zakodowane według UTF-8[1]:

```
french_string = "il \xc3\xa9tait une fois"   # => "il \303\251tait une fois"
```

Wiele języków programowania, między innymi Java, traktuje łańcuchy jako ciągi *znaków*. W języku Ruby łańcuch postrzegany jest jako ciąg *bajtów*. Ponieważ powyższy łańcuch zawiera 14 liter i 3 spacje, można by domniemywać, że jego długość wynosi 17; ponieważ jedna z liter jest znakiem dwubajtowym, łańcuch składa się z 18, nie 17 bajtów:

```
french_string.length                         # => 18
```

Do różnorodnego kodowania znaków powrócimy w recepturach 1.14 i 11.12; specyfiką łańcuchów zawierających znaki wielobajtowe zajmiemy się także w recepturze 1.8.

Znaki specjalne (tak jak binarne dane w łańcuchu `french_string`) mogą być reprezentowane za pomocą tzw. sekwencji unikowych (*escaping*). Ruby udostępnia kilka rodzajów takich sekwencji w zależności od tego, w jaki sposób tworzony jest dany łańcuch. Jeżeli mianowicie łańcuch ujęty jest w *cudzysłów* (`" ... "`), można w nim kodować zarówno znaki binarne (jak w przykładowym łańcuchu francuskim), jak i znak nowego wiersza `\n` znany z wielu innych języków:

```
puts "Ten łańcuch\nzawiera znak nowego wiersza"
# Ten łańcuch
# zawiera znak nowego wiersza
```

W łańcuchu zamkniętym znakami apostrofu (`' ... '`) jedynym dopuszczalnym znakiem specjalnym jest odwrotny ukośnik `\` (*backslash*), umożliwiający reprezentowanie pojedynczego znaku specjalnego; para `\\` reprezentuje pojedynczy *backslash*.

[1] `"\xc3\xa9"` jest zapisem w języku Ruby unikodowego znaku é w reprezentacji UTF-8.

```
puts 'Ten łańcuch wbrew pozorom \nniezawiera znaku nowego wiersza'
# Ten łańcuch wbrew pozorom \nniezawiera znaku nowego wiersza

puts 'To jest odwrotny ukośnik: \\'
# To jest odwrotny ukośnik: \
```

Do kwestii tej powrócimy w recepturze 1.5, a w recepturach 1.2 i 1.3 zajmiemy się bardziej spektakularnymi możliwościami łańcuchów ograniczonych apostrofami.

Oto inna użyteczna możliwość inicjowania łańcucha, nazywana w języku Ruby *here documents*:

```
long_string = <<EOF
To jest długi łańcuch
Składający się z kilku akapitów
EOF
# => "To jest długi łańcuch\nSkładający się z kilku akapitów\n"

puts long_string
# To jest długi łańcuch
# Składający się z kilku akapitów
```

Podobnie jak w przypadku większości wbudowanych klas języka Ruby, także w przypadku łańcuchów można kodować tę samą funkcjonalność na wiele różnych sposobów („idiomów") i programista może dokonać wyboru tego, który odpowiada mu najbardziej. Weźmy jako przykład ekstrakcję podłańcucha z długiego łańcucha: programiści preferujący podejście obiektowe zapewne użyliby do tego celu metody `String#slice`:

```
string                  # "To jest napis"
string.slice(3,4)       # "jest"
```

Programiści wywodzący swe nawyki z języka C skłonni są jednak do traktowania łańcuchów jako tablic bajtów — im także Ruby wychodzi naprzeciw, umożliwiając ekstrakcję poszczególnych bajtów łańcucha:

```
string[3].chr + string[4].chr + string[5].chr + string[6].chr
# => "jest"
```

Podobnie proste zadanie mają programiści przywykli do Pythona:

```
string[3, 4]            # => "jest"
```

W przeciwieństwie do wielu innych języków programowania, łańcuchy Ruby są modyfikowalne (*mutable*) — można je zmieniać już po zadeklarowaniu. Oto wynik działania dwóch metod: `String#upcase` i `String#upcase!` — zwróć uwagę na istotną różnicę między nimi:

```
string.upcase           # => "TO JEST NAPIS"
string                  # => "To jest napis"

string.upcase!          # => "TO JEST NAPIS"
string                  # => "TO JEST NAPIS"
```

Przy okazji widoczna staje się jedna z konwencji składniowych języka Ruby: metody „niebezpieczne" — czyli głównie te modyfikujące obiekty „w miejscu" — opatrywane są nazwami kończącymi się wykrzyknikiem. Inna konwencja syntaktyczna związana jest z *predykatami*, czyli metodami zwracającymi wartość `true` albo `false` — ich nazwy kończą się znakiem zapytania.

```
string.empty?           # => false
string.include? "To"    # => true
```

Użycie znaków przestankowych charakterystycznych dla języka potocznego, w celu uczynienia kodu bardziej czytelnym dla programisty, jest odzwierciedleniem filozofii twórcy języka Ruby, Yukihiro „Matza" Matsumoto, zgodnie z którą to filozofią Ruby powinien być czytelny

przede wszystkim dla ludzi, zaś możliwość wykonywania zapisanych w nim programów przez interpreter jest kwestią wtórną.

Interaktywne sesje języka Ruby są niezastąpionym narzędziem umożliwiającym poznawanie metod języka i praktyczne eksperymentowanie z nimi. Ponownie zachęcamy do osobistego sprawdzania prezentowanego kodu w ramach sesji programu `irb` lub `fxri`, a z biegiem czasu tworzenia i testowania także własnych przykładów.

Dodatkowe informacje na temat łańcuchów Ruby można uzyskać z następujących źródeł:

- Informację o dowolnej wbudowanej metodzie języka można otrzymać wprost w oknie programu `fxri`, wybierając odnośną pozycję w lewym panelu. W programie `irb` można użyć w tym celu polecenia `ri` — na przykład informację o metodzie `String#upcase!` uzyskamy za pomocą polecenia

  ```
  ri String#upcase!
  ```

- *why the lucky stiff* napisał wspaniałe wprowadzenie do instalacji języka Ruby oraz wykorzystywania poleceń `ir` i `irb`. Jest ono dostępne pod adresem *http://poignantguide.net/ruby/expansion-pak-1.html*.
- Filozofię projektową języka Ruby przedstawia jego autor, Yukihiro „Matz" Matsumoto, w wywiadzie dostępnym pod adresem *http://www.artima.com/intv/ruby.html*.

1.1. Budowanie łańcucha z części

Problem

Iterując po strukturze danych, należy zbudować łańcuch reprezentujący kolejne kroki tej iteracji.

Rozwiązanie

Istnieją dwa efektywne rozwiązania tego problemu. W najprostszym przypadku rozpoczynamy od łańcucha pustego, sukcesywnie dołączając do niego podłańcuchy za pomocą operatora <<:

```
hash = { "key1" => "val1", "key2" => "val2" }
string = ""
hash.each { |k,v| string << "#{k} is #{v}\n" }
puts string
# key1 is val1
# key2 is val2
```

Poniższa odmiana tego prostego rozwiązania jest nieco efektywniejsza, chociaż mniej czytelna:

```
string = ""
hash.each { |k,v| string << k << " is " << v << "\n" }
```

Jeśli wspomnianą strukturą danych jest tablica, bądź też struktura ta daje się łatwo przetransformować na tablicę, zwykle bardziej efektywne rozwiązanie można uzyskać za pomocą metody `Array#join`:

```
puts hash.keys.join("\n") + "\n"
# key1
# key2
```

Dyskusja

W językach takich jak Pyton czy Java sukcesywne dołączanie podłańcuchów do pustego początkowo łańcucha jest rozwiązaniem bardzo nieefektywnym. Łańcuchy w tych językach są niemodyfikowalne (*immutable*), a więc każdorazowe dołączenie podłańcucha wiąże się ze stworzeniem nowego obiektu. Dołączanie serii podłańcuchów oznacza więc tworzenie dużej liczby obiektów pośrednich, z których każdy stanowi jedynie „pomost" do następnego etapu. W praktyce przekłada się to na marnotrawstwo czasu i pamięci.

W tych warunkach rozwiązaniem najbardziej efektywnym byłoby zapisanie poszczególnych podłańcuchów w tablicy (lub innej modyfikowalnej strukturze) zdolnej do dynamicznego rozszerzania się. Gdy wszystkie podłańcuchy zostaną już zmagazynowane we wspomnianej tablicy, można połączyć je w pojedynczy łańcuch za pomocą operatora stanowiącego odpowiednik metody `Array#join` języka Ruby. W języku Java zadanie to spełnia klasa `StringBuffer`. Unikamy w ten sposób tworzenia wspomnianych obiektów pośrednich.

W języku Ruby sprawa ma się zgoła inaczej, bo łańcuchy są tu modyfikowalne, podobnie jak tablice. Mogą więc być rozszerzane w miarę potrzeby, bez zbytniego obciążania pamięci lub procesora. W najszybszym wariancie rozwiązania możemy więc w ogóle zapomnieć o tablicy pośredniczącej i umieszczać poszczególne podłańcuchy bezpośrednio wewnątrz łańcucha docelowego. Niekiedy skorzystanie z `Array#join` okazuje się szybsze, lecz zwykle niewiele szybsze, ponadto konstrukcja oparta na << jest generalnie łatwiejsza do zrozumienia.

W sytuacji, gdy efektywność jest czynnikiem krytycznym, nie należy tworzyć nowych łańcuchów, jeżeli możliwe jest dołączanie podłańcuchów do łańcucha istniejącego. Konstrukcje w rodzaju

```
str << 'a' + 'b'
```

czy

```
str << "#{var1} #{var2}"
```

powodują tworzenie nowych łańcuchów, które natychmiast „podłączane" są do większego łańcucha — a tego właśnie chcielibyśmy unikać. Umożliwia nam to konstrukcja

```
str << var1 << ' ' << var2
```

Z drugiej jednak strony, nie powinno się modyfikować łańcuchów nietworzonych przez siebie i względy bezpieczeństwa przemawiają za tworzeniem nowego łańcucha. Gdy definiujesz metodę otrzymującą łańcuch jako parametr, metoda ta nie powinna modyfikować owego łańcucha przez dołączanie podłańcuchów na jego końcu — chyba że właśnie to jest celem metody (której nazwa powinna tym samym kończyć się wykrzyknikiem, dla zwrócenia szczególnej uwagi osoby studiującej kod programu).

Przy okazji ważna uwaga: działanie metody `Array#join` nie jest dokładnie równoważne dołączaniu kolejnych podłańcuchów do łańcucha. `Array#join` akceptuje separator, który wstawiany jest między każde dwa sąsiednie elementy tablicy; w przeciwieństwie do sukcesywnego dołączania podłańcuchów, separator ten *nie jest umieszczany po ostatnim elemencie*. Różnicę tę ilustruje poniższy przykład:

```
data = ['1', '2', '3']
s = ''
data.each { |x| s << x << ' oraz '}
s                                       # => "1 oraz 2 oraz 3 oraz "

data.join(' oraz ')                     # => "1 oraz 2 oraz 3"
```

Aby zasymulować działanie `Array#join` za pomocą iteracji, można wykorzystać metodę `Enumerable#each_with_index`, opuszczając separator dla ostatniego indeksu. Da się to jednak zrobić tylko wówczas, gdy liczba elementów objętych enumeracją znana jest a priori:

```
s = ""
data.each_with_index { |x, i| s << x; s << "|" if i < data.length-1 }
s                                          # => "1|2|3"
```

1.2. Zastępowanie zmiennych w tworzonym łańcuchu

Problem

Należy stworzyć łańcuch zawierający reprezentację zmiennej lub wyrażenia języka Ruby.

Rozwiązanie

Należy wewnątrz łańcucha zamknąć zmienną lub wyrażenie w nawiasy klamrowe i poprzedzić tę konstrukcję znakiem # (*hash*).

```
liczba = 5
"Liczba jest równa #{liczba}."                    # => "Liczba jest równa 5."
"Liczba jest równa #{5}."                         # => "Liczba jest równa 5."
"Liczba następna po #{liczba} równa jest #{liczba.next}."
# => "Liczba następna po 5 równa jest 6."
"Liczba poprzedzająca #{liczba} równa jest #{liczba-1}."
# => "Liczba poprzedzająca 5 równa jest 4."
"To jest ##{number}!"                             # => "To jest #5!"
```

Dyskusja

Łańcuch ujęty w cudzysłów (`" ... "`) jest przez interpreter skanowany pod kątem obecności specjalnych kodów substytucyjnych. Jednym z najbardziej elementarnych i najczęściej używanych kodów tego typu jest znak `\n`, zastępowany znakiem nowego wiersza.

Oczywiście istnieją bardziej skomplikowane kody substytucyjne. W szczególności dowolny tekst zamknięty w nawiasy klamrowe poprzedzone znakiem # (czyli konstrukcja `#{tekst}`) interpretowany jest jako wyrażenie języka Ruby, a cała konstrukcja zastępowana jest w łańcuchu wartością tego wyrażenia. Jeżeli wartość ta nie jest łańcuchem, Ruby dokonuje jej konwersji na łańcuch za pomocą metody `to_s`. Proces ten nosi nazwę *interpolacji*.

Tak powstały łańcuch staje się nieodróżnialny od łańcucha, w którym interpolacji nie zastosowano:

```
"#{liczba}" == '5'                                # => true
```

Za pomocą interpolacji można umieszczać w łańcuchu nawet spore porcje tekstu. Przypadkiem ekstremalnym jest definiowanie klasy wewnątrz łańcucha i wykorzystanie podłańcucha stanowiącego wynik wykonania określonej metody tej klasy. Mimo ograniczonej raczej użyteczności tego mechanizmu, warto go zapamiętać jako dowód wspaniałych możliwości języka Ruby.

```
%{Tutaj jest #{class InstantClass
   def bar
      "pewien tekst"
    end
  end
```

```
  InstantClass.new.bar
}.}
# => "Tutaj jest pewien tekst."
```

Kod wykonywany w ramach interpolacji funkcjonuje dokładnie tak samo, jak każdy inny kod Ruby w tej samej lokalizacji. Definiowana w powyższym przykładzie klasa `InstantClass` nie różni się od innych klas i może być używana także na zewnątrz łańcucha.

Gdy w ramach interpolacji wywoływana jest metoda powodująca efekty uboczne, efekty te widoczne są na zewnątrz łańcucha. W szczególności, jeżeli efektem ubocznym jest nadanie wartości jakiejś zmiennej, zmienna ta zachowuje tę wartość na zewnątrz łańcucha. Mimo iż nie polecamy celowego polegania na tej własności, należy koniecznie mieć świadomość jej istnienia.

```
"Zmiennej x nadano wartość #{x = 5; x += 1}."   # => "Zmiennej x nadano wartość 6."
x                                               # => 6
```

Jeżeli chcielibyśmy potraktować występującą w łańcuchu sekwencję `#{tekst}` w sposób literalny, nie jako polecenie interpolacji, wystarczy poprzedzić ją znakiem odwrotnego ukośnika (\) albo zamknąć łańcuch znakami apostrofu zamiast cudzysłowu:

```
"\#{foo}"                                       # => "\#{foo}"
'#{foo}'                                        # => "\#{foo}"
```

Alternatywnym dla `%{}` kodem substytucyjnym jest konstrukcja *here document*. Pozwala ona na zdefiniowanie wielowierszowego łańcucha, którego ogranicznikiem jest wiersz o wyróżnionej postaci.

```
name = "Mr. Lorum"
email = <<END
Szanowny #{name},

Niestety, nie możemy pozytywnie rozpatrzyć Pańskiej reklamacji w związku
z oszacowaniem szkody, gdyż jesteśmy piekarnią, nie firmą ubezpieczeniową.

Podpisano,
 Buła, Rogal i Precel
 Piekarze Jej Królewskiej Wysokości
END
```

Język Ruby pozostawia programiście dużą swobodę w zakresie wyboru postaci wiersza ograniczającego:

```
<<koniec_wiersza
Pewien poeta z Afryki
Pisał dwuwierszowe limeryki
koniec_wiersza
# => "Pewien poeta z Afryki\nPisał dwuwierszowe limeryki\n"
```

Patrz także

- Za pomocą techniki opisanej w recepturze 1.3 można definiować łańcuchy i obiekty szablonowe, umożliwiające „odroczoną” interpolację.

1.3. Zastępowanie zmiennych w istniejącym łańcuchu

Problem

Należy utworzyć łańcuch umożliwiający interpolację wyrażenia języka Ruby, jednakże bez wykonywania tej interpolacji — ta wykonana zostanie później, prawdopodobnie wtedy, gdy będą znane wartości zastępowanych wyrażeń.

Rozwiązanie

Problem można rozwiązać za pomocą dwojakiego rodzaju środków: łańcuchów typu `printf` oraz szablonów ERB.

Ruby zapewnia wsparcie dla znanych z C i Pythona łańcuchów formatujących typu `printf`. Kody substytucyjne w ramach tych łańcuchów mają postać dyrektyw rozpoczynających się od znaku `%` (modulo):

```
template = 'Oceania zawsze była w stanie wojny z %s.'
template % 'Eurazją'
# => "Oceania zawsze była w stanie wojny z Eurazją."
template % 'Antarktydą'
# => "Oceania zawsze była w stanie wojny z Antarktydą."

'Z dwoma miejscami dziesiętnymi: %.2f' % Math::PI
# => " Z dwoma miejscami dziesiętnymi: 3.14"
'Dopełnione zerami: %.5d' % Math::PI           # => "Dopełnione zerami: 00003"
```

Szablony ERB przypominają swą postacią kod w języku JSP lub PHP. Zasadniczo szablon ERB traktowany jest jako „normalny" łańcuch, jednak pewne sekwencje sterujące traktowane są jako kod w języku Ruby lub aktualne wartości wyrażeń:

```
template = ERB.new %q{Pyszne <%= food %>!}
food = "kiełbaski"
template.result(binding)                       # => "Pyszne kiełbaski!"
food = "masło orzechowe"
template.result(binding)                       # => "Pyszne masło orzechowe!"
```

Poza sesją `irb` można pominąć wywołania metody `Kernel#binding`:

```
puts template.result
# Pyszne masło orzechowe!
```

Szablony ERB wykorzystywane są wewnętrznie przez widoki Rails i łatwo można rozpoznać je w plikach `.rhtml`.

Dyskusja

W szablonach ERB można odwoływać się do zmiennych (jak `food` w powyższym przykładzie), zanim zmienne te zostaną zdefiniowane. Wskutek wywołania metody `ERB#result` lub `ERB#run` szablon jest wartościowany zgodnie z bieżącymi wartościami tych zmiennych.

Podobnie jak kod w języku JSP i PHP, szablony ERB mogą zawierać pętle i rozgałęzienia warunkowe. Oto przykład rozbudowanego szablonu ERB:

```
template = %q{
<% if problems.empty? %>
  Wygląda na to, że w kodzie nie ma błędów!
<% else %>
  W kodzie kryją się nastepujące potencjalne problemy:
  <% problems.each do |problem, line| %>
    * <%= problem %> w wierszu <%= line %>
  <% end %>
<% end %>}.gsub(/^\s+/, '')
template = ERB.new(template, nil, '<>')

problems = [["Użyj is_a? zamiast duck typing", 23],
    ["eval() jest potencjalnie niebezpieczne", 44]]
template.run(binding)
```

```
# W kodzie kryją się nastepujące potencjalne problemy:
# * Użyj is_a? zamiast duck typing w wierszu 23
# * eval() jest potencjalnie niebezpieczne w wierszu 44

problems = []
template.run(binding)
# Wygląda na to, że w kodzie nie ma błędów!
```

ERB jest wyrafinowanym mechanizmem, jednak ani szablony ERB, ani łańcuchy typu `printf` nie przypominają w niczym prostych podstawień prezentowanych w recepturze 1.2. Podstawienia te nie są aktywowane, jeśli łańcuch ujęty jest w apostrofy (`' ... '`) zamiast w cudzysłów (`" ... "`). Można wykorzystać ten fakt do zbudowania szablonu zawierającego metodę `eval`:

```
class String
  def substitute(binding=TOPLEVEL_BINDING)
    eval(%{"#{self}"}, binding)
  end
end

template = %q{Pyszne #{food}!}                    # => "Pyszne \#{food}!"

food = 'kiełbaski'
template.substitute(binding)                      # => "Pyszne kiełbaski!"
food = 'masło orzechowe'
template.substitute(binding)                      # => "Pyszne masło orzechowe!"
```

Należy zachować szczególną ostrożność, używając metody `eval`, bowiem potencjalnie stwarza ona możliwość wykonania dowolnego kodu, z czego skwapliwie skorzystać może ewentualny włamywacz. Nie zdarzy się to jednak w poniższym przykładzie, jako że dowolna wartość zmiennej `food` wstawiona zostaje do łańcucha jeszcze przed jego interpolacją:

```
food = '#{system("dir")}'
puts template.substitute(binding)
# Pyszne #{system("dir")}!
```

Patrz także

- Powyżej prezentowaliśmy proste przykłady szablonów ERB; przykłady bardziej skomplikowane znaleźć można w dokumentacji klas ERB pod adresem *http://www.ruby-doc.org/stdlib/libdoc/erb/rdoc/classes/ERB.html*.
- Receptura 1.2, „Zastępowanie zmiennych w tworzonym łańcuchu”.
- Receptura 10.12, „Ewaluacja kodu we wcześniejszym kontekście”, zawiera informacje na temat obiektów `Binding`.

1.4. Odwracanie kolejności słów lub znaków w łańcuchu

Problem

Znaki lub słowa występują w łańcuchu w niewłaściwej kolejności.

Rozwiązanie

Do stworzenia nowego łańcucha, zawierającego znaki łańcucha oryginalnego w odwrotnej kolejności, można posłużyć się metodą `reverse`:

```
s = ".kapo an sipan tsej oT"
s.reverse                              # => "To jest napis na opak."
s                                      # => ".kapo an sipan tsej oT"

s.reverse!                             # => "To jest napis na opak."
s                                      # => "To jest napis na opak."
```

W celu odwrócenia kolejności słów w łańcuchu, należy podzielić go najpierw na podłańcuchy oddzielone „białymi" znakami[2] (czyli poszczególne słowa), po czym włączyć listę tych słów z powrotem do łańcucha, w odwrotnej kolejności.

```
s = "kolei. po nie Wyrazy "
s.split(/(\s+)/).reverse!.join('')   # => "Wyrazy nie po kolei."
s.split(/\b/).reverse!.join('')      # => "Wyrazy nie po. kolei"
```

Dyskusja

Metoda `String#split` wykorzystuje wyrażenie regularne w roli separatora. Każdorazowo gdy element łańcucha udaje się dopasować do tego wyrażenia, poprzedzająca go część łańcucha włączona zostaje do listy, a metoda `split` przechodzi do skanowania dalszej części łańcucha. Efektem przeskalowania całego łańcucha jest lista podłańcuchów znajdujących się między wystąpieniami separatora. Użyte w naszym przykładzie wyrażenie regularne `/(\s+)/` reprezentuje dowolny ciąg „białych" znaków, zatem metoda `split` dokonuje podziału łańcucha na poszczególne słowa (w potocznym rozumieniu).

Wyrażenie regularne `/b` reprezentuje granicę słowa; to nie to samo co „biały" znak, bowiem granicę słowa może także wyznaczać znak interpunkcyjny. Zwróć uwagę na konsekwencje tej różnicy w powyższym przykładzie.

Ponieważ wyrażenie regularne `/(\s+)/` zawiera parę nawiasów, separatory także są włączane do listy wynikowej. Jeżeli zatem zestawimy elementy tej listy w kolejności odwrotnej, separatory oddzielające słowa będą już obecne na swych miejscach. Poniższy przykład ilustruje różnicę między zachowywaniem a ignorowaniem separatorów:

```
"Trzy banalne wyrazy".split(/\s+/)   # => ["Trzy", "banalne", "wyrazy"]
"Trzy banalne wyrazy".split(/(\s+)/)
# => ["Trzy", " ", "banalne", "  ", "wyrazy"]
```

Patrz także

- Receptura 1.9, „Przetwarzanie poszczególnych słów łańcucha", ilustruje kilka wyrażeń regularnych wyznaczających alternatywną definicję „słowa".
- Receptura 1.11, „Zarządzanie białymi znakami".
- Receptura 1.17, „Dopasowywanie łańcuchów za pomocą wyrażeń regularnych".

[2] Pojęcie „białego znaku" wyjaśnione jest w recepturze 1.11 — *przyp. tłum.*

1.5. Reprezentowanie znaków niedrukowalnych

Problem

Należy stworzyć łańcuch zawierający znaki sterujące, znaki w kodzie UTF-8 lub dowolne znaki niedostępne z klawiatury.

Rozwiązanie

Ruby udostępnia kilka mechanizmów unikowych (*escape*) w celu reprezentowania znaków niedrukowalnych. W łańcuchach ujętych w cudzysłowy mechanizmy te umożliwiają reprezentowanie dowolnych znaków.

Dowolny znak można zakodować w łańcuchu, podając jego kod ósemkowy (*octal*) w formie `\ooo` lub kod szesnastkowy (*hexadecimal*) w formie `\xhh`.

```
octal = "\000\001\010\020"
octal.each_byte { |x| puts x }
# 0
# 1
# 8
# 16

hexadecimal = "\x00\x01\x10\x20"
hexadecimal.each_byte { |x| puts x }
# 0
# 1
# 16
# 32
```

W ten sposób umieszczać można w łańcuchach znaki, których nie można wprowadzić bezpośrednio z klawiatury czy nawet wyświetlić na ekranie terminala. Uruchom poniższy program, po czym otwórz wygenerowany plik *smiley.html* w przeglądarce WWW.

```
open('smiley.html', 'wb') do |f|
  f << '<meta http-equiv="Content-Type" content="text/html;charset=UTF-8">'
  f << "\xe2\x98\xBA"
end
```

Niektóre z niedrukowalnych znaków — te wykorzystywane najczęściej — posiadają specjalne, skrócone kody unikowe:

```
"\a" == "\x07"  # => true  #ASCII 0x07 = BEL (Dźwięk systemowy)
"\b" == "\x08"  # => true  #ASCII 0x08 = BS (Cofanie)
"\e" == "\x1b"  # => true  #ASCII 0x1B = ESC (Escape)
"\f" == "\x0c"  # => true  #ASCII 0x0C = FF (Nowa strona)
"\n" == "\x0a"  # => true  #ASCII 0x0A = LF (Nowy wiersz)
"\r" == "\x0d"  # => true  #ASCII 0x0D = CR (Początek wiersza)
"\t" == "\x09"  # => true  #ASCII 0x09 = HT (Tabulacja pozioma)
"\v" == "\x0b"  # => true  #ASCII 0x0B = VT (Tabulacja pionowa)
```

Dyskusja

W języku Ruby łańcuchy są ciągami bajtów. Nie ma znaczenia, czy bajty te są drukowalnymi znakami ASCII, niedrukowalnymi znakami binarnymi, czy też mieszanką obydwu tych kategorii.

Znaki niedrukowalne wyświetlane są w języku Ruby w czytelnej dla człowieka reprezentacji `\ooo`, gdzie `ooo` jest kodem znaku w reprezentacji ósemkowej; znaki posiadające reprezentację mnemoniczną w postaci `\<znak>` wyświetlane są jednak w tej właśnie postaci. Znaki drukowalne wyświetlane są zawsze w swej naturalnej postaci, nawet jeżeli w tworzonym łańcuchu zakodowane zostały w inny sposób.

```
"\x10\x11\xfe\xff"              # => "\020\021\376\377"
"\x48\145\x6c\x6c\157\x0a"      # => "Hello\n"
```

Znak odwrotnego ukośnika (\) reprezentowany jest przez *parę* takich ukośników (`\\`) — jest to konieczne dla odróżnienia literalnego użycia znaku `\` od mnemonicznej sekwencji unikowej rozpoczynającej się od takiego znaku. Przykładowo, łańcuch `"\\n"` składa się z dwóch znaków: odwrotnego ukośnika i litery `n`.

```
"\\".size                       # => 1
"\\" == "\x5c"                  # => true
"\\n"[0] == ?\\                 # => true
"\\n"[1] == ?n                  # => true
"\\n" =~ /\n/                   # => nil
```

Ruby udostępnia także kilka wygodnych skrótów dla reprezentowania kombinacji klawiszy w rodzaju *Ctrl+C*. Sekwencja `\C-<znak>` oznacza rezultat naciśnięcia klawisza `<znak>` z jednoczesnym przytrzymaniem klawisza *Ctrl*; analogicznie sekwencja `\M-<znak>` oznacza rezultat naciśnięcia klawisza `<znak>` z jednoczesnym przytrzymaniem klawisza *Alt* (lub *Meta*):

```
"\C-a\C-b\C-c"                  # => "\001\002\003"  # Ctrl+A Ctrl+B Ctrl+C
"\M-a\M-b\M-c"                  # => "\341\342\343"  # Alt+A  Alt+B  Alt+C
```

Dowolna z opisywanych sekwencji może pojawić się wszędzie tam, gdzie Ruby spodziewa się znaku. W szczególności możliwe jest wyświetlenie kodu znaku w postaci *dziesiętnej* — należy w tym celu poprzedzić ów znak znakiem zapytania (`?`).

```
?\C-a                           # => 1
?\M-z                           # => 250
```

W podobny sposób można używać rozmaitych reprezentacji znaków do specyfikowania zakresu znaków w wyrażeniach regularnych:

```
contains_control_chars = /[\C-a-\C-^]/
'Foobar' =~ contains_control_chars        # => nil
"Foo\C-zbar" =~ contains_control_chars    # => 3

contains_upper_chars = /[\x80-\xff]/
'Foobar' =~ contains_upper_chars          # => nil
"Foo\212bar" =~ contains_upper_chars      # => 3
```

Poniższa aplikacja śledzi („szpieguje”) naciśnięcia klawiszy, reagując na niektóre kombinacje specjalne:

```
def snoop_on_keylog(input)
  input.each_byte  do |b|
    case b
      when ?\C-c; puts 'Ctrl+C: zatrzymać proces?'
      when ?\C-z; puts 'Ctrl+Z: zawiesić proces?'
      when ?\n;   puts 'Nowy wiersz.'
      when ?\M-x; puts 'Alt+X: uruchomić Emacs?'
    end
  end
end


snoop_on_keylog("ls -ltR\003emacsHello\012\370rot13-other-window\012\032")
# Ctrl+C: zatrzymać proces?
```

```
# Nowy wiersz.
# Alt+X: uruchomić Emacs?
# Nowy wiersz.
# Ctrl+Z: zawiesić proces?
```

Sekwencje reprezentujące znaki specjalne interpretowane są tylko w łańcuchach ujętych w cudzysłów oraz łańcuchach tworzonych za pomocą konstrukcji `%{}` lub `%Q{}`. Nie są one interpretowane w łańcuchach zamkniętych znakami apostrofu oraz łańcuchach tworzonych za pomocą konstrukcji `%q{}`. Fakt ten można wykorzystać w przypadku potrzeby literalnego wyświetlenia sekwencji reprezentujących znaki specjalne oraz w przypadku tworzenia łańcuchów zawierających dużą liczbę odwrotnych ukośników.

```
puts "foo\tbar"
# foo     bar
puts %{foo\tbar}
# foo     bar
puts %Q{foo\tbar}
# foo     bar

puts 'foo\tbar'
# foo\tbar
puts %q{foo\tbar}
# foo\tbar
```

Nieinterpretowanie sekwencji reprezentujących znaki specjalne w łańcuchach zamkniętych znakami apostrofu może wydać się dziwne — i niekiedy nieco kłopotliwe — programistom przyzwyczajonym do języka Python. Jeżeli łańcuch ujęty w cudzysłów sam zawiera znaki cudzysłowu (`"`), jedynym sposobem reprezentowania tychże jest użycie sekwencji unikowej `\"`, `\042` lub `\x22`. W przypadku łańcuchów obfitujących w znaki cudzysłowu może się to wydać kłopotliwe i najwygodniejszym rozwiązaniem jest zamknięcie łańcucha apostrofami — znaków cudzysłowu można wówczas używać literalnie, tracimy jednak możliwość interpretowania znaków specjalnych. Na szczęście istnieje złoty środek pozwalający na pogodzenie tych sprzecznych racji: jeśli chcesz zachować możliwość interpretowania znaków specjalnych w łańcuchach najeżonych znakami cudzysłowu, użyj konstrukcji `%{}`.

1.6. Konwersja między znakami a kodami

Problem

Chcemy otrzymać kod ASCII danego znaku lub przetransformować kod ASCII znaku w sam znak.

Rozwiązanie

Kod ASCII znaku możemy poznać za pomocą operatora `?`:

```
?a                      # => 97
?!                      # => 33
?\n                     # => 10
```

W podobny sposób możemy poznać kod ASCII znaku wchodzącego w skład łańcucha — należy wówczas wyłuskać ów znak z łańcucha za pomocą indeksu:

```
'a'[0]                  # => 97
'kakofonia'[1]          # => 97
```

Konwersję odwrotną — kodu ASCII na znak o tym kodzie — realizuje metoda `chr`, zwracająca jednoznakowy łańcuch:

```
97.chr                  # => "a"
33.chr                  # => "!"
10.chr                  # => "\n"
0.chr                   # => "\000"
256.chr                 # RangeError: 256 out of char range
```

Dyskusja

Mimo iż łańcuch jako taki nie jest tablicą, może być utożsamiany z tablicą obiektów `Fixnum` — po jednym obiekcie dla każdego bajta. Za pomocą odpowiedniego indeksu można wyłuskać obiekt `Fixnum` reprezentujący konkretny bajt łańcucha, czyli kod ASCII tego bajta. Za pomocą metody `String#each_byte` można iterować po wszystkich obiektach `Fixnum` tworzących dany łańcuch.

Patrz także

- Receptura 1.8, „Przetwarzanie kolejnych znaków łańcucha".

1.7. Konwersja między łańcuchami a symbolami

Problem

Mając symbol języka Ruby, należy uzyskać reprezentujący go łańcuch, lub vice versa — zidentyfikować symbol odpowiadający danemu łańcuchowi.

Rozwiązanie

Konwersję symbolu na odpowiadający mu łańcuch realizuje metoda `Symbol#to_s` lub metoda `Symbol#id2name`, dla której `to_s` jest aliasem.

```
:a_symbol.to_s                  # => "a_symbol"
:InnySymbol.id2name             # => "InnySymbol"
:"Jeszcze jeden symbol!".to_s   # => "Jeszcze jeden symbol!"
```

Odwołanie do symbolu następuje zwykle przez jego nazwę. Aby uzyskać symbol reprezentowany przez łańcuch w kodzie programu, należy posłużyć się metodą `String.intern`:

```
:dodecahedron.object_id         # => 4565262
symbol_name = "dodecahedron"
symbol_name.intern              # => :dodecahedron
symbol_name.intern.object_id    # => 4565262
```

Dyskusja

`Symbol` jest najbardziej podstawowym obiektem języka Ruby. Każdy symbol posiada nazwę i wewnętrzny identyfikator (*internal ID*). Użyteczność symboli wynika z faktu, że wielokrotne wystąpienie tej samej nazwy w kodzie programu oznacza każdorazowo odwołanie do tego samego symbolu.

Symbole są często bardziej użyteczne niż łańcuchy. Dwa łańcuchy o tej samej zawartości są dwoma różnymi obiektami — można jeden z nich zmodyfikować bez wpływu na drugi. Dwie identyczne nazwy odnoszą się do tego samego symbolu, co oczywiście przekłada się na oszczędność czasu i pamięci.

```
"string".object_id     # => 1503030
"string".object_id     # => 1500330
:symbol.object_id      # => 4569358
:symbol.object_id      # => 4569358
```

Tak więc *n* wystąpień tej samej nazwy odnosi się do tego samego symbolu, przechowywanego w pamięci w jednym egzemplarzu. *n* identycznych łańcuchów to *n* różnych obiektów o identycznej zawartości. Także porównywanie symboli jest szybsze niż porównywanie łańcuchów, bowiem sprowadza się jedynie do porównywania identyfikatorów.

```
"string1" == "string2"        # => false
:symbol1 == :symbol2          # => false
```

Na koniec zacytujmy hakera od języka Ruby, Jima Wericha:

- Użyj łańcucha, jeśli istotna jest zawartość obiektu (sekwencja tworzących go znaków).
- Użyj symbolu, jeśli istotna jest tożsamość obiektu.

Patrz także

- Receptura 5.1, „Wykorzystywanie symboli jako kluczy”.
- Receptura 8.12, „Symulowanie argumentów zawierających słowa kluczowe”.
- Rozdział 10., a szczególnie receptura 10.4, „Uzyskiwanie referencji do metody”, i receptura 10.10, „Oszczędne kodowanie dzięki metaprogramowaniu”.
- *http://glu.ttono.us/articles/2005/08/19/understanding-ruby-symbols* — interesujący artykuł o symbolach języka Ruby.

1.8. Przetwarzanie kolejnych znaków łańcucha

Problem

Należy wykonać pewną czynność w stosunku do każdego znaku łańcucha z osobna.

Rozwiązanie

W dokumencie złożonym wyłącznie ze znaków ASCII każdy bajt łańcucha odpowiada jednemu znakowi. Za pomocą metody `String#each_byte` można wyodrębnić poszczególne bajty jako liczby, które następnie mogą być skonwertowane na znaki.

```
'foobar'.each_byte { |x| puts "#{x} = #{x.chr}" }
# 102 = f
# 111 = o
# 111 = o
# 98 = b
# 97 = a
# 114 = r
```

Za pomocą metody `String#scan` można wyodrębnić poszczególne znaki łańcucha jako jednoznakowe łańcuchy:

```
'foobar'.scan( /./ ) { |c| puts c }
# f
# o
# o
# b
# a
# r
```

Dyskusja

Ponieważ łańcuch jest sekwencją bajtów, można by oczekiwać, że metoda `String#each` umożliwia iterowanie po tej sekwencji, podobnie jak metoda `Array#each`. Jest jednak inaczej — metoda `String#each` dokonuje podziału łańcucha na podłańcuchy względem pewnego separatora (którym domyślnie jest znak nowego wiersza):

```
"foo\nbar".each { |x| puts x }
# foo
# bar
```

Odpowiednikiem metody `Array#each` w odniesieniu do łańcuchów jest metoda `each_byte`. Każdy element łańcucha może być traktowany jako obiekt `Fixnum`, a metoda `each_byte` umożliwia iterowanie po sekwencji tych obiektów.

Metoda `String#each_byte` jest szybsza niż `String#scan` i jako taka zalecana jest w przypadku przetwarzania plików ASCII — każdy wyodrębniony obiekt `Fixnum` może być łatwo przekształcony w znak (jak pokazano w Rozwiązaniu).

Metoda `String#scan` dokonuje sukcesywnego dopasowywania podanego wyrażenia regularnego do kolejnych porcji łańcucha i wyodrębnia każdą z tych opcji. Jeżeli wyrażeniem tym jest `/./`, wyodrębniane są poszczególne znaki łańcucha.

Jeśli zmienna `$KCODE` jest odpowiednio ustawiona, metoda `scan` może być stosowana także do łańcuchów zawierających znaki w kodzie UTF-8. Jest to najprostsza metoda przeniesienia koncepcji „znaku” na grunt łańcuchów języka Ruby, które z definicji są ciągami bajtów, nie znaków.

Poniższy łańcuch zawiera zakodowaną w UTF-8 francuską frazę „ça va”:

```
french = "\xc3\xa7a va"
```

Nawet jeżeli znaku `ç` nie sposób poprawnie wyświetlić na terminalu, poniższy przykład ilustruje zmianę zachowania metody `String#scan` w sytuacji, gdy określi się wyrażenie regularne stosownie do standardów Unicode lub ustawi zmienną `$KCODE` tak, by Ruby traktował wszystkie łańcuchy jako kodowane według UTF-8:

```
french.scan(/./) { |c| puts c }
# Ã
# §
# a
#
# v
# a

french.scan(/./u) { |c| puts c }
# ç
# a
#
```

```
# v
# a

$KCODE = 'u'
french.scan(/./) { |c| puts c }
# ç
# a
#
# v
# a
```

Gdy Ruby traktuje łańcuchy jako sekwencje znaków UTF-8, a nie ASCII, dwa bajty reprezentujące znak ç traktowane są łącznie, jako pojedynczy znak. Nawet jeśli niektórych znaków UTF-8 nie można wyświetlić na ekranie terminala, można stworzyć programy, które zajmą się ich obsługą.

Patrz także

- Receptura 11.12, „Konwersja dokumentu między różnymi standardami kodowania".

1.9. Przetwarzanie poszczególnych słów łańcucha

Problem

Należy wydzielić z łańcucha jego kolejne słowa i dla każdego z tych słów wykonać pewną czynność.

Rozwiązanie

Najpierw należy zastanowić się nad tym, co rozumiemy pod pojęciem „słowa" w łańcuchu. Co oddziela od siebie sąsiednie słowa? Tylko białe znaki, czy może także znaki interpunkcyjne? Czy „taki-to-a-taki" to pojedyncze słowo, czy może cztery słowa? Te i inne kwestie rozstrzyga się jednoznacznie, definiując wyrażenie regularne reprezentujące pojedyncze słowo (kilka przykładów takich wyrażeń podajemy poniżej w Dyskusji).

Wspomniane wyrażenie regularne należy przekazać jako parametr metody `String#scan`, która tym samym dokona podzielenia łańcucha na poszczególne słowa. Prezentowana poniżej metoda `word_count` zlicza wystąpienia poszczególnych słów w analizowanym tekście; zgodnie z użytym wyrażeniem regularnym „słowo" ma składnię identyczną z identyfikatorem języka Ruby, jest więc ciągiem liter, cyfr i znaków podkreślenia:

```
class String
  def word_count
    frequencies = Hash.new(0)
    downcase.scan(/\w+/) { |word| frequencies[word] += 1 }
   return frequencies
  end
end

%{Dogs dogs dog dog dogs.}.word_count
# => {"dogs"=>3, "dog"=>2}
%{"I have no shame," I said.}.word_count
# => {"no"=>1, "shame"=>1, "have"=>1, "said"=>1, "i"=>2}
```

Dyskusja

Wyrażenie regularne `/\w+/` jest co prawda proste i eleganckie, jednakże ucieleśniana przezeń definicja „słowa" z pewnością pozostawia wiele do życzenia. Przykładowo, rzadko kto skłonny byłby uważać za pojedyncze słowo dwa słowa (w rozumieniu potocznym) połączone znakiem podkreślenia, ponadto niektóre ze słów angielskich — jak „pan-fried" czy „foc's'le" — zawierają znaki interpunkcyjne. Warto więc być może rozważyć kilka alternatywnych wyrażeń regularnych, opartych na bardziej wyszukanych koncepcjach słowa:

```
# Podobne do /\w+/, lecz nie dopuszcza podkresleń wewnątrz słowa.
/[0-9A-Za-z]/

# Dopuszcza w słowie dowolne znaki oprócz białych znaków.
/[^\S]+/

# Dopuszcza w słowie litery, cyfry, apostrofy i łączniki
/[-'\w]+/

# Zadowalająca heurystyka reprezentowania słów angielskich
/(\w+([-'.]\w+)*)/
```

Ostatnie z prezentowanych wyrażeń regularnych wymaga krótkiego wyjaśnienia. Reprezentowana przezeń koncepcja dopuszcza znaki interpunkcyjne wewnątrz słowa, lecz nie na jego krańcach — i tak na przykład „Work-in-progress" zostanie w świetle tej koncepcji uznane za pojedyncze słowo, lecz już łańcuch „--never--" rozpoznany zostanie jako słowo „never" otoczone znakami interpunkcyjnymi. Co więcej, poprawnie rozpoznane zostaną akronimy w rodzaju „U.N.C.L.E." czy „Ph.D." — no, może nie do końca poprawnie, ponieważ *ostatnia* z kropek, równouprawniona z poprzednimi, nie zostanie zaliczona w poczet słowa i pierwszy z wymienionych akronimów zostanie rozpoznany jako słowo „U.N.C.L.E", po którym następuje kropka.

Napiszmy teraz na nowo naszą metodę `word_count`, wykorzystując ostatnie z prezentowanych wyrażeń regularnych. Różni się ono od wersji poprzedniej pewnym istotnym szczegółem: otóż wykorzystywane wyrażenie regularne składa się tym razem z dwóch grup. Metoda `String#scan` wyodrębni więc każdorazowo *dwa* podłańcuchy i przekaże je jako dwa argumenty do swego bloku kodowego. Ponieważ tylko pierwszy z tych argumentów reprezentować będzie rzeczywiste słowo, drugi z nich musimy zwyczajnie zignorować.

```
class String
  def word_count
    frequencies = Hash.new(0)
    downcase.scan(/(\w+([-'.]\w+)*)/) { |word, ignore| frequencies[word] += 1 }
    return frequencies
  end
end

%{"That F.B.I. fella--he's quite the man-about-town."}.word_count
# => {"quite"=>1, "f.b.i"=>1, "the"=>1, "fella"=>1, "that"=>1,
#     "man-about-town"=>1, "he's"=>1}
```

Zwróćmy uwagę, iż fraza `\w` reprezentować może różne rzeczy w zależności od wartości zmiennej `$KCODE`. Domyślnie reprezentuje ona jedynie słowa składające się wyłącznie ze znaków ASCII:

```
french = "il \xc3\xa9tait une fois"
french.word_count
# => {"fois"=>1, "une"=>1, "tait"=>1, "il"=>1}
```

Jeśli jednak włączymy obsługę kodu UTF-8, reprezentować będzie ona także słowa zawierające znaki w tymże kodzie:

```
$KCODE='u'
french.word_count
# => {"fois"=>1, "une"=>1, "était"=>1, "il"=>1}
```

Grupa `/b` w wyrażeniu regularnym reprezentuje granicę słowa, czyli ostatnie słowo poprzedzające biały znak lub znak interpunkcyjny. Fakt ten bywa użyteczny w odniesieniu do metody `String#split` (patrz receptura 1.4), lecz już nie tak użyteczny w stosunku do metody `String#scan`.

Patrz także

- Receptura 1.4, „Odwracanie kolejności słów lub znaków w łańcuchu".
- W bibliotece `Facets core` zdefiniowana jest metoda `String#each_word`, wykorzystująca wyrażenie regularne `/([-'\w]+)/`.

1.10. Zmiana wielkości liter w łańcuchu

Problem

Wielkie/małe litery są niewłaściwie użyte w łańcuchu.

Rozwiązanie

Klasa `String` definiuje kilka metod zmieniających wielkość liter w łańcuchu:

```
s = 'WITAM, nie ma Mnie W Domu, JesTeM W kaWIArNi.'
s.upcase           # => "WITAM, NIE MA MNIE W DOMU, JESTEM W KAWIARNI."
s.downcase         # => "witam, nie ma mnie w domu, jestem w kawiarni."
s.swapcase         # => "witam, NIE MA mNIE w dOMU, jEStEm w KAwiaRnI."
s.capitalize       # => "Witam, nie ma mnie w domu, jestem w kawiarni."
```

Dyskusja

Metody `upcase` i `downcase` wymuszają zmianę wszystkich liter w łańcuchu na (odpowiednio) wielkie i małe. Metoda `swapcase` dokonuje zamiany małych liter na wielkie i vice versa. Metoda `capitalize` dokonuje zamiany pierwszego znaku łańcucha na wielką literę *pod warunkiem, że znak ten jest literą*; wszystkie następne litery w łańcuchu zamieniane są na małe.

Każda z czterech wymienionych metod posiada swój odpowiednik dokonujący stosownej zamiany liter *w miejscu* — `upcase!`, `downcase!`, `swapcase!` i `capitalize!`. Przy założeniu, że oryginalny łańcuch nie jest dłużej potrzebny, użycie tych metod może zmniejszyć zajętość pamięci, szczególnie w przypadku długich łańcuchów:

```
un_banged = 'Hello world.'
un_banged.upcase              # => "HELLO WORLD."
un_banged                     # => "Hello world."

banged = 'Hello world.'
banged.upcase!                # => "HELLO WORLD."
banged                        # => "HELLO WORLD."
```

W niektórych przypadkach istnieje potrzeba zamiany pierwszego znaku łańcucha na wielką literę (jeśli w ogóle jest literą) *bez* zmiany wielkości pozostałych liter — w łańcuchu mogą bowiem występować nazwy własne. Czynność tę realizują dwie poniższe metody — druga oczywiście dokonuje stosownej zamiany „w miejscu":

```
class String
  def capitalize_first_letter
    self[0].chr.capitalize + self[1, size]
  end

  def capitalize_first_letter!
    unless self[0] == (c = self[0,1].upcase[0])
      self[0] = c
      self
    end
    # Zwraca nil, jeśli nie dokonano żadnych zmian, podobnie jak np. upcase!.
  end
end

s = 'teraz jestem w Warszawie. Jutro w Sopocie.'
s.capitalize_first_letter          # => "Teraz jestem w Warszawie. Jutro w Sopocie."
s                                  # => "teraz jestem w Warszawie. Jutro w Sopocie."
s.capitalize_first_letter!
s                                  # => "Teraz jestem w Warszawie. Jutro w Sopocie."
```

Do zmiany wielkości wybranej litery w łańcuchu, bez zmiany wielkości pozostałych liter, można wykorzystać metodę `tr` lub `tr!`, dokonującą translacji jednego znaku na inny:

```
'LOWERCASE ALL VOWELS'.tr('AEIOU', 'aeiou')
# => "LoWeRCaSe aLL VoWeLS"

'Swap case of ALL VOWELS'.tr('AEIOUaeiou', 'aeiouAEIOU')
# => "SwAp cAsE Of aLL VoWeLS"
```

Patrz także

- Receptura 1.18, „Zastępowanie wielu wzorców w pojedynczym przebiegu".
- W bibliotece `Facets core` zdefiniowana jest metoda `String#camelcase` oraz metody predykatowe `String#lowercase?` i `String#uppercase?`.

1.11. Zarządzanie białymi znakami

Problem

Łańcuch zawiera zbyt dużo lub zbyt mało białych znaków, bądź użyto w nim niewłaściwych białych znaków.

Rozwiązanie

Za pomocą metody `strip` można usunąć białe znaki z początku i końca łańcucha.

```
" \tWhitespace at beginning and end. \t\n\n".strip
# => "Whitespace at beginning and end."
```

Metody `ljust`, `rjust` i `center` dokonują (odpowiednio) wyrównania łańcucha do lewej strony, wyrównania do prawej oraz wyśrodkowania:

```
s = "To jest napis."           # => "To jest napis."
s.center(30) =>                # => "        To jest napis.        "
s.ljust(30) =>                 # => "To jest napis.                "
s.rjust(30) =>                 # => "                To jest napis."
```

Za pomocą metody `gsub`, w połączeniu z wyrażeniami regularnymi, można dokonywać zmian bardziej zaawansowanych, na przykład zastępować jeden typ białych znaków innym:

```
# Normalizacja kodu przez zastępowanie każdego tabulatora ciągiem dwóch spacji
rubyCode.gsub("\t", "  ")

# Zamiana ograniczników wiersza z windowsowych na uniksowe
"Line one\n\rLine two\n\r".gsub("\n\r", "\n")
# => "Line one\nLine two\n"

# Zamiana każdego ciągu białych znaków na pojedynczą spację
"\n\rThis string\t\t\tuses\n all\tsorts\nof whitespace.".gsub(/\s+/, " ")
# => " This string uses all sorts of whitespace."
```

Dyskusja

Białym znakiem (*whitespace*) jest każdy z pięciu następujących znaków: spacja, tabulator (`\t`), znak nowego wiersza (`\n`), znak powrotu do początku wiersza (`\r`) i znak nowej strony (`\f`). Wyrażenie regularne `/\s/` reprezentuje dowolny znak z tego zbioru. Metoda `strip` dokonuje usunięcia dowolnej kombinacji tych znaków z początku i końca łańcucha.

Niekiedy konieczne jest przetwarzanie innych niedrukowalnych znaków w rodzaju *backspace* (`\b` lub `\010`) czy tabulatora pionowego (`\v` lub `\012`). Znaki te nie należą do grupy znaków reprezentowanych przez `/s` w wyrażeniu regularnym i trzeba je reprezentować explicite:

```
" \bIt's whitespace, Jim,\vbut not as we know it.\n".gsub(/[\s\b\v]+/, " ")
# => " It's whitespace, Jim, but not as we know it. "
```

Do usunięcia białych znaków tylko z początku lub tylko z końca łańcucha można wykorzystać metody (odpowiednio) `lstrip` i `rstrip`:

```
s = "   Whitespace madness! "
s.lstrip                       # => "Whitespace madness! "
s.rstrip                       # => "   Whitespace madness!"
```

Metody dopełniające spacjami do żądanej długości (`ljust`, `rjust` i `center`) posiadają jeden argument wywołania — tę właśnie długość. Jeżeli wyśrodkowanie łańcucha nie może być wykonane idealnie, bo liczba dołączanych spacji jest nieparzysta, z prawej strony dołączana jest jedna spacja więcej niż z lewej.

```
"napis".center(9)              # => "  napis  "
"napis".center(10)             # => "  napis   "
```

Podobnie jak większość metod modyfikujących łańcuchy, metody `strip`, `gsub`, `lstrip` i `rstrip` posiadają swe odpowiedniki operujące „w miejscu” — `strip!`, `gsub!`, `lstrip!` i `rstrip!`.

1.12. Czy można potraktować dany obiekt jak łańcuch?

Problem

Czy dany obiekt przejawia elementy funkcjonalności charakterystyczne dla łańcuchów?

Rozwiązanie

Sprawdź, czy obiekt definiuje metodę `to_str`.

```
'To jest napis'.respond_to? :to_str          # => true
Exception.new.respond_to? :to_str            # => true
4.respond_to? :to_str                        # => false
```

Sformułowany powyżej problem możemy jednak rozważać w postaci bardziej ogólnej: czy mianowicie dany obiekt definiuje pewną konkretną metodę klasy `String`, z której to metody chcielibyśmy skorzystać. Oto przykład konkatenacji obiektu z jego następnikiem i konwersji wyniku do postaci łańcucha — to wszystko wykonalne jest jednak tylko wtedy, gdy obiekt definiuje metodę `succ` wyznaczającą następnik:

```
def join_to_successor(s)
  raise ArgumentError, 'Obiekt nie definiuje metody succ!' unless s.respond_to? :succ
  return "#{s}#{s.succ}"
end

join_to_successor('a')            # => "ab"
join_to_successor(4)              # => "45"
join_to_successor(4.01)           # ArgumentError: Obiekt nie definiuje metody succ!
```

Gdybyśmy zamiast predykatu `s.respond_to? :succ` użyli predykatu `s.is_a? String`, okazałoby się, że nie jest możliwe wyznaczenie następnika dla liczby całkowitej:

```
def join_to_successor(s)
  raise ArgumentError, 'Obiekt nie jest łańcuchem!' unless s.is_a? String
  return "#{s}#{s.succ}"
end

join_to_successor('a')            # => "ab"
join_to_successor(4)              # => ArgumentError: 'Obiekt nie jest łańcuchem!'
join_to_successor(4.01)           # => ArgumentError: 'Obiekt nie jest łańcuchem!'
```

Dyskusja

To, co widzimy powyżej, jest najprostszym przykładem pewnego aspektu filozofii języka Ruby, zwanego „kaczym typowaniem" (*duck typing*): jeśli mianowicie chcemy przekonać się, że dane zwierzę jest kaczką, możemy skłonić je do wydania głosu — powinniśmy wówczas usłyszeć kwakanie. Na podobnej zasadzie możemy badać rozmaite aspekty funkcjonalności obiektu, sprawdzając, czy obiekt ów definiuje metody o określonych nazwach, realizujące tę właśnie funkcjonalność.

Jak przekonaliśmy się przed chwilą, predykat `obj.is_a? String` nie jest najlepszym sposobem badania, czy mamy do czynienia z łańcuchem. Owszem, jeśli predykat ten jest spełniony, obiekt łańcuchem jest niewątpliwie, jego klasa wywodzi się bowiem z klasy `String`; zależność odwrotna nie zawsze jest jednak prawdziwa — pewne zachowania typowe dla łańcuchów mogą być przejawiane przez obiekty niewywodzące się z klasy `String`.

Jako przykład posłużyć może klasa `Exceptions`, której obiekty są koncepcyjnie łańcuchami wzbogaconymi o pewne dodatkowe informacje. Klasa `Exceptions` nie jest jednak subklasą klasy `String` i użycie w stosunku do niej predykatu `is_a? String` może spowodować przeoczenie jej „łańcuchowości". Wiele modułów języka Ruby definiuje inne rozmaite klasy o tejże własności.

Warto więc zapamiętać (i stosować) opisaną filozofię: jeśli chcemy badać pewien aspekt funkcjonalny obiektu, powinniśmy czynić to, sprawdzając (za pomocą predykatu `respond_to?`), czy obiekt ten definiuje określoną metodę, zamiast badać jego genealogię za pomocą predykatu `is_a?`. Pozwoli to w przyszłości na definiowanie nowych klas oferujących te same możliwości, bez krępującego uzależniania ich od istniejącej hierarchii klas. Jedynym uzależnieniem będzie wówczas uzależnienie od konkretnych *nazw* metod.

Patrz także

- Rozdział 8., szczególnie wstęp oraz receptura 8.3, „Weryfikacja funkcjonalności obiektu".

1.13. Wyodrębnianie części łańcucha

Problem

Mając dany łańcuch, należy wyodrębnić określone jego fragmenty.

Rozwiązanie

W celu wyodrębnienia podłańcucha możemy posłużyć się metodą `slice` lub wykorzystać operator indeksowania tablicy (czyli de facto wywołać metodę `[]`). W obydwu przypadkach możemy określić bądź to zakres (obiekt `Range`) wyodrębnianych znaków, bądź parę liczb całkowitych (obiektów `Fixnum`) określających (kolejno) indeks pierwszego wyodrębnianego znaku oraz liczbę wyodrębnianych znaków:

```
s = "To jest napis"
s.slice(0,2)              # => "To"
s[3,4]                    # => "jest"
s[8,5]                    # => "napis"
s[8,0]                    # => ""
```

Aby wyodrębnić pierwszą porcję łańcucha pasującą do danego wyrażenia regularnego, należy wyrażenia tego użyć jako argumentu wywołania metody `slice` lub operatora indeksowego:

```
s[/.pis/]                 # => "apis"
s[/na.*/]                 # => "napis"
```

Dyskusja

Dla uzyskania pojedynczego bajta łańcucha (jako obiektu `Fixnum`) wystarczy podać jeden argument — indeks tego bajta (pierwszy bajt ma indeks 0). Aby otrzymać znakową postać owego bajta, należy podać dwa argumenty: jego indeks oraz `1`:

```
s.slice(3)                # => 106
s[3]                      # => 106
106.chr                   # => "j"
s.slice(3,1)              # => "j"
s[3,1]                    # => "j"
```

Ujemna wartość pierwszego argumentu oznacza indeks liczony *względem końca* łańcucha:

```
s.slice(-1,1)             # => "s"
s.slice(-5,5)             # => "napis"
s[-5,5]                   # => "napis"
```

Jeżeli specyfikowana długość podłańcucha przekracza długość całego łańcucha liczoną od miejsca określonego przez pierwszy argument, zwracana jest cała reszta łańcucha począwszy od tego miejsca. Umożliwia to wygodne specyfikowanie „końcówek" łańcuchów:

```
s[8,s.length]                 # => "napis"
s[-5,s.length]                # => "napis"
s[-5, 65535]                  # => "napis"
```

Patrz także

- Receptura 1.9, „Przetwarzanie poszczególnych słów łańcucha".
- Receptura 1.17, „Dopasowywanie łańcuchów za pomocą wyrażeń regularnych".

1.14. Obsługa międzynarodowego kodowania

Problem

W łańcuchu znajdują się znaki niewchodzące w skład kodu ASCII — na przykład znaki Unicode kodowane według UTF-8.

Rozwiązanie

Aby zapewnić poprawną obsługę znaków Unicode, należy na początku kodu umieścić następującą sekwencję:

```
$KCODE='u'
require 'jcode'
```

Identyczny efekt można osiągnąć, uruchamiając interpreter języka Ruby w następujący sposób:

```
$ ruby -Ku –rjcode
```

W środowisku Uniksa można określić powyższe parametry w poleceniu uruchamiającym skrypt (*shebang line*):

```
#!/usr/bin/ruby -Ku –rjcode
```

W bibliotece `jcode` większość metod klasy `String` została przedefiniowana tak, by metody te zapewniały obsługę znaków wielobajtowych. *Nie* przedefiniowano metod `String#length`, `String#count` i `String#size`, definiując w zamian trzy nowe metody, `String#jlength`, `String#jcount` i `String#jsize`.

Dyskusja

Rozpatrzmy przykładowy łańcuch zawierający sześć znaków Unicode: `efbca1` (`A`), `efbca2` (`B`), `efbca3` (`C`), `efbca4` (`D`), `efbca5` (`E`) i `efbca6` (`F`):

```
string = "\xef\xbc\xa1" + "\xef\xbc\xa2" + "\xef\xbc\xa3" +
         "\xef\xbc\xa4" + "\xef\xbc\xa5" + "\xef\xbc\xa6"
```

Łańcuch ten składa się z 18 bajtów, kodujących 6 znaków:

```
string.size                                   # => 18
string.jsize                                  # => 6
```

Metoda `String#count` zlicza wystąpienia określonych bajtów w łańcuchu, podczas gdy metoda `String#jcount` dokonuje zliczania określonych znaków:

```
string.count "\xef\xbc\xa2"                    # => 13
string.jcount "\xef\xbc\xa2"                   # => 1
```

W powyższym przykładzie metoda `count` traktuje argument `"\xef\xbc\xa2"` jak *trzy oddzielne bajty* `\xef`, `\xbc` i `\xa2`, zwracając *sumę* liczby ich wystąpień w łańcuchu (6+6+1). Metoda `jcount` traktuje natomiast swój argument jako *pojedynczy znak*, zwracając liczbę jego wystąpień w łańcuchu (w tym przypadku znak występuje tylko raz).

```
"\xef\xbc\xa2".length                          # => 3
"\xef\xbc\xa2".jlength                         # => 1
```

Metoda `String#length` zwraca, jak wiadomo, liczbę bajtów łańcucha niezależnie od tego, jakie znaki są za pomocą tych bajtów kodowane. Metoda `String#jlength` zwraca natomiast liczbę kodowanych znaków.

Mimo tych wyraźnych różnic obsługa znaków Unicode odbywa się w języku Ruby w większości „pod podszewką" — przetwarzanie łańcuchów zawierających znaki kodowane według UTF-8 odbywa się w sposób elegancki i naturalny, bez jakiejś szczególnej troski ze strony programisty. Stanie się to całkowicie zrozumiałe, gdy uświadomimy sobie, że twórca Ruby — Yukihiro Matsumoto — jest Japończykiem.

Patrz także

- Tekst złożony ze znaków kodowanych w systemie innym niż UTF-8 może być łatwo przekodowany do UTF-8 za pomocą biblioteki `iconv`, o czym piszemy w recepturze 11.2, „Ekstrakcja informacji z drzewa dokumentu".
- Istnieje kilka wyszukiwarek on-line obsługujących znaki Unicode; dwiema godnymi polecenia wydają się naszym zdaniem *http://isthisthingon.org/unicode/* oraz *http://www.fileformat.info/info/unicode/char/search.htm*.

1.15. Zawijanie wierszy tekstu

Problem

Łańcuch zawierający dużą liczbę białych znaków należy sformatować, dzieląc go na wiersze, tak aby możliwe było jego wyświetlenie w oknie lub wysłanie e-mailem.

Rozwiązanie

Najprostszym sposobem wstawienia do łańcucha znaków nowego wiersza jest użycie wyrażenia regularnego podobnego do poniższego:

```
def wrap(s, width=78)
  s.gsub(/(.{1,#{width}})(\s+|\Z)/, "\\1\n")
end

wrap("Ten tekst jest zbyt krótki, by trzeba go było zawijać.")
# => "Ten tekst jest zbyt krótki, by trzeba go było zawijać. \n"

puts wrap("Ten tekst zostanie zawinięty.", 15)
```

```
# Ten tekst
# zostanie
# zawinięty.

puts wrap("Ten tekst zostanie zawinięty.", 20)
# Ten tekst zostanie
# zawinięty.


puts wrap("Być albo nie być – oto jest pytanie!",5)
# Być
# albo
# nie
# być –
# oto
# jest
# pytanie!
```

Dyskusja

W prezentowanym przykładzie zachowane zostało oryginalne formatowanie łańcucha, jednocześnie w kilku jego miejscach wstawione zostały znaki nowego wiersza. W efekcie uzyskaliśmy łańcuch zdatny do wyświetlenia w stosunkowo niewielkim obszarze ekranu.

```
poetry = %q{It is an ancient Mariner,
And he stoppeth one of three.
"By thy long beard and glittering eye,
Now wherefore stopp'st thou me?}

puts wrap(poetry, 20)
# It is an ancient
# Mariner,
# And he stoppeth one
# of three.
# "By thy long beard
# and glittering eye,
# Now wherefore
# stopp'st thou me?
```

Niekiedy jednak białe znaki nie są istotne, co więcej — zachowanie ich w łańcuchu powoduje pogorszenie końcowego rezultatu formatowania:

```
prose = %q{Czułem się tak samotny tego dnia, jak rzadko kiedy,
spoglądając apatycznie na deszcz padający za oknem. Jak długo jeszcze będzie
padać? W gazecie była prognoza pogody, ale któż w ogóle zadaje sobie trud
jej czytania?}

puts wrap(prose, 50)
# Czułem się tak samotny tego dnia, jak rzadko
# kiedy,
# spoglądając apatycznie na deszcz padający za
# oknem. Jak długo jeszcze będzie
# padać? W gazecie była prognoza pogody, ale któż w
# ogóle zadaje sobie trud
# jej czytania?
```

By zniwelować efekt „postrzępienia” tekstu, należałoby najpierw usunąć z niego istniejące znaki nowego wiersza. Należy w tym celu użyć innego wyrażenia regularnego:

```
def reformat_wrapped(s, width=78)
 s.gsub(/\s+/, " ").gsub(/(.{1,#{width}})( |\Z)/, "\\1\n")
end
```

Przetwarzanie sterowane wyrażeniami regularnymi jest jednak stosunkowo powolne; znacznie efektywniejszym rozwiązaniem byłoby podzielenie łańcucha na poszczególne słowa i złożenie z nich nowego łańcucha, podzielonego na wiersze nieprzekraczające określonej długości:

```
def reformat_wrapped(s, width=78)
  lines = []
  line = ""
  s.split(/\s+/).each do |word|
    if line.size + word.size >= width
      lines << line
      line = word
    elsif line.empty?
      line = word
    else
      line << " " << word
    end
  end
  lines << line if line
  return lines.join "\n"
end

puts reformat_wrapped(prose, 50)

# Czułem się tak samotny tego dnia, jak rzadko
# kiedy, spoglądając apatycznie na deszcz padający
# za oknem. Jak długo jeszcze będzie padać? W
# gazecie była prognoza pogody, ale któż w ogóle
# zadaje sobie trud jej czytania?
```

Patrz także

- W bibliotece `Facets Core` zdefiniowane są metody `String#word_wrap` i `String#word_wrap!`.

1.16. Generowanie następnika łańcucha

Problem

Należy wykonać iterację po ciągu łańcuchów zwiększających się alfabetycznie — w sposób podobny do iterowania po ciągu kolejnych liczb.

Rozwiązanie

Jeśli znany jest początkowy i końcowy łańcuch z zakresu objętego iteracją, można do tego zakresu (reprezentowanego jako obiekt `Range`) zastosować metodę `Range#each`:

```
('aa'..'ag').each { |x| puts x }
# aa
# ab
# ac
# ad
# ae
# af
# ag
```

Metodą generującą następnik danego łańcucha jest `String#succ`. Jeśli nie jest znany łańcuch, na którym należy skończyć iterowanie, można na bazie tej metody zdefiniować iterację nieskończoną, którą przerwie się w momencie spełnienia określonego warunku:

```
def endless_string_succession(start)
  while true
    yield start
    start = start.succ
  end
end
```

W poniższym przykładzie iteracja jest kończona w momencie, gdy dwa ostatnie znaki łańcucha są identyczne:

```
endless_string_succession('fol') do |x|
  puts x
  break if x[-1] == x[-2]
end
# fol
# fom
# fon
# foo
```

Dyskusja

Wyobraźmy sobie, że łańcuch jest czymś na kształt (uogólnionego) licznika przejechanych kilometrów — każdy znak łańcucha jest osobną pozycją tego licznika. Na każdej z pozycji mogą pojawiać się znaki *tylko jednego rodzaju*: cyfry, małe litery albo wielkie litery[3].

Następnikiem (*successor*) łańcucha jest łańcuch powstający w wyniku zwiększenia o 1 (inkrementacji) wskazania wspomnianego licznika. Rozpoczynamy od zwiększenia prawej skrajnej pozycji; jeśli spowoduje to jej „przekręcenie" na wartość początkową, zwiększamy o 1 sąsiednią pozycję z lewej strony — która też może się przekręcić, więc opisaną zasadę stosujemy rekurencyjnie:

```
'89999'.succ                    # => "90000"
'nzzzz'.succ                    # => "oaaaa"
```

Jeśli „przekręci" się *skrajna lewa* pozycja, dołączamy z lewej strony łańcucha nową pozycję tego samego rodzaju co ona i ustawiamy tę dodaną pozycję na wartość początkową:

```
'Zzz'.succ                      # => "AAaa"
```

W powyższym przykładzie skrajna lewa pozycja wyświetla wielkie litery; jej inkrementacja powoduje „przekręcenie" z wartości `Z` do wartości `A`, dodajemy więc z lewej strony łańcucha nową pozycję, także wyświetlającą wielkie litery, ustawiając ją na wartość początkową `A`.

Oto przykłady inkrementacji łańcuchów zawierających wyłącznie małe litery:

```
'z'.succ                        # => "aa"
'aa'.succ                       # => "ab"
'zz'.succ                       # => "aaa"
```

W przypadku wielkich liter sprawa ma się podobnie — należy pamiętać, że wielkie i małe litery nigdy nie występują razem na tej samej pozycji:

```
'AA'.succ                       # =>  "AB"
'AZ'.succ                       # =>  "BA"
```

[3] Ograniczamy się tylko do liter alfabetu angielskiego `a .. z` i `A .. Z` — *przyp. tłum.*

```
'ZZ'.succ                       # => "AAA"
'aZ'.succ                       # =>  "bA"
'Zz'.succ                       # => "AAa"
```

Inkrementowanie cyfr odbywa się w sposób naturalny — inkrementacja cyfry `9` oznacza jej „przekręcenie" na wartość `0`:

```
'foo19'.succ                    # => "foo20"
'foo99'.succ                    # => "fop00"
'99'.succ                       # => "100"
'9Z99'.succ                     # => "10A00"
```

Znaki niealfanumeryczne — czyli inne niż cyfry, małe litery i wielkie litery — są przy inkrementowaniu łańcucha *ignorowane* — wyjątkiem jest jednak sytuacja, gdy łańcuch składa się wyłącznie ze znaków tej kategorii. Umożliwia to inkrementowanie łańcuchów sformatowanych:

```
'10-99'.succ                    # => "11-00"
```

Jeśli łańcuch składa się wyłącznie ze znaków niealfanumerycznych, jego pozycje inkrementowane są zgodnie z uporządkowaniem znaków w kodzie ASCII; oczywiście w wyniku inkrementacji mogą pojawić się w łańcuchu znaki alfanumeryczne, wówczas kolejna jego inkrementacja odbywa się według reguł wcześniej opisanych.

```
'a-a'.succ                      # => "a-b"
'z-z'.succ                      # => "aa-a"
'Hello!'.succ                   # => "Hellp!"
%q{'zz'}.succ                   # => "'aaa'"
%q{z'zz'}.succ                  # => "aa'aa'"
'$$$$'.succ                     # => "$$$%"

s = '!@-'
13.times { puts s = s.succ }
# !@.
# !@/
# !@0
# !@1
# !@2
# ...
# !@8
# !@9
# !@10
```

Nie istnieje metoda realizująca funkcję odwrotną do metody `String#succ`. Zarówno twórca języka Ruby, jak i cała wspólnota jego użytkowników zgodni są co do tego, że wobec ograniczonego zapotrzebowania na taką metodę nie warto wkładać wysiłku w jej tworzenie, a zwłaszcza poprawną obsługę różnych warunków granicznych. Iterowanie po zakresie łańcuchów w kierunku malejącym najlepiej jest wykonywać, transformując ów zakres na tablicę i organizując iterację po tejże w kierunku malejących indeksów:

```
("a".."e").to_a.reverse_each { |x| puts x }
# e
# d
# c
# b
# a
```

Patrz także

- Receptura 2.15, „Generowanie sekwencji liczb".
- Receptura 3.4, „Iterowanie po datach".

1.17. Dopasowywanie łańcuchów za pomocą wyrażeń regularnych

Problem

Chcemy sprawdzić, czy dany łańcuch zgodny jest z pewnym wzorcem.

Rozwiązanie

Wzorce są zwykle definiowane za pomocą wyrażeń regularnych. Zgodność („pasowanie") łańcucha z wyrażeniem regularnym testowane jest przez operator =~.

```
string = 'To jest łańcuch 27-znakowy.'
if string =~ /([0-9]+)-character/ and $1.to_i == string.length
  "Tak, to jest łańcuch #$1-znakowy."
end
# "Tak, to jest łańcuch 27-znakowy."
```

Można także użyć metody `Regexp#match`:

```
match = Regexp.compile('([0-9]+)-znakowy').match(string)
if match && match[1].to_i == string.length
  "Tak, to jest łańcuch #{match[1]}-znakowy."
end
# "Tak, to jest łańcuch 27-znakowy."
```

Za pomocą instrukcji `case` można sprawdzić zgodność łańcucha z całym ciągiem wyrażeń regularnych:

```
string = "123"

case string
when /^[a-zA-Z]+$/
  "Litery"
when /^[0-9]+$/
  "Cyfry"
else
  "Zawartość mieszana"
end
# => "Cyfry"
```

Dyskusja

Wyrażenia regularne stanowią mało czytelny, lecz użyteczny minijęzyk umożliwiający dopasowywanie łańcuchów do wzorców oraz ekstrakcję podłańcuchów. Wyrażenia regularne wykorzystywane są od dawna przez wiele narzędzi uniksowych (jak `sed`), lecz to Perl był pierwszym uniwersalnym językiem zapewniającym ich obsługę. Obecnie wyrażenia regularne w stylu zbliżonym do wersji z Perla obecne są w większości nowoczesnych języków programowania.

W języku Ruby wyrażenia regularne inicjować można na wiele sposobów. Każda z poniższych konstrukcji daje w rezultacie taki sam obiekt klasy `Regexp`:

```
/cokolwiek/
Regexp.new("cokolwiek")
Regexp.compile("cokolwiek")
%r{ cokolwiek}
```

W wyrażeniach regularnych można używać następujących modyfikatorów:

`Regexp::IGNORECASE`	`i`	Przy dopasowywaniu nieistotna jest wielkość liter — małe litery utożsamiane są z ich wielkimi odpowiednikami.
`Regexp:MULTILINE`	`m`	Domyślnie dopasowywanie realizowane jest w odniesieniu do łańcucha mieszczącego się w jednym wierszu. Gdy użyty zostanie ten modyfikator, znaki nowego wiersza traktowane są na równi z innymi znakami łańcucha.
`Regexp::EXTENDED`	`x`	Użycie tego modyfikatora daje możliwość bardziej czytelnego zapisu wyrażenia regularnego, przez wypełnienie go białymi znakami i komentarzami.

Oto przykład wykorzystania wymienionych powyżej modyfikatorów w definicji wyrażenia regularnego:

```
/something/mxi
Regexp.new('something',
           Regexp::EXTENDED + Regexp::IGNORECASE + Regexp::MULTILINE)
%r{something}mxi
```

A oto efekt działania tychże modyfikatorów:

```
case_insensitive = /mangy/i
case_insensitive =~ "I'm mangy!"                    # => 4
case_insensitive =~ "Mangy Jones, at your service." # => 0

multiline = /a.b/m
multiline =~ "banana\nbanana"                       # => 5
/a.b/ =~ "banana\nbanana"                           # => nil
# Ale zwróć uwagę na to:
/a\nb/ =~ "banana\nbanana"                          # => 5

extended = %r{ \ was # Dopasowano " was"
               \s    # Dopasowano jeden biały znak
                a    # Dopasowano "a" }xi
extended =~ "What was Alfred doing here?"           # => 4
extended =~ "My, that was a yummy mango."           # => 8
extended =~ "It was\n\n\na fool's errand"           # => nil
```

Patrz także

- Książka Jeffreya Friedla *Mastering Regular Expressions*[4] dostarcza eleganckiego i zwięzłego wprowadzenia w tematykę wyrażeń regularnych, ilustrowanego wieloma praktycznymi przykładami.
- Witryna RegExLib.com (*http://regexlib.com/default.aspx*) jest obszerną bazą wyrażeń regularnych, wyposażoną w wyszukiwarkę.
- Przewodnik po wyrażeniach regularnych i ich wykorzystywaniu w języku Ruby dostępny jest pod adresem *http://www.regular-expressions.info/ruby.html*.
- Informacje na temat klasy `Regexp` możesz uzyskać za pomocą polecenia `ri Regexp`.
- Receptura 1.19, „Weryfikacja poprawności adresów e-mailowych”.

[4] Wydanie polskie: *Wyrażenia regularne*, wyd. Helion 2001 (*http://helion.pl/ksiazki/wyrare.htm*) — *przyp. tłum.*

1.18. Zastępowanie wielu wzorców w pojedynczym przebiegu

Problem

Chcemy wykonać kilka operacji typu „znajdź i zamień", sterowanych oddzielnymi wyrażeniami regularnymi — równolegle, w pojedynczym przejściu przez łańcuch.

Rozwiązanie

Musimy użyć metody `Regexp.union` do zagregowania poszczególnych wyrażeń regularnych w pojedyncze wyrażenie, pasujące do każdego z wyrażeń cząstkowych. Zagregowane wyrażenie musimy następnie przekazać jako parametr metody `String#gsub` wraz z blokiem kodowym bazującym na obiekcie `MatchData`. Wiedząc, do którego z wyrażeń cząstkowych przyporządkować można znalezioną frazę, możemy wybrać odpowiednią frazę zastępującą:

```
class String
  def mgsub(key_value_pairs=[].freeze)
    regexp_fragments = key_value_pairs.collect { |k,v| k }
    gsub(Regexp.union(*regexp_fragments)) do |match|
      key_value_pairs.detect{|k,v| k =~ match}[1]
    end
  end
end
```

Oto prosty przykład użycia metody `mgsub`:

```
"GO HOME!".mgsub([[/.*GO/i, 'Home'], [/home/i, 'is where the heart is']])
# => "Home is where the heart is!"
```

W powyższym przykładzie żądamy zamiany dowolnego ciągu kończącego się na `GO` (bez względu na wielkość liter) na ciąg `Home`, zaś ciągu `Home` (bez względu na wielkość liter) na ciąg `is where the heart is`.

W poniższym przykładzie zamieniamy wszystkie litery na znak #, a każdy znak # na literę `P`:

```
"To jest liczba #123".mgsub([[/[a-z]/i, '#'], [/#/, 'P']])
# => "#### ## ###### P123"
```

Dyskusja

Wydawałoby się, że naiwne podejście polegające na sukcesywnym wywołaniu metody `gsub` dla każdej operacji „znajdź i zamień" da identyczny efekt i tylko efektywnością ustępować będzie rozwiązaniu wyżej opisanemu. Jest jednak inaczej, o czym możemy się przekonać, spoglądając na poniższe przykłady:

```
"GO HOME!".gsub(/.*GO/i, 'Home').gsub(/home/i, 'is where the heart is')
# => "is where the heart is is where the heart is!"

"To jest liczba #123".gsub(/[a-z]/i, '#').gsub(/#/, 'P')
# => "PP PPPP PPPPPP P123"
```

Przyczyna rozbieżności z rozwiązaniem „równoległym" nie jest żadną tajemnicą: otóż w obydwu przypadkach materiałem wejściowym dla drugiego wywołania metody `gsub` jest *wynik*

jej pierwszego wywołania. W wariancie równoległym natomiast obydwa wywołania metody `gsub` operują na *łańcuchu oryginalnym*. W pierwszym przypadku można zniwelować ową interferencję, zamieniając kolejność operacji, w drugim jednak nawet i to nie pomoże.

Do metody `mgsub` można przekazać także hasz, w którym poszukiwane frazy są kluczami, a frazy zastępujące — wartościami. Nie jest to jednak rozwiązanie bezpieczne, bowiem elementy hasza są z natury nieuporządkowane i w związku z tym kolejność zastępowania fraz wymyka się spod kontroli. Znacznie lepszym wyjściem byłoby użycie tablicy elementów typu „klucz-wartość". Poniższy przykład z pewnością ułatwi zrozumienie tego problemu:

```
"between".mgsub(/ee/ => 'AA', /e/ => 'E')        # Zły kod
# => "bEtwEEn"

"between".mgsub([[/ee/, 'AA'], [/e/, 'E']])      # Dobry kod
# => "bEtwAAn"
```

W drugim przypadku najpierw wykonywane jest pierwsze zastępowanie. W pierwszym przypadku jest ono wykonywane jako drugie i szukana fraza nie zostaje znaleziona — to jedna z osobliwości implementacji haszów w języku Ruby.

Jeśli efektywność programu jest czynnikiem krytycznym, należy zastanowić się nad inną implementacją metody `mgsub`. Im więcej bowiem fraz do znalezienia i zastąpienia, tym dłużej trwać będzie cała operacja, ponieważ metoda `detect` wykonuje sprawdzenie dla każdego wyrażenia regularnego i dla każdej znalezionej frazy.

Patrz także

- Receptura 1.17, „Dopasowywanie łańcuchów za pomocą wyrażeń regularnych".
- Czytelnikom, którym zagadkowa wydaje się składnia `Regexp.union(*regexp_fragments)`, polecamy przestudiowanie receptury 8.11, „Metody wywoływane ze zmienną liczbą argumentów".

1.19. Weryfikacja poprawności adresów e-mailowych

Problem

Chcemy sprawdzić, czy podany adres e-mailowy jest poprawny.

Rozwiązanie

Oto kilka przykładowych adresów e-mail — poprawnych

```
test_addresses = [ # Poniższe adresy czynią zadość specyfikacji RFC822.
                   'joe@example.com', 'joe.bloggs@mail.example.com',
                   'joe+ruby-mail@example.com', 'joe(and-mary)@example.museum',
                   'joe@localhost',
```

i niepoprawnych

```
# Poniższe adresy są niezgodne ze specyfikacją RFC822
'joe', 'joe@', '@example.com',
'joe@example@example.com',
'joe and mary@example.com' ]
```

Oto kilka przykładowych wyrażeń regularnych filtrujących błędne adresy e-mailowe. Pierwsze z nich ogranicza się do bardzo elementarnej kontroli.

```
valid = '[^ @]+' # Wyeliminowanie znaków bezwzględnie niedopuszczalnych w adresie e-mail
username_and_machine = /^#{valid}@#{valid}$/

test_addresses.collect { |i| i =~ username_and_machine }
# => [0, 0, 0, 0, 0, nil, nil, nil, nil, nil]
```

Drugie z wyrażeń eliminuje adresy typowe dla sieci lokalnej, w rodzaju `joe@localhost` — większość aplikacji nie zezwala na ich używanie.

```
username_and_machine_with_tld = /^#{valid}@#{valid}\.#{valid}$/

test_addresses.collect { |i| i =~ username_and_machine_with_tld }
# => [0, 0, 0, 0, nil, nil, nil, nil, nil, nil]
```

Niestety, jak za chwilę zobaczymy, prawdopodobnie poszukujemy rozwiązania *nie tego* problemu.

Dyskusja

Większość systemów weryfikacji adresów e-mailowych opiera swe funkcjonowanie na naiwnych wyrażeniach regularnych, podobnych do prezentowanych powyżej. Niestety, wyrażenia takie bywają często zbyt rygorystyczne, wskutek czego zdarza się, że poprawny adres zostaje odrzucony. Jest to powszechna przyczyna frustracji użytkowników posługujących się nietypowymi adresami w rodzaju *joe(and-mary)@example.museum* oraz użytkowników wykorzystujących w swych adresach specyficzne cechy systemu e-mail (*joe+ruby-mail@example.com*). Prezentowane powyżej wyrażenia regularne cierpią na dokładnie odwrotną przypadłość — nie kwestionując nigdy adresów poprawnych, akceptują niektóre niepoprawne.

Dlaczego więc nie stworzyć (publicznie znanego) wyrażenia regularnego, które z zadaniem weryfikacji adresów e-mailowych poradzi sobie zawsze? Otóż dlatego, że być może wyrażenie takie wcale nie istnieje — definicji składni adresu e-mailowego zarzucić można wszystko, tylko nie prostotę. Haker języka Perl, Paul Warren, w stworzonym przez siebie module `Mail::RFC822:Address` zdefiniował wyrażenie regularne składające się z 6343 znaków, lecz nawet ono wymaga przetwarzania wstępnego dla absolutnie (w zamierzeniu) bezbłędnej weryfikacji adresu. Wyrażenia tego można użyć bez zmian w języku Ruby — zainteresowani Czytelnicy mogą znaleźć je w katalogu `Mail-RFC822-Address-0.3` na CD-ROM-ie dołączonym do niniejszej książki.

Weryfikuj prawdziwość, nie poprawność

Jednak najbardziej nawet wyszukane wyrażenie regularne nie potrafi zapewnić nic więcej niż tylko weryfikację *składniowej poprawności* adresu. Poprawność składniowa nie oznacza wcale, że dany adres jest *istniejącym* adresem.

Przy wpisywaniu adresu łatwo można się pomylić, wskutek czego poprawny adres zamienia się w (także poprawny) adres kogo innego (*joe@example.com*). Adres *!@* jest składniowo poprawny, lecz nikt na świecie go nie używa. Nawet zbiór domen najwyższego poziomu (*top-level domains*) też nie jest ustalony i jako taki nie może być przedmiotem weryfikacji w oparciu o statyczną listę. Reasumując — weryfikacja poprawności składniowej adresu e-mail jest tylko małą częścią rozwiązania rzeczywistego problemu.

Jedynym sposobem stwierdzenia poprawności adresu jest udane wysłanie listu na ów adres. O tym, czy adres ten jest *właściwy*, możemy przekonać się dopiero po otrzymaniu odpowiedzi od adresata. Jak widać, nietrudny na pozór problem wymaga wcale niemało zachodu przy tworzeniu aplikacji.

Nie tak dawno jeszcze adres e-mailowy użytkownika związany był nierozerwalnie w jego tożsamością w sieci, bo przydzielany był przez dostawcę internetowego (ISP). Przestało tak być w dobie poczty webowej, gdzie każdy użytkownik może sobie przydzielić tyle adresów, ile tylko zechce. W efekcie weryfikacja poprawności adresów nie jest w stanie zapobiec ani dublowaniu kont, ani też antyspołecznym zachowaniom w sieci (i wątpliwe jest, czy kiedykolwiek mogła).

Nie oznacza to bynajmniej, że weryfikacja składni adresu e-mailowego jest całkowicie bezużyteczna, albo że nie jest problemem niezamierzone zniekształcenie wpisywanego adresu („literówka"). Aby usprawnić pracę użytkownika aplikacji wpisującego adres e-mailowy, bez obawy o kwestionowanie poprawnych adresów, możesz zrobić trzy następujące rzeczy oprócz weryfikacji adresu w oparciu o prezentowane wcześniej wyrażenia regularne:

1. Użyj drugiego, naiwnego i bardziej restrykcyjnego wyrażenia regularnego, lecz w przypadku stwierdzenia niepoprawności adresu ogranicz się do wypisania komunikatu ostrzegawczego, nie blokując użytkownikowi możliwości użycia tego adresu. Nie jest to tak użyteczne, jak mogłoby się wydawać, bo adres będący wynikiem pomyłki literowej jest często także adresem poprawnym składniowo (po prostu jedna litera zamieniona zostaje na inną).

```
def probably_valid?(email)
 valid = '[A-Za-z\d.+-]+' # Znaki powszechnie spotykane w adresach
 (email =~ /#{valid}@#{valid}\.#{valid}/) == 0
end

# Wyniki weryfikacji zgodne z oczekiwaniami
probably_valid? 'joe@example.com'               # => true
probably_valid? 'joe+ruby-mail@example.com'     # => true
probably_valid? 'joe.bloggs@mail.example.com'   # => true
probably_valid? 'joe@examplecom'                # => false
probably_valid? 'joe+ruby-mail@example.com'     # => true
probably_valid? 'joe@localhost'                 # => false


# Adres poprawny, lecz kwestionowany przez metodę probably_valid?
probably_valid? 'joe(and-mary)@example.museum'  # => false


# Adres składniowo poprawny, lecz ewidentnie błędny
probably_valid? 'joe@example.cpm'               # => true
```

2. Wydziel adres serwera z adresu e-mailowego (np. *example.com*) i sprawdź (za pomocą DNS), czy serwer ten zapewnia obsługę poczty (tzn. czy da się z niego odczytać rekord MX DNS). Poniższy fragment kodu zdolny jest wychwycić większość pomyłek w zapisie adresu serwera, co jednak nie chroni przed podaniem nazwy nieistniejącego użytkownika. Ponadto ze względu na złożoność samego dokumentu RFC822 nie można zagwarantować, że analiza adresu serwera zawsze będzie przeprowadzona bezbłędnie:

```
require 'resolv'
def valid_email_host?(email)
  hostname = email[(email =~ /@/)+1..email.length]
  valid = true
```

```
    begin
      Resolv::DNS.new.getresource(hostname, Resolv::DNS::Resource::IN::MX)
    rescue Resolv::ResolvError
      valid = false
    end
    return valid
  end

  # example.com jest adresem rzeczywistej domeny, lecz jej serwer
  # nie obsługuje poczty.
  valid_email_host?('joe@example.com')          # => false

  # lcqkxjvoem.mil nie jest adresem istniejącej domeny.
  valid_email_host?('joe@lcqkxjvoem.mil')       # => false

  # domena oreilly.com istnieje i jej serwer zapewnia obsługę poczty, jednakże
  # uzytkownik 'joe' może nie być zdefiniowany na tym serwerze.
  valid_email_host?('joe@oreilly.com')          # => true
```

3. Wyślij list na adres wpisany przez użytkownika aplikacji, z prośbą do adresata o potwierdzenie poprawności adresu. Aby ułatwić adresatowi zadanie, można w treści listu umieścić stosowny URL (gotowy do kliknięcia) z odpowiednim komentarzem. Jest to jedyny sposób upewnienia się, że użyto właściwego adresu. Powrócimy do tej kwestii w recepturach 14.5 i 15.19.

   Mimo iż rozwiązanie to stanowczo podnosi poprzeczkę wymagań wobec programisty tworzącego aplikację, może okazać się nieskuteczne z bardzo prostej przyczyny — rozmaitych sposobów walki z niechcianą pocztą. Użytkownik może zdefiniować filtr, który zaklasyfikuje wspomnianą wiadomość jako niechcianą (*junk*), bądź też generalnie odrzucać wszelką pocztę pochodzącą z nieznanego źródła. Jeżeli jednak weryfikacja adresów e-mail nie jest dla aplikacji zagadnieniem krytycznym, opisywane sposoby tej weryfikacji powinny okazać się wystarczające.

Patrz także

- Receptura 14.5, „Wysyłanie poczty elektronicznej”.
- Receptura 15.19, „Przesyłanie wiadomości pocztowych za pomocą aplikacji Rails”.
- Wspomniane wcześniej kolosalne wyrażenie regularne autorstwa Paula Warrena dostępne jest do pobrania pod adresem *http://search.cpan.org/~pdwarren/Mail-RFC822-Address-0.3/Address.pm*.

1.20. Klasyfikacja tekstu za pomocą analizatora bayesowskiego

Problem

Mając dany fragment tekstu, chcemy dokonać jego klasyfikacji — na przykład zdecydować, czy otrzymany list można potraktować jako spam, bądź czy zawarty w liście dowcip jest naprawdę śmieszny.

Rozwiązanie

Można w tym celu skorzystać w biblioteki `Classifier` Lucasa Carlsona, dostępnej w gemie `classifier`. W bibliotece tej znajduje się naiwny klasyfikator bayesowski oraz klasyfikator wykorzystujący bardziej zaawansowaną technikę ukrytego indeksowania semantycznego (LSI — *Latent Semantic Indexing*).

Interfejs naiwnego klasyfikatora bayesowskiego jest elementarny: tworzy się obiekt `Classifier::Bayes` z określeniem rodzaju klasyfikacji jako parametrem, po czym dokonuje się „uczenia" tegoż obiektu za pomocą fragmentów tekstu o znanym wyniku klasyfikacji.

```
require 'rubygems'
require 'classifier'

classifier = Classifier::Bayes.new('Spam', 'Not spam')

classifier.train_spam 'are you in the market for viagra? we sell viagra'
classifier.train_not_spam 'hi there, are we still on for lunch?'
```

Następnie można przekazać do obiektu nieznany tekst i zaobserwować wynik klasyfikacji:

```
classifier.classify "we sell the cheapest viagra on the market"
# => "Spam"
classifier.classify "lunch sounds great"
# => "Not spam"
```

Dyskusja

Bayesowska analiza tekstu opiera się na rachunku prawdopodobieństwa. Klasyfikator w procesie uczenia się analizuje wzorcowy tekst w rozbiciu na słowa, zapamiętując prawdopodobieństwo występowania każdego z tych słów w podanej kategorii. W prostym przykładzie podanym w Rozwiązaniu rozkład tego prawdopodobieństwa może być opisany przez następujące hasze:

```
classifier
# => #<Classifier::Bayes:0xb7cec7c8
#      @categories={:"Not spam"=>
#                     { :lunch=>1, :for=>1, :there=>1,
#                       :"?"=>1, :still=>1, :","=>1 },
#                   :Spam=>
#                     { :market=>1, :for=>1, :viagra=>2, :"?"=>1, :sell=>1 }
#                   },
#      @total_words=12>
```

Hasze te wykorzystywane są następnie do budowania statystyki analizowanego (nieznanego) tekstu. Zwróćmy uwagę, że słowo „viagra" dwukrotnie wystąpiło we wzorcowym tekście zaliczonym do kategorii „Spam", słowo „sell" — jednokrotnie w kategorii „Spam", zaś słowo „for" — jednokrotnie w obydwu kategoriach, „Spam' i „Not spam". Oznacza to, że wystąpienie słowa „for" w analizowanym (nieznanym) tekście nie daje żadnej przesłanki klasyfikacyjnej, wystąpienie słowa „sell" daje pewną przesłankę w kierunku kategorii „Spam", zaś wystąpienie słowa „viagra" stanowi dwukrotnie silniejszą przesłankę w tym samym kierunku.

Im większa objętość tekstu wzorcowego przeanalizowana zostanie na etapie uczenia się klasyfikatora, tym generalnie trafniejszych rezultatów można się spodziewać w procesie rozpoznawania kategorii nieznanego tekstu. Wynik tej klasyfikacji — zaproponowany przez klasyfikator bądź skorygowany przez użytkownika — może być wykorzystany jako kolejna porcja danych „uczących".

Bieżący „dorobek” klasyfikatora w procesie uczenia się można zapisać na dysku do późniejszego użytku, za pomocą Madeleine (patrz receptura 13.3).

Klasyfikator bayesowski może rozpoznawać dowolną liczbę kategorii. Kategorie „Spam” i „Not spam” należą do najczęściej wykorzystywanych, ale liczba kategorii nie jest bynajmniej ograniczona do dwóch. Można także wykorzystać rodzimą metodę `train` zamiast specyficznych metod `train_<kategoria>`. Klasyfikator użyty w poniższym przykładzie wykorzystuje tę rodzimą metodę i dokonuje klasyfikacji tekstu do jednej z trzech kategorii:

```
classifier = Classifier::Bayes.new('Interesting', 'Funny', 'Dramatic')

classifier.train 'Interesting', "Leaving reminds us of what we can part
  with and what we can't, then offers us something new to look forward
  to, to dream about."
classifier.train 'Funny', "Knock knock. Who's there? Boo boo. Boo boo
  who? Don't cry, it is only a joke."
classifier.train 'Dramatic', 'I love you! I hate you! Get out right
  now.'

classifier.classify 'what!'
# => "Dramatic"
classifier.classify "who's on first?"
# => "Funny"
classifier.classify 'perchance to dream'
# => "Interesting"
```

Za pomocą metody `untrain` można anulować efekt obecności danego słowa we wzorcowym tekście określonej kategorii, co okazuje się nieodzowne w przypadku niereprezentatywnego tekstu wzorcowego bądź błędnego typowania:

```
classifier.untrain_funny "boo"
classifier.untrain "Dramatic", "out"
```

Patrz także

- Receptura 13.3, „Utrwalanie obiektów z wykorzystaniem biblioteki Madeleine”.
- Plik `README` biblioteki `Classifier` zawiera przykład klasyfikatora LSI.
- `Bishop` (*http://bishop.rubyforge.org/*) jest innym klasyfikatorem bayesowskim, przeniesionym z Python Reverend i dostępnym w gemie `bishop`.
- *http://pl.wikipedia.org/wiki/Naiwny_klasyfikator_bayesowski*.
- *http://en.wikipedia.org/wiki/Latent_Semantic_Analysis*.