helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

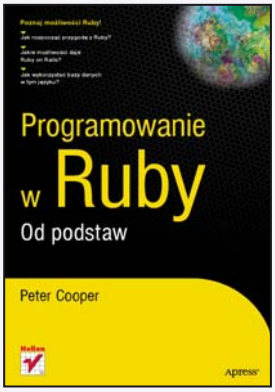
Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

Czytelnia

- Fragmenty książek online

Programowanie w Ruby. Od podstaw



Autor: Peter Cooper
Tłumaczenie: Daniel Kaczmarek
ISBN: 978-83-246-1953-5
Tytuł oryginału: Beginning Ruby: From Novice to Professional
Format: 158×235, stron: 608

Poznaj możliwości Ruby!

- Jak rozpocząć przygodę z Ruby?
- Jakie możliwości daje Ruby on Rails?
- Jak wykorzystać bazy danych w tym języku?

Co sprawiło, że nieufne zwykle środowisko programistów przyjęło Ruby z entuzjazmem? Jakie to unikalne możliwości posiada ów język? Odpowiedź jest prosta – jego główne atuty to przejrzysta i elastyczna składnia z wbudowanymi wyrażeniami regularnymi, automatyczne oczyszczanie pamięci oraz przeciążanie operatorów. Ponadto skupiona wokół Ruby ogromna i chętna do pomocy społeczność sprawia, że to rozwiązanie staje się jeszcze bardziej atrakcyjne i rozwojowe. „Programowanie w Ruby. Od podstaw" to książka, która pomoże Ci zorientować się w specyfice tego języka.

Zanim rozpoczniesz przygodę z Ruby, warto dowiedzieć się, jak przygotować swoje środowisko pracy, oraz poznać podstawowe zagadnienia związane z programowaniem obiektowym. Po krótkim wstępie przejdziesz do konkretów – zapoznasz się ze składnią, podstawowymi konstrukcjami oraz metodami sterowania przepływem. Zdobędziesz także wiedzę na temat wykonywania operacji na plikach i bazach danych oraz możliwości Ruby w zastosowaniach sieciowych. Nauczysz się tworzyć strukturę projektu, przygotowywać dokumentację, wyszukiwać przydatne biblioteki. Z pewnością zainteresuje Cię rozdział poświęcony Ruby on Rails – szkieletowi aplikacyjnemu, który niewątpliwie miał swój wpływ na wzrost popularności tego języka. To wszystko pozwoli Ci na swobodne wykorzystanie możliwości języka Ruby w codziennej pracy!

- Przygotowanie środowiska pracy
- Podstawowe zagadnienia z dziedziny programowania obiektowego
- Składnia i konstrukcje języka Ruby
- Sterowanie przepływem
- Tworzenie dokumentacji
- Obsługa błędów, testowanie i debugowanie aplikacji
- Obsługa plików
- Wykorzystanie baz danych
- Możliwości i zastosowanie Ruby on Rails
- Wykorzystanie zasobów sieci Internet w Ruby
- Obsługa sieci, gniazd i demonów
- Przydatne pakiety i biblioteki w Ruby

Poznaj Ruby – od podstaw do perfekcji!

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2010

Spis treści

Przedmowa **15**

O autorze **19**

O redaktorach technicznych **21**

Podziękowania **23**

Wprowadzenie **25**

Część I Podstawy **27**

Rozdział 1. Pierwsze kroki. Instalacja języka Ruby **29**
Instalowanie języka Ruby 30
Windows 30
Apple Mac OS X 33
Linux 35
Inne platformy 37
Podsumowanie 39

Rozdział 2. Programowanie == zabawa. Krótki przegląd języka Ruby i zasad obiektowości **41**
Pierwsze kroki 42
irb: interaktywny Ruby 42
Ruby to angielski dla komputerów 43
Dlaczego Ruby jest doskonałym językiem programowania? 43
Ścieżki umysłu 44
Zapisywanie pomysłów w kodzie źródłowym języka Ruby 46
Jak Ruby interpretuje rzeczy za pomocą obiektów i klas 47
Tworzenie osoby 47
Podstawowe zmienne 49
Od osób do zwierząt 50
Wszystko jest obiektem 53
Metody klasy Kernel 55
Przekazywanie danych do metod 55
Metody klasy String 56
Korzystanie z języka Ruby z pominięciem orientacji obiektowej 58
Podsumowanie 59

Rozdział 3. Podstawowe konstrukcje języka Ruby: dane, wyrażenia i przepływ sterowania ... 61
Liczby i wyrażenia ... 61
Podstawowe wyrażenia ... 62
Zmienne ... 62
Operatory i wyrażenia porównywania ... 64
Iterowanie przez liczby przy użyciu bloków i iteratorów ... 65
Liczby zmiennoprzecinkowe ... 67
Stałe ... 68
Tekst i ciągi znaków ... 69
Literały ciągów znaków ... 69
Wyrażenia z ciągami znaków ... 71
Interpolacja ... 72
Metody klasy String ... 73
Wyrażenia regularne i manipulowanie ciągami znaków ... 74
Tablice i listy ... 80
Tablice podstawowe ... 80
Dzielenie ciągów znaków na tablice ... 82
Iterowanie w tablicy ... 83
Inne metody stosowane do działania na tablicach ... 84
Tablice asocjacyjne ... 86
Podstawowe metody do obsługi tablic asocjacyjnych ... 86
Tablice asocjacyjne z tablicami asocjacyjnymi ... 88
Przepływ sterowania ... 89
Instrukcje if i unless ... 89
Operator trójargumentowy ?: ... 90
Instrukcje elsif oraz case ... 91
Instrukcje while i until ... 92
Bloki kodu ... 93
Inne przydatne konstrukcje ... 95
Daty i czas ... 95
Duże liczby ... 98
Zakresy ... 100
Symbole ... 101
Przekształcanie klas ... 102
Podsumowanie ... 103

Rozdział 4. Prosta aplikacja w języku Ruby ... 105
Praca z plikami zawierającymi kod źródłowy ... 105
Tworzenie pliku testowego ... 106
Testowy plik z kodem źródłowym ... 107
Wykonywanie kodu źródłowego ... 108
Pierwsza aplikacja: analizator tekstu ... 111
Podstawowe funkcje aplikacji ... 111
Implementacja aplikacji ... 112
Uzyskanie przykładowego tekstu ... 112
Ładowanie plików tekstowych i zliczanie wierszy ... 113
Zliczanie znaków ... 114
Zliczanie słów ... 115
Zliczanie zdań i akapitów ... 117
Obliczenie wartości średnich ... 119
Kod źródłowy aplikacji ... 119

Dodatkowe funkcje aplikacji ... 120
Procent słów „znaczących” ... 120
Podsumowanie prezentujące zdania „znaczące” ... 122
Analiza plików innych niż text.txt ... 124
Program w wersji finalnej ... 125
Podsumowanie ... 127

Rozdział 5. Ekosystem języka Ruby ... 129
Historia języka Ruby ... 129
Kraj Wschodzącego Słońca ... 130
Korzenie języka Ruby ... 131
Inwazja na Zachód ... 132
Ruby on Rails ... 133
Po co utworzono platformę Rails ... 134
Jak zdobyto Web 2.0 ... 135
Kultura Open Source ... 135
Na czym polega ruch Open Source? ... 136
Gdzie i jak uzyskać pomoc? ... 137
Listy dystrybucyjne ... 137
Grupy dyskusyjne Usenet ... 138
Internet Relay Chat (IRC) ... 138
Dokumentacja ... 139
Fora ... 140
Dołączanie do społeczności ... 140
Świadczenie pomocy innym ... 141
Dzielenie się kodem źródłowym ... 141
Blogi ... 142
Podsumowanie ... 143

Część II Rdzeń języka Ruby ... 145

Rozdział 6. Klasy, obiekty i moduły ... 147
Po co używać orientacji obiektowej? ... 147
Podstawy orientacji obiektowej ... 150
Zmienne lokalne, globalne, obiektu i klasy ... 151
Metody klasy a metody obiektu ... 155
Dziedziczenie ... 157
Nadpisywanie istniejących metod ... 159
Refleksja i odkrywanie metod dostępnych w obiektach ... 161
Enkapsulacja ... 162
Wielopostaciowość ... 167
Klasy zagnieżdżone ... 168
Zasięg stałych ... 169
Moduły, przestrzenie nazw i włączanie kodu ... 170
Przestrzenie nazw ... 170
Włączanie kodu ... 173
Obiektowa tekstowa gra przygodowa ... 180
Idea gry ... 180
Klasy początkowe ... 180
Struktury: proste klasy danych ... 182
Tworzenie komnat ... 184
Uruchamianie gry ... 185
Podsumowanie ... 188

Rozdział 7. Projekty i biblioteki ... 191
Projekty i wykorzystanie kodu z innych plików ... 191
Proste dołączanie pliku ... 192
Dołączanie kodu z innych katalogów ... 193
Logika programu a dołączanie kodu ... 194
Dołączenia zagnieżdżone ... 195
Biblioteki ... 195
Biblioteki standardowe ... 196
RubyGems ... 198
Podsumowanie ... 207

Rozdział 8. Tworzenie dokumentacji, obsługa błędów, debugowanie i testowanie ... 209
Tworzenie dokumentacji ... 209
Generowanie dokumentacji przy użyciu RDoc ... 210
Techniki pracy z RDoc ... 212
Debugowanie i błędy ... 215
Wyjątki i obsługa błędów ... 215
Metody catch i throw ... 219
Debuger języka Ruby ... 220
Testowanie ... 223
Filozofia programowania sterowanego testami ... 224
Testy jednostkowe ... 226
Inne asercje biblioteki Test::Unit ... 228
Testy wzorcowe i profilowanie ... 229
Proste wzorcowe testy wydajności ... 230
Profilowanie ... 232
Podsumowanie ... 233

Rozdział 9. Pliki i bazy danych ... 237
Wejście i wyjście ... 237
Dane wejściowe z klawiatury ... 238
Wejście i wyjście do pliku ... 239
Proste bazy danych ... 253
Bazy danych w postaci plików tekstowych ... 253
Przechowywanie obiektów i struktur danych ... 256
Relacyjne bazy danych i język SQL ... 259
Idea relacyjnych baz danych ... 260
Wielka czwórka: MySQL, PostgreSQL, Oracle i SQLite ... 261
Instalacja SQLite ... 262
Krótki kurs podstawowych czynności w bazie danych i języka SQL ... 262
Korzystanie z SQLite w języku Ruby ... 267
Łączenie się z innymi systemami zarządzania bazami danych ... 271
ActiveRecord. Krótki opis ... 276
Podsumowanie ... 277

Rozdział 10. Wdrażanie aplikacji i bibliotek języka Ruby ... 281
Dystrybucja prostych programów napisanych w Ruby ... 281
Wiersz ze ścieżką dostępu do interpretera ... 283
Skojarzone typy plików w systemie Windows ... 284
„Kompilowanie” kodu języka Ruby ... 284
Wykrywanie środowiska wykonawczego języka Ruby ... 286
Łatwe wykrywanie systemu operacyjnego za pomocą zmiennej RUBY_PLATFORM ... 287
Zmienne środowiskowe ... 287
Odczytywanie argumentów wiersza poleceń ... 289

Udostępnianie i dystrybuowanie bibliotek języka Ruby w postaci pakietów gem 290
Tworzenie pakietu gem 291
Dystrybucja pakietu gem 295
RubyForge 295
Wdrażanie aplikacji Ruby jako usług zdalnych 296
Skrypty CGI 296
Serwery HTTP ogólnego przeznaczenia 299
Zdalne wywołania procedur 303
Podsumowanie 307

Rozdział 11. Zaawansowane mechanizmy języka Ruby 309
Dynamiczne wykonywanie kodu 309
Wiązania 310
Inne postacie metody eval 311
Tworzenie własnej wersji attr_accessor 313
Wykonywanie innych programów z poziomu języka Ruby 314
Odczytywanie wyników działania innych programów 314
Przekazywanie wykonania do innego programu 315
Wykonywanie dwóch programów jednocześnie 315
Komunikacja z innym programem 316
Bezpieczne przetwarzanie danych i niebezpiecznych metod 317
Niebezpieczne dane i obiekty 317
Poziomy zabezpieczeń 320
Praca z systemem Microsoft Windows 321
Korzystanie z API systemu Windows 321
Kontrolowanie programów systemu Windows 323
Wątki 325
Podstawowe czynności z wykorzystaniem wątków Ruby 325
Zaawansowane operacje z wykorzystaniem wątków 326
Biblioteka RubyInline 328
Po co używać C jako języka wplatanego? 329
Tworzenie prostej metody lub funkcji 329
Testy wzorcowe wydajności kodu C i Ruby 331
Obsługa Unicode i UTF-8 332
Podsumowanie 335

Rozdział 12. Zaawansowana aplikacja w języku Ruby 337
Implementacja bota 337
Czym jest bot? 337
Dlaczego bot? 339
Jak implementuje się bota? 339
Biblioteka narzędzi do przetwarzania tekstu 340
Implementacja biblioteki WordPlay 341
Testowanie biblioteki 346
Kod źródłowy biblioteki WordPlay 349
Implementacja głównego modułu bota 351
Cykl życia programu i jego elementy 352
Dane dla bota 353
Implementacja klasy Bot oraz mechanizmu ładowania danych 357
Metoda response_to 358
Zabawa z botem 363
Kod źródłowy najważniejszych elementów bota 366
bot.rb 367
basic_client.rb 369

Rozszerzanie bota ... 369
Wykorzystanie plików tekstowych jako źródła danych wejściowych do rozmowy ... 370
Udostępnienie bota w sieci WWW ... 370
Rozmowy między dwoma botami ... 373
Podsumowanie ... 374

Część III Ruby w sieci ... 375

Rozdział 13. Ruby on Rails: zabójcza aplikacja języka Ruby ... 377
Pierwsze kroki ... 377
Czym jest platforma Rails i po co się jej używa? ... 378
Instalacja platformy Rails ... 379
Bazy danych ... 381
Implementacja pierwszej aplikacji Rails ... 381
Tworzenie pustej aplikacji Rails ... 381
Inicjalizacja bazy danych ... 385
Tworzenie modelu i migracji ... 388
Scaffolding ... 392
Kontrolery i widoki ... 395
Routing ... 404
Relacje między modelami ... 405
Sesje i filtry ... 407
Pozostałe funkcje ... 409
Makiety ... 409
Testowanie ... 412
Moduły rozszerzające ... 413
Dodatkowe zasoby i przykładowe aplikacje ... 414
Witryny i samouczki ... 415
Przykładowe aplikacje Rails ... 415
Podsumowanie ... 416

Rozdział 14. Ruby i internet ... 419
HTTP i sieć WWW ... 419
Pobieranie stron WWW ... 420
Generowanie stron WWW i kodu HTML ... 427
Przetwarzanie treści WWW ... 432
Obsługa poczty elektronicznej ... 437
Odczytywanie poczty przy użyciu POP3 ... 437
Wysyłanie wiadomości pocztowych przy użyciu SMTP ... 439
Wysyłanie wiadomości pocztowych przy użyciu ActionMailer ... 441
Transfer plików przy użyciu protokołu FTP ... 442
Nawiązywanie połączenia i wykonywanie podstawowych czynności FTP ... 443
Pobieranie plików ... 444
Ładowanie plików na serwer ... 445
Podsumowanie ... 447

Rozdział 15. Obsługa sieci, gniazd i demonów ... 449
Najważniejsze mechanizmy sieciowe ... 449
TCP i UDP ... 450
Adresy IP i DNS ... 450
Podstawowe operacje sieciowe ... 451
Sprawdzanie dostępności komputera lub usługi ... 451
Wykonywanie zapytań DNS ... 453
Nawiązywanie bezpośredniego połączenia z serwerem TCP ... 455

Serwery i klienty ... 457
Klient i serwer UDP ... 457
Implementacja prostego serwera TCP ... 459
Serwery TCP do obsługi więcej niż jednego klienta ... 460
GServer ... 462
Serwer czatów oparty na GServer ... 465
Serwery WWW i HTTP ... 468
Demony ... 468
Podsumowanie ... 470

Rozdział 16. Przydatne biblioteki i pakiety gem języka Ruby ... 473
abbrev ... 475
Instalacja ... 475
Przykłady ... 475
Dodatkowe informacje ... 476
base64 ... 477
Instalacja ... 477
Przykłady ... 477
Dodatkowe informacje ... 479
BlueCloth ... 480
Instalacja ... 480
Przykłady ... 480
Dodatkowe informacje ... 481
cgi ... 482
Instalacja ... 482
Przykłady ... 482
Dodatkowe informacje ... 487
chronic ... 488
Instalacja ... 488
Przykłady ... 488
Dodatkowe informacje ... 489
Digest ... 490
Instalacja ... 490
Przykłady ... 490
Dodatkowe informacje ... 492
English ... 493
Instalacja ... 493
Przykłady ... 493
Dodatkowe informacje ... 494
ERB ... 495
Instalacja ... 495
Przykłady ... 495
Dodatkowe informacje ... 497
FasterCSV ... 498
Instalacja ... 498
Przykłady ... 498
Dodatkowe informacje ... 502
iconv ... 503
Instalacja ... 503
Przykłady ... 503
Dodatkowe informacje ... 504
logger ... 505
Instalacja ... 505
Przykłady ... 505
Dodatkowe informacje ... 507

pp ... 508
Instalacja ... 508
Przykłady ... 508
Dodatkowe informacje ... 509
RedCloth ... 510
Instalacja ... 510
Przykłady ... 510
Dodatkowe informacje ... 511
StringScanner ... 512
Instalacja ... 512
Przykłady ... 512
Dodatkowe informacje ... 514
tempfile ... 515
Instalacja ... 515
Przykłady ... 515
Dodatkowe informacje ... 517
uri ... 518
Instalacja ... 518
Przykłady ... 518
Dodatkowe informacje ... 521
zlib ... 522
Instalacja ... 522
Przykłady ... 522
Dodatkowe informacje ... 523

Dodatki ... 525

Dodatek A Podsumowanie mechanizmów języka Ruby dla programistów ... 527
Podstawy ... 528
Definicje i pojęcia ... 528
Interpreter języka Ruby i wykonywanie kodu Ruby ... 529
Interactive Ruby ... 530
Wyrażenia, logika i przepływ sterowania ... 531
Podstawowe wyrażenia ... 531
Niezgodności klas ... 532
Wyrażenia porównawcze ... 533
Przepływ sterowania ... 534
Orientacja obiektowa ... 538
Obiekty ... 538
Klasy i metody ... 539
Refleksja ... 541
Ponowne otwieranie klas ... 542
Widoczność metod ... 543
Dane ... 544
Ciągi znaków ... 544
Wyrażenia regularne ... 545
Liczby ... 546
Tablice ... 548
Tablice asocjacyjne ... 549
Struktury złożone ... 549
Wejście-wyjście ... 550
Pliki ... 550
Bazy danych ... 551
Sieć WWW ... 551

Biblioteki ... 552
Organizacja plików ... 552
Tworzenie pakietów ... 553

Dodatek B Przegląd języka Ruby ... 555
Przydatne klasy i metody ... 555
Array ... 555
Bignum i Fixnum ... 557
Enumerable ... 558
Float ... 559
Hash ... 559
Integer ... 560
Numeric ... 560
Object ... 561
String ... 562
Składnia wyrażeń regularnych ... 564
Opcje wyrażeń regularnych ... 564
Specjalne znaki i formacje ... 565
Sufiksy znaków i podwyrażeń ... 565
Klasy wyjątków ... 566
Zmienne specjalne ... 568
Licencja języka Ruby ... 570

Dodatek C Przydatne źródła informacji ... 573
Informacje ogólne ... 573
Ruby ... 573
Ruby on Rails ... 574
Blogi ... 575
Blogi społecznościowe i spisy blogów ... 575
Blogi prywatne ... 576
Fora i grupy dyskusyjne ... 577
Listy dystrybucyjne ... 577
Czaty w czasie rzeczywistym ... 578
Samouczki i przewodniki ... 579
Instalacja ... 579
Ruby i sposoby korzystania z niego ... 580
Ruby on Rails ... 581
Inne ... 582

Skorowidz ... 583

Rozdział 4.

Prosta aplikacja w języku Ruby

Jak dotąd skupialiśmy się na podstawach języka Ruby i przedstawianiu jego mechanizmów na najbardziej podstawowym poziomie. W tym rozdziale przejdziemy już do praktycznych zagadnień związanych z implementacją aplikacji i opracujemy pełną, choć prostą aplikację w języku Ruby, która będzie wykorzystywać proste funkcje. Po zaimplementowaniu i przetestowaniu prostej aplikacji przedstawione zostaną sposoby jej rozbudowy, tak by stała się bardziej użyteczna. W trakcie tworzenia aplikacji omówione zostaną nowe aspekty programowania, o których nie wspominano jeszcze we wcześniejszych rozdziałach.

Najpierw zostaną opisane podstawowe zagadnienia dotyczące organizacji kodu źródłowego. Dopiero potem rozpoczniemy programowanie z prawdziwego zdarzenia.

Praca z plikami zawierającymi kod źródłowy

Dotychczas do nauki języka używaliśmy interaktywnego interpretera języka Ruby — irb. Jednak aby zaimplementować jakikolwiek program, którego będzie można użyć również w późniejszym czasie, kod źródłowy programu trzeba zapisać w pliku przechowywanym na dysku (albo przesyłanym w internecie, przechowywanym na płycie CD itp.).

Narzędzia do tworzenia i manipulowania plikami z kodem źródłowym zależą od używanego systemu operacyjnego oraz osobistych preferencji programisty. W systemie Windows do tworzenia i edycji plików tekstowych wystarczy znany większości użytkowników Notatnik. Z kolei w wierszu poleceń Linuksa można skorzystać z edytora vi, Emacs albo pico/nano. Użytkownicy komputerów Mac mają do dyspozycji program TextEdit. Bez względu na to, które narzędzie zostanie ostatecznie użyte, musi ono umożliwiać tworzenie nowych plików i zapisywanie ich jako zwykłych plików tekstowych, aby

interpreter języka Ruby mógł odpowiednio z nich korzystać. W następnych kilku punktach przedstawione zostaną wybrane narzędzia dostępne dla każdej z wymienionych platform, które nadają się do tworzenia programów w języku Ruby.

Tworzenie pliku testowego

Pierwszym krokiem na drodze do utworzenia aplikacji w języku Ruby jest zaznajomienie się z edytorem tekstu. Oto kilka wskazówek dotyczących najważniejszych platform systemowych.

Czytelnicy, którzy potrafią już korzystać z edytorów tekstu i wiedzą, jak pisze się i zapisuje kod źródłowy, mogą od razu przejść do lektury punktu pod tytułem „Testowy plik z kodem źródłowym".

Windows

Jeżeli wykonano przedstawione w rozdziale 1. instrukcje dotyczące pobrania i instalacji języka Ruby, w grupie programów Ruby w menu *Start* powinny być dostępne dwa edytory tekstu o nazwach SciTE oraz FreeRIDE. SciTE to ogólne narzędzie do edycji kodu źródłowego, natomiast FreeRIDE to edytor kodu źródłowego w języku Ruby, zresztą również zaimplementowany w Ruby. SciTE działa nieco szybciej, natomiast FreeRIDE jest wystarczająco szybki, aby wykonywać w nim ogólne zadania programistyczne, a poza tym jest lepiej zintegrowany z interpreterem języka Ruby.

Po uruchomieniu edytora otwierany jest pusty dokument, w którym można zacząć wpisywać kod źródłowy Ruby (we FreeRIDE nowy dokument tworzy się przez wybranie odpowiedniego polecenia z menu *File*). W menu *File* znajdują się również polecenia do zapisania kodu źródłowego na dysku, co uczynimy w następnym punkcie. FreeRIDE pozwala ponadto zgrupować pojedyncze pliki z kodem w ramach jednego **projektu**.

Mac OS X

System Mac OS X udostępnia kilka edytorów tekstu. Najczęściej używanym edytorem wśród programistów Ruby jest TextMate firmy MacroMates (*http://www.macromates.com/*), widoczny na rysunku 4.1. Edytor ten nie jest jednak darmowy — jego koszt wynosi w przybliżeniu 50 dolarów. Ciekawą alternatywą jest edytor Xcode dołączany do pakietu OS X Development Tools, lecz wymaga on umiejętności instalowania i używania narzędzi programistycznych (dostępnych na dysku instalacyjnym OS X). W zależności od tego, jaki komputer Mac jest używany, Xcode może wydawać się stosunkowo wolnym narzędziem.

Darmowym edytorem dostępnym w OS X jest TextEdit. Aby go uruchomić, należy otworzyć folder *Applications* i dwukrotnie kliknąć ikonę TextEdit. W domyślnym trybie pracy TextEdit nie jest edytorem zwykłego tekstu, natomiast gdy w menu *Format* wybierze się polecenie *Make Plain Text*, edytor zostanie przełączony w tryb odpowiedni do edytowania kodu źródłowego języka Ruby.

Rysunek 4.1.
Edytor TextMate

Na razie wystarczy wpisać albo wkleić kawałek kodu Ruby i wybrać polecenie *File/Save*, aby zapisać kod w wybranej lokalizacji na dysku. Dobrym rozwiązaniem będzie zapewne utworzenie we własnym folderze domowym (czyli w folderze po lewej stronie, którego nazwą jest nazwa użytkownika) folderu o nazwie *ruby* i zapisanie w nim właśnie pliku z kodem źródłowym. W dalszych instrukcjach będziemy bowiem przyjmować założenie, że w folderze domowym użytkownika istnieje właśnie folder o nazwie *ruby*.

Linux

Dystrybucje systemu Linux zwykle zawierają co najmniej kilka różnych edytorów tekstu; zawsze natomiast w każdej dystrybucji znajduje się co najmniej jeden edytor. Użytkownicy korzystający wyłącznie z wiersza poleceń albo z okna terminala będą zapewne umieli korzystać z edytora vi, Emacs, pico albo nano — wszystkie te narzędzia również nadają się do tworzenia kodu źródłowego Ruby. Jeżeli używany jest Linux z graficznym interfejsem użytkownika, zapewne dostępny jest edytor Kate (KDE Advanced Text Editor) i (lub) gedit (GNOME Editor). Wszystkie wspomniane programy są doskonałymi edytorami kodów źródłowych.

Można także pobrać i zainstalować FreeRIDE, czyli darmowy, wieloplatformowy edytor kodu źródłowego stworzony z myślą o programistach Ruby. Edytor pozwala wykonywać kod źródłowy jednym kliknięciem myszy bezpośrednio z poziomu narzędzia (jeżeli używany jest graficzny interfejs użytkownika X). Kod wpisywany w edytorze jest ponadto kolorowany zgodnie ze składnią języka, dzięki czemu łatwiej się go czyta. Więcej informacji na temat FreeRIDE znajduje się na stronie pod adresem *http://freeride.rubyforge.org/*.

Na razie najlepszym pomysłem będzie utworzenie we własnym folderze domowym nowego folderu o nazwie *ruby*, w którym później zapisywane będą pliki z kodem języka Ruby.

Testowy plik z kodem źródłowym

Po wybraniu środowiska, w którym będą edytowane i zapisywane pliki tekstowe, można wpisać następujący kod:

```
x = 2
print "Aplikacja działa poprawnie, jeśli 2 + 2 = #{x + x}"
```

Uwaga

Jeżeli przedstawiony kod nie jest zrozumiały, może to oznaczać, że pominięto zbyt wiele punktów z poprzednich rozdziałów. Należy zatem powrócić do lektury rozdziału 3. W niniejszym rozdziale potrzebna jest znajomość wszystkich zagadnień prezentowanych w rozdziale 3.

Kod źródłowy należy zapisać w pliku o nazwie *a.rb*, w wybranym folderze. Warto w tym celu utworzyć folder o nazwie *ruby* w łatwo dostępnej lokalizacji. W systemie Windows folder *ruby* może znajdować się w katalogu głównym na dysku *C*, natomiast w systemach OS X i Linux można go umieścić we własnym katalogu domowym.

Uwaga

RB to standardowe rozszerzenie plików z kodem języka Ruby, tak samo jak *PHP* jest rozszerzeniem plików z kodem PHP, *TXT* dotyczy plików ze zwykłym tekstem, a *JPG* jest standardowym rozszerzeniem obrazków w formacie JPEG.

Czas więc uruchomić kod źródłowy.

Wykonywanie kodu źródłowego

Gdy utworzony już zostanie plik z kodem źródłowym języka Ruby o nazwie *a.rb*, trzeba ten kod wykonać. Jak zwykle sposób uruchamiania kodu zależy od używanego systemu operacyjnego, dlatego najlepiej jest przeczytać treść któregoś z kolejnych punktów, poświęconych konkretnym systemom. Jeżeli używany system nie jest opisywany w żadnym z punktów, wówczas najprawdopodobniej trzeba będzie wykonać czynności identyczne z tymi, które dotyczą systemu OS X albo Linux.

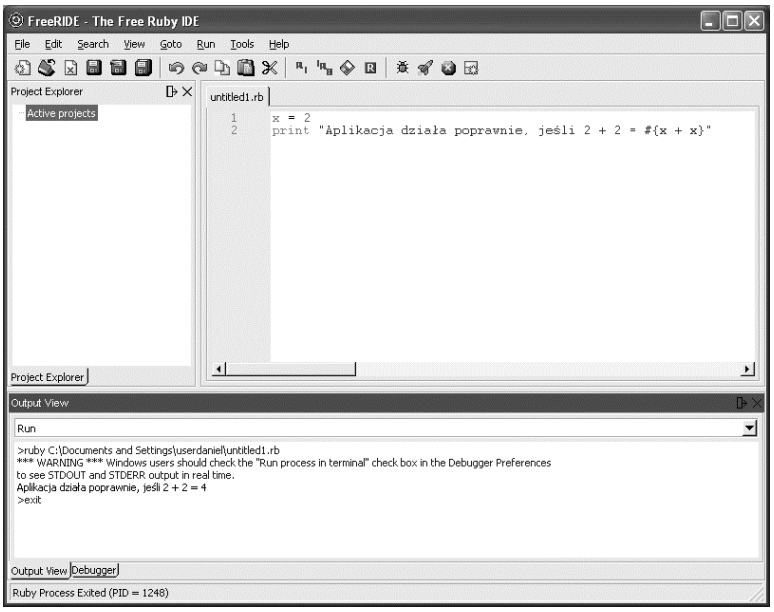
Zawsze, gdy w książce będzie mowa o „wykonaniu” albo „uruchomieniu” programu, chodzić będzie o wykonanie czynności opisywanych w punkcie poświęconym określonemu systemowi operacyjnemu.

Uwaga

W tym rozdziale utworzona zostanie pełna aplikacja, jednak i tak nadal w niektórych momentach warto będzie korzystać z irb, aby sprawdzić działanie niektórych instrukcji albo przetestować opisywane rozwiązania. Najlepiej samodzielnie i na bieżąco decydować, którego z tych dwóch narzędzi (irb i edytor kodu) użyć. W przypadku krótkich rozwiązań i niewielkich bloków kodu bardziej użyteczne będzie zapewne irb, ponieważ nie będzie trzeba wówczas poświęcać dodatkowego czasu na przełączanie się między edytorem kodu a interpreterem.

Windows

Jeżeli używany jest program SciTE albo FreeRIDE zainstalowany przez instalator Ruby dla systemu Windows, wówczas bezpośrednio z ich poziomu można uruchamiać programy napisane w języku Ruby (rysunek 4.2). W obu programach do uruchamiania kodu źródłowego służy klawisz *F5*. Podobnym rozwiązaniem jest wybranie polecenia z menu (*Tools/Go* w SciTE albo *Run/Run* w FreeRIDE). Jednak przed wykonaniem kodu trzeba najpierw zapisać plik, w którym ten kod się znajduje. Jeżeli plik nie zostanie zapisany, kod może dać nieprzewidziane wyniki (na przykład może dojść do wykonania ostatnio zapisanej wersji kodu) albo wyświetlony zostanie komunikat z prośbą o zapisanie pliku.



Rysunek 4.2. *Efekt wykonania kodu w narzędziu FreeRIDE w systemie Windows (wynik działania kodu znajduje się w dolnym oknie edytora)*

Programy Ruby można także uruchamiać w wierszu poleceń. W tym celu należy przejść do wiersza poleceń (w menu *Start* wybrać polecenie *Uruchom* i wpisać `cmd`, a następnie kliknąć przycisk *OK*), za pomocą polecenia `cd` przejść do folderu, w którym znajduje się plik *a.rb*, i wpisać polecenie `ruby a.rb`.

Metoda z wierszem poleceń jest jednak przeznaczona dla osób, które potrafią poruszać się po dysku przy użyciu poleceń systemu operacyjnego. Inną opcją, tym razem dla użytkowników potrafiących definiować skróty, jest utworzenie skrótu prowadzącego do pliku wykonywalnego Ruby (*ruby.exe*) i przeciąganie na ten skrót pliku zawierającego kod źródłowy.

Mac OS X

W systemie Mac OS X najprostszą metodą uruchamiania aplikacji napisanych w Ruby jest wykorzystanie okna *Terminal*, tak samo jak do uruchamiania interpretera irb. Działanie okna *Terminal* opisano w rozdziale 1. Jeżeli wykonane zostały przedstawione tam instrukcje, należy podjąć następujące dodatkowe działania:

1. Uruchomić *Terminal* (znajduje się on w *Applications/Utilities*).
2. Za pomocą polecenia `cd` przejść do folderu, w którym zapisano plik *a.rb*, na przykład `cd ~/ruby`. Polecenie o tej treści spowoduje, że *Terminal* przejdzie do folderu *ruby* znajdującego się w domowym folderze użytkownika.
3. Wpisać polecenie `ruby a.rb` i nacisnąć klawisz *Enter*, aby wykonać skrypt Ruby.
4. Jeżeli zwrócony zostanie błąd o treści `ruby: No such file or directory - a.rb (LoadError)`, oznacza to, że bieżącym folderem jest folder inny niż ten, w którym zapisano plik *a.rb*, i konieczne jest ponowne ustalenie, gdzie plik z kodem źródłowym się znajduje.

Jeżeli wykonanie pliku *a.rb* da prawidłowy wynik, będzie można przejść od razu do punktu „Pierwsza aplikacja: Analizator tekstu”.

Linux i inne systemy uniksowe

W Linuksie oraz systemach z rodziny Unix aplikacje napisane w języku Ruby uruchamia się z poziomu powłoki (czyli w oknie terminala) w taki sam sposób, w jaki uruchamia się irb. Sposób uruchamiania irb opisano w rozdziale 1., dlatego przedstawione tam procedury trzeba wykonać ponownie, a następnie podjąć działania zaprezentowane poniżej:

1. Uruchomić emulator terminala (na przykład *xterm*), aby przejść do wiersza poleceń lub powłoki Linuksa.
2. Za pomocą polecenia `cd` przejść do katalogu, w którym znajduje się plik *a.rb* (na przykład polecenie `cd ~/ruby` spowoduje przejście do katalogu *ruby* znajdującego się bezpośrednio w katalogu domowym, którym zazwyczaj jest */home/nazwaużytkownika/*).
3. Wpisać polecenie `ruby a.rb` i nacisnąć *Enter*, aby wykonać skrypt *a.rb*.

Jeżeli na skutek wykonania skryptu *a.rb* zwrócone zostaną oczekiwane wyniki, można przejść do lektury kolejnych punktów.

Edytory tekstu a edytory kodu źródłowego

Wcześniej powiedziano, że zasadniczo kod źródłowy jest zwykłym tekstem. To oczywiście prawda i kod źródłowy można pisać w zwykłym edytorze tekstów, jednak wykorzystanie do tego specjalistycznego edytora kodu źródłowego (albo zintegrowanego środowiska programistycznego IDE) z pewnością przyniesie dodatkowe korzyści.

Edytor FreeRIDE jest przykładem edytora stworzonego konkretnie z myślą o programistach Ruby. Narzędzie pozwala edytować tekst, podobnie jak każdy inny edytor tekstu, lecz oprócz tego udostępnia także dodatkowe funkcje, takie jak kolorowanie kodu źródłowego, a także umożliwia wykonywanie kodu bezpośrednio z poziomu edytora.

Według niektórych programistów kolorowanie składni kodu źródłowego jest funkcją nie do przecenienia, ponieważ dzięki temu kod jest bardziej czytelny. Nazwy zmiennych, wyrażenia, literały ciągów znaków oraz inne elementy kodu źródłowego są prezentowane w odmiennych kolorach, dzięki czemu łatwiej je wyłowić z tekstu.

Wybór między edytorem kodu źródłowego a zwykłym edytorem tekstów jest wyłącznie decyzją programisty, najlepiej jednak wypróbować narzędzia reprezentujące obydwie grupy. Wielu programistów ceni sobie wolność korzystania ze zwykłego edytora tekstów i uruchamiania tak zaimplementowanego kodu w wierszu poleceń; inni z kolei są zdeklarowanymi użytkownikami zintegrowanego środowiska programistycznego.

FreeRIDE jest dostępny w witrynie pod adresem *http://freeride.rubyforge.org/*, zaś konkurencyjne wobec niego narzędzie do edycji kodu źródłowego Ruby **i** Rails noszące nazwę RadRails jest dostępne w witrynie pod adresem *http://www.radrails.org/*. Warto jest wypróbować w używanym systemie operacyjnym obydwa wymienione narzędzia, ponieważ mogą one lepiej spełniać oczekiwania i ułatwiać pracę programistyczną.

Pierwsza aplikacja: analizator tekstu

Celem niniejszego rozdziału jest zaimplementowanie aplikacji będącej analizatorem tekstu. Kod języka Ruby będzie wczytywał tekst znajdujący się w pliku zewnętrznym, analizował tekst pod kątem statystycznym oraz występowania różnych wzorców, a następnie wyświetlał wyniki analizy użytkownikowi. Nie jest to wprawdzie żadna aplikacja z trójwymiarową grafiką ani efektowna witryna internetowa, lecz programy do przetwarzania tekstu to podstawowe narzędzia wykorzystywane w celu administrowania systemami i tworzenia aplikacji. Tego rodzaju aplikacje są nieocenione, gdy zachodzi konieczność parsowania plików dziennika zdarzeń oraz tekstu wpisywanego przez użytkownika w witrynach internetowych, a także w przypadku wykonywania operacji na innych danych tekstowych.

Dzięki obecności funkcji do obsługi wyrażeń regularnych język Ruby doskonale nadaje się do analizy tekstu i dokumentów; równie przydatne są metody `scan` i `split`. Zarówno wspomniane metody, jak i funkcje do obsługi wyrażeń regularnych będą często wykorzystywane w dalszej części książki.

Uwaga

> W trakcie tworzenia aplikacji skupimy się na jak najszybszym zaimplementowaniu potrzebnych funkcji kosztem projektowania precyzyjnej struktury obiektowej, tworzenia dokumentacji czy testowania aplikacji zgodnie z przyjętymi metodologiami. Reguły orientacji obiektowej zostaną szczegółowo omówione w rozdziale 6., zaś zagadnienia dotyczące dokumentowania i testowania kodu będą przedmiotem rozdziału 8.

Podstawowe funkcje aplikacji

Analizator tekstu będzie generował statystyki, w których będą się znajdować następujące dane:

- Liczba znaków.
- Liczba znaków bez znaków spacji.
- Liczba wierszy.
- Liczba słów.
- Liczba zdań.
- Liczba akapitów.
- Średnia liczba słów w zdaniu.
- Średnia liczba zdań w akapicie.

Ostatnie dwie wartości łatwo jest obliczyć na podstawie danych szczegółowych. Jeżeli znana jest liczba wszystkich słów oraz liczba wszystkich zdań, to wystarczy wykonać prostą operację dzielenia, aby wyznaczyć średnią liczbę słów w jednym zdaniu.

Implementacja aplikacji

Przed przystąpieniem do implementowania nowego programu warto jest najpierw zastanowić się nad najważniejszymi czynnościami, które program będzie wykonywał. W przeszłości często rysowano tak zwane **diagramy przepływu** (ang. *flow charts*), ilustrujące kolejność operacji wykonywanych przez komputer. Obecnie jednak dostępne nowoczesne narzędzia, takie jak Ruby, pozwalają na eksperymentowanie i modyfikowanie aplikacji „na żywo". Aplikacja będzie musiała wykonywać następujące czynności:

1. Załadować plik lub dokument zawierający tekst, który ma podlegać analizie.
2. W trakcie ładowania tekstu z pliku zliczać kolejne wiersze (jest to jedna ze zwracanych statystyk).
3. Umieszczać tekst w ciągu znaków i zmierzyć jego długość, aby uzyskać liczbę znaków.
4. Tymczasowo usunąć wszystkie znaki niewidoczne i ponownie zmierzyć długość tekstu, aby otrzymać liczbę znaków bez znaków spacji.
5. Podzielić tekst względem znaków niewidocznych, aby uzyskać liczbę słów.
6. Podzielić tekst względem znaków kropki, aby uzyskać liczbę zdań.
7. Podzielić tekst względem występujących jeden po drugim znaków nowego wiersza, aby uzyskać liczbę akapitów.
8. Wykonać obliczenia, aby uzyskać wartości średnie.

Należy utworzyć nowy, pusty plik kodu źródłowego Ruby i zapisać go w folderze *ruby* pod nazwą *analyzer.rb*. Plik będzie wypełniany kolejnymi fragmentami kodu źródłowego w następnych punktach.

Uzyskanie przykładowego tekstu

Przed rozpoczęciem kodowania trzeba najpierw przygotować dane testowe, które będą przetwarzane przez analizator. Doskonałym przykładem może być fragment książki *Oliver Twist* w wersji oryginalnej, której treść jest dostępna za darmo i łatwa do znalezienia. Tekst powieści znajduje się pod adresem *http://www.rubyinside.com/book/oliver.txt* oraz *http://www.dickens-literature.com/Oliver_Twist/0.html* i można jego fragment skopiować do pliku tekstowego. Plik najlepiej jest zapisać w tym samym folderze, w którym znajduje się już *analyzer.rb*, pod nazwą *text.txt*. Domyślnie nasza aplikacja będzie przetwarzać właśnie zawartość pliku o nazwie *text.txt* (choć później zostanie rozszerzona o możliwość wskazywania jej również innych źródeł danych).

Jeżeli strony pod podanymi adresami są w danym momencie niedostępne, można wpisać w wyszukiwarce tekst „twist workhouse rendered profound thingummy" — powinno to ułatwić znalezienie innego źródła tekstu powieści. Alternatywnym rozwiązaniem jest wykorzystanie innego dostępnego fragmentu tekstu.

Jeżeli jako przykład używany jest fragment powieści *Oliver Twist*, to aby uzyskać wyniki przynajmniej zbliżone do prezentowanych w kolejnych przykładach z tego rozdziału, należy skopiować jedynie fragment tekstu z początku powieści, zawarty między następującymi zdaniami:

```
Among other public buildings in a certain town, which for many reasons it will be prudent
↪to refrain from mentioning
```

Oraz:

```
Oliver cried lustily. If he could have known that he was an orphan, left to the tender
↪mercies of church-wardens and overseers, perhaps he would have cried the louder.
```

Ładowanie plików tekstowych i zliczanie wierszy

Czas zacząć kodować! Pierwsza czynność polega na załadowaniu pliku. Ruby udostępnia bogaty zbiór metod do operowania na plikach, które należą do klasy `File`. W niektórych innych językach programowania przetwarzanie pliku wymaga wywoływania metod różnych klas, natomiast w języku Ruby cały interfejs jest bardzo prosty. Oto polecenie, które otwiera plik *text.txt*:

```
File.open("text.txt").each { |line| puts line }
```

Polecenie można zapisać w pliku *analyzer.rb* i je wykonać. Jeżeli *text.txt* znajduje się w bieżącym katalogu, w wyniku wykonania polecenia na ekranie pojawi się pełen tekst zawarty w pliku.

Przedstawione polecenie żąda od klasy `File` otwarcia pliku o nazwie *text.txt*, a następnie — podobnie jak w przypadku tablicy — bezpośrednio na tym pliku można wywołać metodę `each`, dzięki czemu każdy wiersz tekstu zostanie przekazany do wewnętrznego bloku kodu. W bloku kodu natomiast znajduje się metoda `puts`, która wyświetli wiersz na ekranie. (W rozdziale 9. mechanizmy otwierania i manipulowania plikami zostaną przedstawione bardziej szczegółowo; omówione zostaną także rozwiązania bardziej efektywne niż stosowane w tym rozdziale!).

Treść kodu należy zmienić na następującą:

```
line_count = 0
File.open("text.txt").each { |line| line_count += 1 }
puts line_count
```

Najpierw inicjowana jest zmienna `line_count`, która będzie przechowywać liczbę wierszy tekstu. Następnie plik zostaje otwarty i wykonywana jest iteracja przez kolejne jego wiersze; jednocześnie w każdej iteracji wartość zmiennej `line_count` jest zwiększana o `1`. Po zakończeniu iteracji uzyskana liczba jest wyświetlana na ekranie (w przypadku tekstu z powieści *Oliver Twist* powinna zostać zwrócona liczba zbliżona do 121). W ten sposób uzyskano pierwszą statystykę!

Wiersze zostały zatem policzone, wciąż jednak nie mamy dostępu do zawartości pliku, aby policzyć występujące w nim słowa, akapity, zdania i tak dalej. Nie jest to jednak trudne zadanie. Wystarczy zmienić nieco dotychczasowy kod i dodać w nim zmienną `text`, do której dopisywane będą kolejne odczytywane wiersze:

```
text=''
line_count = 0
File.open("text.txt").each do |line|
  line_count += 1
  text << line
end

puts "#{line_count} wierszy"
```

Uwaga

Należy pamiętać, że nawiasy klamrowe { i } otaczające blok kodu to standardowy styl wyznaczania bloków zawierających tylko jeden wiersz kodu, natomiast w przypadku bloków wielowierszowych lepiej jest używać słów `do` i `end`. Jest to jednak tylko konwencja, a nie wymóg.

W odróżnieniu od poprzedniej wersji kodu nowa wersja zawiera zmienną `text` i dodaje do niej kolejne wiersze odczytywane z pliku. Gdy iteracja przez zawartość pliku dobiegnie końca — to znaczy gdy wszystkie wiersze zostaną już wczytane — `text` będzie zawierać ciąg znaków z pełną zawartością tekstu z pliku, gotowy do przetwarzania.

Jest to najprostszy sposób wczytywania zawartości pliku do pojedynczego ciągu znaków i zliczania wierszy. Klasa File udostępnia jednak również inne metody, za pomocą których tekst z pliku można wczytać szybciej. Przykładowy kod można przepisać do następującej postaci:

```
lines = File.readlines("text.txt")
line_count = lines.size
text = lines.join

puts "#{line_count} wierszy"
```

Ten sposób jest o wiele prostszy! Klasa `File` implementuje metodę `readlines`, która wczytuje całą zawartość do tablicy wiersz po wierszu. W ten sposób można jednocześnie policzyć wiersze i połączyć je w jeden ciąg znaków.

Zliczanie znaków

Drugą statystyką, którą najłatwiej będzie policzyć, jest liczba znaków znajdujących się w pliku. Dzięki temu, że cała zawartość pliku znajduje się w zmiennej `text` i zmienna ta jest ciągiem znaków, wystarczy użyć udostępnianej przez każdy ciąg znaków metody `length`, aby uzyskać rozmiar pliku, a co za tym idzie — liczbę znaków.

Na końcu kodu z poprzedniego przykładu w pliku *analyzer.rb* trzeba dopisać następujące dwa wiersze:

```
total_characters = text.length
puts "#{total_characters} znaków"
```

Wykonanie kodu z pliku *analyzer.rb* na tekście *Oliver Twist* spowoduje zwrócenie następujących wyników:

```
121 wierszy
6165 znaków
```

Drugą wymaganą statystyką, która wiąże się z liczbą znaków, jest łączna liczba znaków w tekście z wyłączeniem znaków spacji. Jak pamiętamy z rozdziału 3., ciągi znaków udostępniają metodę `gsub`, która wykonuje globalne zastąpienie (podobnie jak operacja „szukaj i zastąp") w ciągu znaków, na przykład:

```
"to jest test".gsub(/t/, 'X')
```

```
Xo jesX XesX
```

W taki sam sposób za pomocą metody `gsub` można usunąć z ciągu znaków `text` wszystkie znaki spacji, a następnie ponownie przy użyciu `length` odczytać długość zmiennej `text` w nowej postaci. W pliku *analyzer.rb* należy dopisać następujące wiersze kodu:

```
total_characters_nospaces = text.gsub(/\s+/, '').length
puts "#{total_characters_nospaces} znaków nie licząc spacji"
```

Wykonanie kodu z pliku *analyzer.rb* na fragmencie powieści *Oliver Twist* spowoduje zwrócenie następujących wyników:

```
121 wierszy
6165 znaków
5055 znaków nie licząc spacji
```

Zliczanie słów

Funkcją standardowo udostępnianą przez programy do przetwarzania tekstu jest „licznik słów". Zadaniem funkcji jest wskazywanie liczby pełnych słów znajdujących się w całym dokumencie lub w zaznaczonym fragmencie tekstu. Na podstawie tej informacji można oszacować, ile stron zajmie dany tekst po wydrukowaniu. Często zdarza się również, że trzeba napisać dokument o określonej liczbie słów — wówczas funkcja zliczania słów jest wręcz nieoceniona.

Funkcję można zaimplementować co najmniej na dwa różne sposoby, poprzez:

1. odczytanie liczby ciągłych grup znaków za pomocą metody `scan`,
2. podzielenie tekstu na części względem znaków spacji i odczytanie liczby uzyskanych w ten sposób fragmentów przy użyciu metod `split` i `size`.

Przeanalizujemy obydwa podejścia i wybierzemy lepsze z nich. Jak pamiętamy z rozdziału 3., metoda `scan` działa w ten sposób, że iteruje przez ciąg znaków z tekstem i odnajduje wskazane wzorce, na przykład:

```
puts "to jest test".scan(/\w/).join
```

```
tojesttest
```

W przykładzie metoda `scan` wyszukuje w ciągu znaków sekwencje pasujące do wzorca `\w`, czyli specjalnego wzorca reprezentującego wszystkie znaki alfanumeryczne (wraz ze znakiem podkreślenia), po czym umieszcza je w tablicy. Elementy tablicy zostają złączone ze sobą do postaci ciągu znaków, który na końcu jest wyświetlany na ekranie.

Tak samo można postąpić z grupami znaków alfanumerycznych. W rozdziale 3. wspomniano, że aby do wyrażenia regularnego dopasować większą liczbę znaków, należy użyć znaku +. Spróbujmy zatem:

```
puts "to jest test".scan(/\w+/).join('-')
```

```
to-jest-test
```

Tym razem metoda wyszukała wszystkie *grupy* znaków alfanumerycznych i umieściła je w tablicy, której elementy zostały następnie połączone w jeden ciąg znaków z wykorzystaniem znaku - jako separatora.

Aby odczytać liczbę słów znajdujących się w ciągu znaków, można użyć metod tablicowych `length` albo `size`. Obydwie metody zwracają liczbę elementów, zamiast je ze sobą łączyć:

```
puts "to jest test".scan(/\w+/).length
```

```
3
```

Doskonale! A w jaki sposób można wykorzystać metodę `split` do podzielenia tekstu na części?

Zastosowanie metody `split` ilustruje główną zasadę języka Ruby (a także niektórych innych języków, w szczególności Perla), według której „każdą operację można wykonać na więcej niż jeden sposób!”. Analiza różnych rozwiązań danego problemu jest warunkiem zostania dobrym programistą, ponieważ każde z możliwych rozwiązań może charakteryzować się odmienną wydajnością.

Ciąg znaków należy podzielić względem znaków spacji, a następnie sprawdzić długość tak uzyskanej tablicy:

```
puts "to jest test".split.length
```

```
3
```

Domyślnie metoda `split` dzieli ciąg znaków względem znaków niewidocznych (pojedynczego lub więcej niż jeden następujących po sobie znaków spacji, tabulacji, nowego wiersza i tak dalej). Dzięki temu kod źródłowy jest krótszy i bardziej czytelny niż rozwiązanie wykorzystujące metodę `scan`.

Czym się zatem różnią obydwa rozwiązania? Krótko mówiąc, pierwsze z nich polega na wyszukaniu słów i zwróceniu ich w celu sprawdzenia ich liczby, zaś drugie sprowadza się do podzielenia ciągu znaków na podstawie znaków oddzielających od siebie kolejne słowa — czyli znaków niewidocznych — i sprawdzeniu, na ile części ciąg rzeczywiście został podzielony. Co ciekawe, każde z tych rozwiązań może zwrócić odmienne wyniki:

```
text = "Nic nie jest czarno-białe."
puts "Metoda scan: #{text.scan(/\w+/).length}"
puts "Metoda split: #{text.split.length}"
```

```
Metoda scan: 6
Metoda split: 4
```

To ciekawe! Metoda `scan` iteruje przez wszystkie bloki znaków alfanumerycznych, których oczywiście jest sześć w przykładowym zdaniu. Jeżeli natomiast ciąg znaków zostanie podzielony na podstawie znaków spacji, okaże się, że słowa są tylko cztery. Różnica wynika z obecności w ciągu znaków słów ze znakiem myślnika. Myślnik nie jest znakiem „alfanumerycznym", dlatego `scan` traktuje „czarno" i „białe" jako dwa oddzielne słowa.

W pliku *analyzer.rb* warto wykorzystać nowe informacje. Należy zatem dopisać następujące dwa wiersze kodu:

```
word_count = text.split.length
puts "#{word_count} słów"
```

Wykonanie kodu znajdującego się w pliku *analyzer.rb* spowoduje wyświetlenie następujących wyników:

```
121 wierszy
6165 znaków
5055 znaków nie licząc spacji
1093 słów
```

Zliczanie zdań i akapitów

Dzięki temu, że wiemy już, w jaki sposób należy zliczać słowa, zliczenie zdań i akapitów jawi się już jako proste zadanie. W przypadku analizy zdań i akapitów trzeba obrać inne kryteria podziału niż dzielenie ciągu znaków względem znaków niewidocznych.

Zdania kończą się znakiem kropki, znakiem zapytania albo wykrzyknikiem. Czasem zdania mogą się też kończyć myślnikiem albo innym znakiem interpunkcyjnym, jednak są to przypadki na tyle rzadkie, że nie będziemy ich brali pod uwagę. Sam podział jest bardzo prosty. Zamiast nakazać interpreterowi Ruby, by podzielił tekst na części względem tylko jednego znaku, trzeba wydać polecenie podziału względem dowolnego z trzech znaków w następujący sposób:

```
sentence_count = text.split(/\.|\?|!/).length
```

Wyrażenie regularne zastosowane w przykładzie może wydać się dziwne, lecz znaki kropki, zapytania i wykrzyknika są w nim wyraźnie widoczne. Przyjrzyjmy się samemu wyrażeniu regularnemu:

```
/\.|\?|!/
```

Znak ukośnika znajdujący się na początku i na końcu jest standardowym separatorem wyrażenia regularnego, zatem można go zignorować. Pierwszym elementem wyrażenia jest `\.` — fragment ten oznacza znak kropki. W wyrażeniu nie można wpisać samego

znaku kropki bez poprzedzającego go ukośnika, ponieważ w wyrażeniach regularnych `.` oznacza „dowolny znak” (zgodnie z informacjami zawartymi w rozdziale 3.). Kropkę należy zatem *zneutralizować* za pomocą znaku ukośnika, aby została ona potraktowana dosłownie jako znak kropki. Z tego samego powodu znak ukośnika poprzedza znak zapytania — znak zapytania oznacza zwykle w wyrażeniach regularnych „żaden lub jeden ze znaków poprzedzających”, o czym również wspomniano w rozdziale 3. Znaku `!` nie trzeba neutralizować, ponieważ nie ma on żadnej dodatkowej interpretacji w wyrażeniach regularnych.

Znaki potoku (czyli znaki `|`) oddzielają od siebie trzy znaki główne, co oznacza, że znaki główne mają być traktowane oddzielnie i metoda `split` ma dopasowywać ciąg znaków do dowolnego z nich. Dzięki temu ciąg znaków można dzielić jednocześnie względem kropki, znaku zapytania *i* wykrzyknika. Aby się o tym przekonać, można wykonać następujący kod:

```
puts "Kod testowy! Działa. Na pewno? Tak.".split(/\.|\?|!/).length
```

```
4
```

Akapity również można wydzielić za pomocą wyrażeń regularnych. W książkach drukowanych, takich jak ta, akapity zwykle nie są oddzielane od siebie dodatkowym pustym wierszem, natomiast akapity tekstu wpisywanego w komputerze są zazwyczaj tak od siebie oddzielone. Zatem aby oddzielić od siebie poszczególne akapity, wystarczy podzielić tekst względem dwóch występujących obok siebie znaków nowego wiersza (reprezentowanych przez specjalną kombinację `\n\n`, oznaczającą dwa sąsiadujące ze sobą znaki nowego wiersza), na przykład:

```
text = %q{
To jest test
akapitu pierwszego.

To jest test
drugiego akapitu.

A to jest test
akapitu numer trzy.
}

puts text.split(/\n\n/).length
```

```
3
```

Obydwa rozwiązania należy zatem dopisać do pliku *analyzer.rb*:

```
paragraph_count = text.split(/\n\n/).length
puts "#{paragraph_count} akapitów"

sentence_count = text.split(/\.|\?|!/).length
puts "#{sentence_count} zdań"
```

Obliczenie wartości średnich

Ostatnimi statystykami, jakie ma liczyć nasza pierwsza aplikacja, są: średnia liczba słów w jednym zdaniu oraz średnia liczba zdań w jednym paragrafie. Zmienne `word_count`, `paragraph_count` i `sentence_count` zawierają już wartości wskazujące odpowiednio liczbę słów, akapitów i zdań w tekście. Pozostaje zatem jedynie wykonać proste obliczenia arytmetyczne w następujący sposób:

```
puts "#{sentence_count / paragraph_count} zdań w jednym akapicie (średnio)"
puts "#{word_count / sentence_count} słów w jednym zdaniu (średnio)"
```

Obliczenia są na tyle proste, że można je umieścić bezpośrednio w poleceniach prezentujących ostateczne wartości, zamiast wykonywać je w oddzielnych instrukcjach.

Kod źródłowy aplikacji

W miarę opracowywania koncepcji działania kolejnych funkcji kod aplikacji był stopniowo rozbudowywany i za każdym razem odpowiednia logika była implementowana jako część instrukcji `puts` wyświetlającej wynik obliczeń użytkownikowi. Jednak w ostatecznej wersji aplikacji lepiej jest zachować porządek i oddzielić logikę od warstwy prezentacji, a obliczenia przenieść do oddzielnego bloku kodu i wykonywać je w całości przed rozpoczęciem wyświetlania wyników.

Sama logika pozostaje bez zmian, natomiast po wprowadzeniu opisanych modyfikacji kod znajdujący się w pliku *analyzer.rb* będzie nieco bardziej uporządkowany:

```
lines = File.readlines("text.txt")
line_count = lines.size
text = lines.join
word_count = text.split.length
character_count = text.length
character_count_nospaces = text.gsub(/\s+/, '').length
paragraph_count = text.split(/\n\n/).length
sentence_count = text.split(/\.|\?|!/).length

puts "#{line_count} wierszy"
puts "#{character_count} znaków"
puts "#{character_count_nospaces} znaków, nie licząc spacji"
puts "#{word_count} słów"
puts "#{paragraph_count} akapitów"
puts "#{sentence_count} zdań"
puts "#{sentence_count / paragraph_count} zdań w jednym akapicie (średnio)"
puts "#{word_count / sentence_count} słów w jednym zdaniu (średnio)"
```

Jeżeli na razie wszystko jest zrozumiałe, jest powód do dumy. W dalszej kolejności zastanowimy się, w jaki sposób można rozszerzyć nieco funkcjonalność aplikacji o dodatkowe interesujące statystyki.

Dodatkowe funkcje aplikacji

Analizator tekstu realizuje kilka podstawowych funkcji, nie robi jednak jakiegoś szczególnego wrażenia. Liczba wierszy, akapitów i słów to z pewnością przydatna informacja, lecz Ruby pozwala na wyciąganie zdecydowanie ciekawszych informacji na temat przetwarzanego tekstu. Jedynym ograniczeniem jest tak naprawdę wyobraźnia programisty. W tym punkcie opisanych zostanie kilka dodatkowych funkcji, a także przedstawiony zostanie sposób ich implementacji.

Uwaga

W trakcie tworzenia aplikacji zawsze warto jednocześnie brać pod uwagę prawdopodobieństwo tego, że program zostanie w przyszłości rozszerzony lub zmieniony i od razu uwzględniać tę możliwość w trakcie implementacji. Zbyt restrykcyjne projektowanie aplikacji powoduje zwykle, że późniejsze ich rozszerzanie jest zadaniem niezwykle trudnym i czasochłonnym.

Procent słów „znaczących”

W większości materiałów pisanych, również w tej książce, znajduje się wiele słów, które nakreślają kontekst i wyznaczają strukturę wypowiedzi, natomiast same w sobie nie niosą żadnego konkretnego znaczenia. Choćby znajdujące się w zdaniu poprzednim słowa „w”, „się” czy „i” nie mają specjalnego znaczenia nawet pomimo tego, że bez nich całe zdanie stałoby się bezsensowne.

Tego typu słowa to tak zwane „słowa pomijane” (ang. *stop words*), które przez systemy komputerowe odpowiedzialne za analizę i przeszukiwanie tekstu są najczęściej pomijane. Nie są to bowiem słowa, które są wyszukiwane przez użytkowników systemu (w odróżnieniu na przykład od rzeczowników). Doskonałym przykładem tej reguły jest wyszukiwarka Google, która nie przechowuje słów mających zerową wartość informacyjną i bez znaczenia dla wykonywanych operacji wyszukiwania.

Uwaga

Więcej informacji na temat słów pomijanych, w tym pełne listy takich słów, można znaleźć na stronach pod adresami *http://pl.wikipedia.org/wiki/Wikipedia:Stopwords* oraz *http://en.wikipedia.org/wiki/Stop_words*.

Można się spodziewać, że tekst bardziej „interesujący” albo napisany przez bardziej kompetentnego autora będzie zawierał niższy odsetek słów pomijanych, a odsetek słów przydatnych albo interesujących będzie wyższy. Łatwo jest rozszerzyć aplikację o funkcję obliczania procentowej ilości słów innych niż słowa pomijane w analizowanym tekście.

Aby zaimplementować taką funkcję, trzeba najpierw utworzyć listę słów pomijanych (ponieważ przykładowy tekst analizowany przez aplikację jest tekstem języka angielskiego, również lista słów pomijanych będzie zawierać słowa angielskie). Słów pomijanych mogą być setki, jednak na początek nasza lista będzie zawierać ich tylko kilka. Słowa będą przechowywane w specjalnie w tym celu utworzonej tablicy:

```
stop_words = %w{the a by on for of are with just but and to the my I has some in}
```

Polecenie spowoduje utworzenie tablicy ze słowami pomijanymi, która jest przypisywana zmiennej `stop_words`.

Wskazówka

> W rozdziale 3. tablice definiowano przy użyciu następującego zapisu: `x = ['a', 'b', 'c']`. Jednak, podobnie jak w wielu innych językach, Ruby pozwala na stosowanie zapisów skrótowych, za pomocą których można szybko tworzyć tablice z tekstem podzielonym na ciągi znaków. Zapis z rozdziału 3. można zastąpić krótszym zapisem `x = %w{a b c}`, który został już zastosowany w przedstawionym przed chwilą kodzie tworzącym listę słów pomijanych.

Dla celów demonstracyjnych można napisać krótki, samodzielny program, aby za jego pomocą przetestować nowe rozwiązanie:

```
text = %q{Los Angeles has some of the nicest weather in the country.}
stop_words = %w{the a by on for of are with just but and to the my I has some}

words = text.scan(/\w+/)
key_words = words.select { |word| !stop_words.include?(word) }

puts key_words.join(' ')
```

Po wykonaniu przykładowego kodu zwrócony zostanie następujący wynik:

```
Los Angeles nicest weather country
```

Ciekawe, prawda? Najpierw do programu wczytywany jest tekst, a następnie lista słów pomijanych. W kolejnym kroku wszystkie słowa zostają wczytane ze zmiennej `text` do tablicy o nazwie `words`, aż w końcu następuje gwóźdź programu:

```
key_words = words.select { |word| !stop_words.include?(word) }
```

W przedstawionym wierszu kodu najpierw na tablicy słów o nazwie `words` wywoływana jest metoda `select`, do której przekazywany jest blok kodu przetwarzający każde słowo (podobnie jak iteratory, z którymi mieliśmy do czynienia w rozdziale 3.). Metoda `select` jest dostępna dla wszystkich tablic oraz tablic asocjacyjnych i zwraca elementy tablicy albo tablicy asocjacyjnej, które pasują do wyrażenia zawartego w bloku kodu.

W tym konkretnym przykładzie każde słowo jest przekazywane do bloku kodu za pośrednictwem zmiennej `word`, po czym kod sprawdza w tablicy `stop_words`, czy znajduje się w niej element identyczny jak `word`. Tak właśnie działa fragment `stop_words.`
`↪include?(word)`.

Znak wykrzyknika (`!`) znajdujący się przed wyrażeniem oznacza jego negację (wykrzyknik jest znakiem negacji dla każdego wyrażenia w języku Ruby). Zastosowano go dlatego, że *niepotrzebne* są nam słowa, które *znajdują się* w tablicy `stop_words` — interesują nas wyłącznie te słowa, których w tej tablicy *nie ma*.

Reasumując, metoda `select` pobiera wszystkie elementy z tablicy `words`, których *nie* ma w tablicy `stop_words`, i przypisuje je zmiennej `key_words`. Aby kontynuować lekturę, trzeba precyzyjnie zrozumieć działanie kodu, ponieważ tego typu jednowierszowe konstrukcje często wykorzystuje się w trakcie implementowania aplikacji w języku Ruby.

Po uzyskaniu listy słów znaczących obliczenie procentowej ilości słów innych niż słowa pomijane w stosunku do wszystkich słów w tekście wymaga już tylko wykonania prostej operacji arytmetycznej:

```
((key_words.length.to_f / words.length.to_f) * 100).to_i
```

Metody `to_f` użyto dlatego, że długości ciągu znaków są traktowane jak liczby zmiennopozycyjne, dzięki czemu obliczona wartość ułamkowa będzie wyznaczona bardziej precyzyjnie. Gdy wynik dzielenia zostanie już sprowadzony do wartości procentowej, można ją z powrotem przekształcić w liczbę całkowitą.

Ostateczna implementacja mechanizmu zostanie zaprezentowana w finalnej wersji kodu aplikacji na końcu tego rozdziału.

Podsumowanie prezentujące zdania „znaczące”

Edytory tekstu, takie jak Microsoft Word, posiadają funkcje podsumowujące, które wyciągają z tekstu o znacznej objętości najbardziej znaczące zdania i na ich podstawie generują podsumowanie. Mechanizm generowania takich podsumowań stawał się z biegiem lat coraz bardziej skomplikowany, lecz jednym z najprostszych rozwiązań pozwalających na samodzielne utworzenie takiego podsumowania jest przeskanowanie zdań pod kątem pewnych cech.

Jedna z technik polega na wyszukaniu zdań, których długość jest zbliżona do średniej długości zdania w tekście i które zawierają rzeczowniki. Krótkie zdania prawdopodobnie nie niosą ze sobą wielkiego znaczenia, zaś zdania wyraźnie dłuższe od przeciętnej są po prostu zbyt długie, aby zamieszczać je w podsumowaniu. Do odnajdywania rzeczowników trzeba by zaimplementować system zdecydowanie bardziej skomplikowany niż rozwiązania, którymi zajmujemy się w tej książce, można jednak pójść „na skróty” i wyszukiwać takie słowa, które wskazują potencjalną obecność rzeczowników w tym samym zdaniu. Słowami takimi w języku angielskim mogą być na przykład „is” i „are” (na przykład „Noun is”, „Nouns are” albo „There are *x* nouns”).

Załóżmy, że należy zignorować dwie trzecie zdań zawartych w tekście: jedną trzecią zdań, które są najkrótsze, i jedną trzecią zdań najdłuższych. Pozostanie wówczas tylko jedna trzecia zdań z tekstu oryginalnego, co w zupełności wystarczy do utworzenia podsumowania.

Aby ułatwić sobie implementację, najlepiej jest zaimplementować program zupełnie od nowa, a dopiero potem przenieść zaimplementowaną logikę do głównej aplikacji. Należy więc utworzyć nowy program o nazwie *summarize.rb* i wpisać w nim następujący kod:

```
text = %q{
Ruby is a great programming language. It is object oriented and has many groovy
features. Some people don't like it, but that's not our problem! It's easy to
learn. It's great. To learn more about Ruby, visit the official Ruby Web site
today.
}
```

```
sentences = text.gsub(/\s+/, ' ').strip.split(/\.|\?|\!/)
sentences_sorted = sentences.sort_by { |sentence| sentence.length }
puts sentences_sorted.length
one_third = sentences_sorted.length / 3
ideal_sentences = sentences_sorted.slice(one_third, one_third + 1)
ideal_sentences = ideal_sentences.select { |sentence| sentence =~ /is|are/ }
puts ideal_sentences.join(". ")
```

Od razu można przetestować działanie kodu:

```
Ruby is a great programming language. It is object oriented and has many groovy features
```

Wygląda doskonale! Przeanalizujmy działanie programu.

Najpierw definiowana jest zmienna `text`, która przechowuje długi ciąg znaków zawierający kilka zdań — podobnie rozpoczyna się kod w pliku *analyzer.rb*. Następnie tekst w zmiennej `text` jest dzielony na tablicę pojedynczych zdań w następujący sposób:

```
sentences = text.gsub(/\s+/, ' ').strip.split(/\.|\?|\!/)
```

Rozwiązanie to różni się nieco od rozwiązania zastosowanego w pliku *analyzer.rb*. W łańcuchu metod znajduje się dodatkowe ogniwo w postaci metody `gsub`, a także metoda `split`. Metoda `gsub` usuwa wszystkie fragmenty złożone z więcej niż jednego znaku niewidocznego i zastępuje je pojedynczym znakiem spacji (zapis `\s+` oznacza „jeden lub więcej znaków niewidocznych"). Działanie metody ma więc tylko charakter kosmetyczny. Z kolei `strip` usuwa wszystkie nadmiarowe znaki niewidoczne z początku i końca ciągu znaków. Metoda `strip` jest więc używana w taki sam sposób jak w aplikacji analizatora.

W kolejnym kroku zdania są porządkowane na podstawie ich długości po to, by móc odrzucić jedną trzecią zdań najkrótszych i jedną trzecią zdań najdłuższych:

```
sentences_sorted = sentences.sort_by { |sentence| sentence.length }
```

Tablice i tablice asocjacyjne posiadają metodę `sort_by`, która zmienia kolejność znajdujących się w nich elementów niemal w dowolny sposób. Metoda `sort_by` przyjmuje jako argument blok kodu, którym jest wyrażenie definiujące sposób sortowania. W naszym przykładzie sortowaniu podlega zawartość tablicy `sentences`. Każde zdanie jest przekazywane do bloku w postaci zmiennej `sentence`, a następnie jego pozycja jest wyznaczana na podstawie długości zdania za pomocą wywołanej na nim metody `length`. Po wykonaniu kodu w tym wierszu zmienna `sentences_sorted` będzie zawierać tablicę ze zdaniami ułożonymi w kolejności zależnej od ich długości.

Następnie z tablicy `sentences_sorted` trzeba wyodrębnić jedną trzecią zdań, które mają średnią długość, ponieważ to właśnie te zdania z największym prawdopodobieństwem są najbardziej interesujące. W tym celu długość tablicy trzeba podzielić przez `3`, aby uzyskać liczbę zdań wchodzących w skład jednej trzeciej tekstu, po czym wczytać tak wyznaczoną liczbę zdań do nowej tablicy (warto zwrócić uwagę, że z tablicy `sentences_sorted` pobierane jest tak naprawdę o jedno zdanie więcej, aby zniwelować zaokrąglenie powstałe w trakcie dzielenia liczb całkowitych). Czynności te są realizowane w ramach następujących dwóch poleceń:

```
one_third = sentences_sorted.length / 3
ideal_sentences = sentences_sorted.slice(one_third, one_third + 1)
```

W pierwszym wierszu odczytywana jest długość tablicy, po czym zostaje ona podzielona przez `3` w celu uzyskania liczby równej „jednej trzeciej tablicy". W drugim wierszu wykonana zostaje metoda `slice`, która wycina fragment tablicy i przypisuje ten fragment zmiennej `ideal_sentences`. Załóżmy na przykład, że tablica `sentences_sorted` zawiera sześć elementów. Sześć podzielone przez trzy daje dwa, zatem trzecia część tablicy składa się z dwóch zdań. Metoda `slice` wycina *z* tablicy dwa elementy (plus jeden) począwszy *od* elementu numer dwa; ostatecznie więc z tablicy wydzielone zostaną elementy o numerach `2`, `3` i `4` (jak pamiętamy, numerowanie elementów w tablicach zaczyna się od zera). W ten sposób uzyskiwana jest środkowa z trzech części tablicy, zawierająca zdania o najlepszej długości.

W przedostatnim wierszu kodu następuje sprawdzenie, czy zdanie zawiera słowa „is" lub „are"; ostatecznie zaakceptowane zostają wyłącznie zdania, w których te słowa występują:

```
ideal_sentences = ideal_sentences.select { |sentence| sentence =~ /is|are/ }
```

W powyższym wierszu metoda `select` jest wykorzystywana w taki sam sposób jak w poleceniu usuwającym słowa pomijane w poprzednim punkcie. Blok kodu zawiera wyrażenie regularne, do którego dopasowywana jest zawartość zmiennej `sentence`. Wyrażenie zawarte w bloku ma wartość `true` jedynie wówczas, gdy w zmiennej `sentence` znajduje się słowo „is" lub „are". Dzięki temu w tablicy `ideal_sentences` znajdą się ostatecznie tylko te zdania, które należą do środkowej z trzech części zbioru zdań i zawierają słowo „is" lub „are".

W ostatnim wierszu następuje już tylko połączenie zawartości tablicy `ideal_sentences` za pomocą znaku kropki wraz ze znakiem spacji, aby zwiększyć czytelność podsumowania w ostatecznej postaci:

```
puts ideal_sentences.join(". ")
```

Analiza plików innych niż text.txt

W kodzie aplikacji w obecnej postaci jawnie wpisana jest nazwa pliku *text.txt*. Można nad takim rozwiązaniem przejść do porządku dziennego, lecz lepiej byłoby zapewnić, by przy każdym uruchomieniu programu można było wskazać konkretny plik przeznaczony do analizy.

Uwaga

> Prezentowana technika jest przydatna jedynie wówczas, gdy program *analyzer.rb* będzie uruchamiany z poziomu wiersza poleceń albo w powłoce, w systemie Mac OS X lub Linux (bądź też w Windows, jeżeli używany jest wiersz poleceń tego systemu). Jeżeli natomiast używane jest środowisko IDE działające w Windows, wówczas rozwiązanie opisane w tym punkcie można potraktować jedynie jako ciekawostkę.

Zazwyczaj gdy program uruchamia się w wierszu poleceń, na końcu polecenia wywołującego program można dopisywać dodatkowe parametry, które program ma przetworzyć. Tak samo można postąpić z aplikacją napisaną w języku Ruby.

Ruby automatycznie umieszcza wszystkie parametry dopisane w poleceniu uruchamiającym program w specjalnej tablicy o nazwie `ARGV`. Aby się o tym przekonać, można utworzyć nowy skrypt o nazwie *argv.rb* i wpisać w nim następujący kod:

```
puts ARGV.join('-')
```

W wierszu poleceń skrypt należy uruchomić w taki sposób:

```
ruby argv.rb
```

Żadne dane wynikowe nie zostaną wówczas zwrócone. W drugiej próbie skrypt należy wykonać za pomocą polecenia:

```
ruby argv.rb test 123
```

```
test-123
```

Tym razem z tablicy odczytywane są znajdujące się tam parametry, zostają one połączone znakiem myślnika i wyświetlone na ekranie. W ten sposób w kodzie analizatora z pliku *analyzer.rb* wskazanie pliku *text.txt* można zastąpić zapisem `ARGV[0]` albo `ARGV.first` (w obu przypadkach zwrócony zostanie pierwszy element tablicy `ARGV`). Polecenie, które odpowiada za wczytanie zawartości pliku, będzie wówczas mieć postać:

```
lines = File.readlines(ARGV[0])
```

Aby teraz przetworzyć plik *text.txt*, program należy uruchomić za pomocą polecenia:

```
ruby analyzer.rb text.txt
```

Więcej informacji na temat wdrażania programów i ułatwiania pracy z nimi, w tym również przez wykorzystanie tablicy `ARGV`, znajdzie się w rozdziale 10.

Program w wersji finalnej

Kod źródłowy naszego pierwszego programu został ukończony już wcześniej, wypada jednak dopisać w pliku *analyzer.rb* kod wszystkich nowych funkcji opisanych w punkcie poprzednim. W ten sposób uzyskana zostanie finalna wersja analizatora tekstów.

Uwaga

> Wszystkie kody źródłowe prezentowane w tej książce są dostępne na stronie Wydawnictwa Helion pod adresem *http://helion.pl/przyklady/prubpo.zip*. Nie trzeba więc ręcznie przepisywać kodu źródłowego przedstawionego poniżej.

Oto kod aplikacji w wersji finalnej:

```
# analyzer.rb -- Analizator tekstu

stop_words = %w{the a by on for of are with just but and to the my I has some in}
lines = File.readlines("text.txt")
line_count = lines.size
text = lines.join
```

```
# zliczenie znaków
character_count = text.length
character_count_nospaces = text.gsub(/\s+/, '').length

# zliczenie słów, zdań i akapitów
word_count = text.split.length
sentence_count = text.split(/\.|\?|!/).length
paragraph_count = text.split(/\n\n/).length

# utworzenie listy słów z tekstu, które nie są słowami pomijanymi
# zliczenie tych słów i wyznaczenie procentowej ilości słów pomijanych
# w zbiorze wszystkich słów
all_words = text.scan(/\w+/)
good_words = all_words.select{ |word| !stop_words.include?(word) }
good_percentage = ((good_words.length.to_f / all_words.length.to_f) * 100).to_i

# podsumowanie – wyodrębnienie zdań potencjalnie najbardziej znaczących
sentences = text.gsub(/\s+/, ' ').strip.split(/\.|\?|\!/)
sentences_sorted = sentences.sort_by { |sentence| sentence.length }
one_third = sentences_sorted.length / 3
ideal_sentences = sentences_sorted.slice(one_third, one_third + 1)
ideal_sentences = ideal_sentences.select { |sentence| sentence =~ /is|are/ }

# wyświetlenie użytkownikowi wyników analizy
puts "#{line_count} wierszy"
puts "#{character_count} znaków"
puts "#{character_count_nospaces} znaków, nie licząc spacji"
puts "#{word_count} słów"
puts "#{paragraph_count} akapitów"
puts "#{sentence_count} zdań"
puts "#{sentence_count / paragraph_count} zdań w jednym akapicie (średnio)"
puts "#{word_count / sentence_count} słów w jednym zdaniu (średnio)"
puts "#{good_percentage}% słów to słowa znaczące"
puts "Podsumowanie:\n\n" + ideal_sentences.join(". ")
puts "-- Koniec analizy"
```

Uwaga

Użytkownicy systemu Windows powinni zastąpić odwołanie `ARGV[0]` jawnym wskazaniem pliku *text.txt*, aby zapewnić, że skrypt będzie działał prawidłowo w FreeRIDE i SciTE. Jeżeli jednak program będzie uruchamiany z poziomu wiersza poleceń, to powinien działać prawidłowo także bez tej zmiany.

Wykonanie programu *analyzer.rb* w wersji finalnej na przykładowym tekście z powieści *Oliver Twist* spowoduje zwrócenie następujących danych wynikowych:

```
121 wierszy
6165 znaków
5055 znaków nie licząc spacji
1093 słów
18 akapitów
45 zdań
2 zdań w jednym akapicie (średnio)
24 słów w jednym zdaniu (średnio)
76% słów to słowa znaczące
```

```
Podsumowanie:

' The surgeon leaned over the body, and raised the left hand.  Think what it is
↪to be a mother, there's a dear young lamb do.  'The old story,' he said, shaking
↪his head: 'no wedding-ring, I see.  What an excellent example of the power of
↪dress, young Oliver Twist was. ' Apparently this consolatory perspective of a mother's
↪prospects failed in producing its due effect. ' The surgeon had been sitting with
↪his face turned towards the fire: giving the palms of his hands a warm and a rub
↪alternately. ' 'You needn't mind sending up to me, if the child cries, nurse,'
↪said the surgeon, putting on his gloves with great deliberation.  She had walked
↪some distance, for her shoes were worn to pieces; but where she came from, or
↪where she was going to, nobody knows. ' He put on his hat, and, pausing by the
↪bed-side on his way to the door, added, 'She was a good-looking girl, too; where
↪did she come from
-- Koniec analizy
```

Warto sprawdzić działanie programu *analyzer.rb* również na innych fragmentach tekstu (pochodzących na przykład z wybranej strony internetowej) i zobaczyć, czy można jeszcze rozszerzyć jego działanie o dodatkowe funkcje. Aplikację będzie można rozszerzyć na przykład o elementy, o których będzie mowa w następnych kilku rozdziałach książki. Warto o tym pamiętać, gdy najdzie kogoś ochota na poeksperymentowanie z jakimś kodem.

Komentarze w kodzie źródłowym

W finalnym kodzie aplikacji znajdują się wiersze poprzedzone znakiem #. Są to tak zwane komentarze, które wpisuje się w kodzie programów i z których później korzystają zarówno sami autorzy kodu, jak i inni użytkownicy, którzy ten kod później przeglądają. Komentarze przydają się szczególnie wówczas, gdy zawierają informacje na temat przyczyn, dla których zdecydowano się na zaimplementowanie rozwiązania w taki, a nie inny sposób i które po jakimś czasie mogą autorowi kodu ulecieć z pamięci.

W kodzie źródłowym języka Ruby komentarze mogą znajdować się w oddzielnych wierszach albo na końcu wiersza kodu. Oto dwa przykładowe komentarze w kodzie Ruby:

```
puts "2+2 = #{2+2}" # Dodaje 2+2 i zwraca 4
# Komentarz zapisany w oddzielnym wierszu
```

Jeśli tylko komentarz znajduje się w oddzielnym wierszu albo na końcu wiersza kodu, działanie programu nie zmieni się. Warto zamieszczać jak najwięcej komentarzy, aby po upływie dłuższego czasu łatwiej było sobie przypomnieć działanie kodu.

Podsumowanie

W tym rozdziale zaimplementowano pełną, choć prostą aplikację zgodnie z wcześniej opracowanym zbiorem wymagań i pożądanych funkcji. Następnie aplikację rozszerzono o kilka dodatkowych funkcji, których obecność nie wpływała jednak na realizację podstawowych funkcji programu. Wszystko dzięki temu, że Ruby pozwala na szybkie implementowanie aplikacji.

Na podstawie aplikacji zaimplementowanej w tym rozdziale pokazano, że jeżeli konieczne jest przeprowadzenie analizy znacznej ilości tekstu albo wykonanie obliczeń i zrobienie tych czynności ręcznie byłoby zbyt żmudne, można posłużyć się językiem Ruby.

Rozdział 4. jest ostatnim rozdziałem pierwszej części książki, w którym prezentowane są praktyczne ćwiczenia. W rozdziale 5. przedstawiona zostanie krótka historia języka Ruby, społeczność programistów Ruby, historyczne uwarunkowania niektórych funkcji udostępnianych przez Ruby, a także sposoby uzyskiwania pomocy i dołączenia do społeczności Ruby. Pisanie kodu to tylko połowa drogi do zostania doskonałym programistą!