ГЛАВА 3
Базовые блоки Ruby: данные, Выражения и управление потоком

Компьютерные программы тратят почти все свое время на манипулирование данными или ожидание поступления данных откуда-либо еще. Мы вводим слова, фразы и цифры, слушаем музыку и смотрим видео, в то время как компьютер выполняет вычисления, принимает решения и передает нам информацию. Для написания компьютерных программ важно понимать основы работы с данными и манипулирования ими.

В этой главе рассматриваются некоторые из основных форм данных, поддерживаемых Ruby, а также способы работы с ними и манипулирования ими. Темы, рассмотренные в этой главе, послужат основой для создания знания, на основе которых будут разрабатываться ваши будущие программы на Ruby.

Числа и выражения

На самом низком уровне компьютеры полностью основаны на числах, где все представлено потоками чисел. Такой язык, как Ruby, пытается изолировать вас от внутренней работы компьютера, и числа в Ruby используются в основном для тех же целей, для которых вы используете числа в реальной жизни, таких как подсчет, логические сравнения, арифметика и так далее. Давайте посмотрим, как вы можете использовать числа в Ruby и как с ними что-то делать.

Основные выражения

При программировании выражение представляет собой комбинацию данных (таких как числа или текстовые строки), операторов (таких как + или -) и переменных, которые, будучи поняты компьютером, приводят к получению ответа в той или иной форме.
Например, все эти выражения:

```
5
1 + 2
"a" + "b" + "c"
100 - 5 * (2 - 1)
x + y
```

Все четыре лучших выражения сразу же работают с irb (попробуйте их прямо сейчас!) и получите ответы, которые вы ожидаете от таких базовых операций (*1 + 2* дает результат в *3*, *"a" + "b" + "c"* дает результат в *abc* и так далее). Заключительное выражение не сработает, но все равно попробуйте его и подумайте о возвращенной ошибке и о том, как вы могли бы разрешить ситуацию (совет: установите для переменных x и y какое-нибудь значение!).

Круглые скобки работают так же, как и в обычной арифметике. Все, что находится внутри круглых скобок, вычисляется первым (или, что более технически, имеет более высокий приоритет).

■ Примечание. Вы можете изучить все разделы этой главы с помощью irb, интерактивного интерпретатора Ruby. Если вы застряли, просто выйдите из irb, набрав exit или нажав Ctrl+D в любой момент, и запустите irb снова, как показано в Главе 1. Если это не удается, нажмите Ctrl+C, затем клавишу Enter и введите exit.

Выражения регулярно используются во всех компьютерных программах, и не только с числами. Однако, понимание того, как выражения и операции работают с числами, трансформируется в базовые знания о том, как они работают с текстом, списками и другими элементами.

Переменные

В главе 2 мы рассмотрели множество концепций, включая переменные. Переменные - это заполнители или ссылки на объекты, включая числа, текст или любые типы объектов, которые вы выбрали для создания. Например:

```
x = 10
puts x
result: 10
```

Здесь вы присваиваете числовое значение 10 переменной с именем x. Имейте в виду, что вам всегда нужно инициализировать переменные (то есть присвойте им значение) перед их использованием, в противном случае вы получите сообщение об ошибке.

Вы можете называть переменные так, как вам нравится, с некоторыми ограничениями. Имена переменных должны состоять из одного слова (***или_без пробелов***); начинаться либо с буквы, либо со знака подчеркивания; содержать только буквы, цифры или знаки подчеркивания; и учитывать регистр символов. В таблице 3-1 приведены допустимые и недопустимые имена переменных.

Таблица 3-1. Допустимые и недопустимые имена переменных

Имя переменной	действительным или недействительным?
X	Valid
y2	Valid
_x	Valid
7x	Invalid (начинается с цифры)
this_is_a_test	Valid
this is a test	Invalid (не единое слово)
this'is@a'test!	Invalid(содержит недопустимые символы: ', @ и!)
this-is-a-test	Invalid (выглядит как вычитание)

Переменные важны, потому что они позволяют создавать и использовать программы, которые выполняют операции с различными данными. Например, рассмотрим небольшую программу, единственной задачей которой является вычитание двух чисел:

```
x = 100
y = 10
puts x — y

result: 90
```

Если бы код был написан просто как *puts 100 - 10*, вы бы получили тот же результат, но он не такой гибкий. Используя переменные, вы можете получить значения для x и y от пользователя, файла или из какого-либо другого источника. Единственное логическое действие - это вычитание.

Поскольку переменные являются заполнителями значений и данных, им также могут быть присвоены результаты выражения (например, *x = 2 - 1*) и могут использоваться в самих выражениях (например, *x - y + 2*). Вот более сложный пример:

```
x = 50
y = x * 100
x += y
puts x

result: 5050
```

Пройдитесь по примеру построчно. Сначала вы устанавливаете значение *x* равным 50. Затем вы устанавливаете значение *y* равным *x * 100 (50 * 100 или 5000*). Затем вы добавляете *y* к *x* перед выводом результата 5050 на экран. Это имеет смысл, но третья строка сначала неочевидна. Добавление *y* к x выглядит более логичным, если вы говорите *x = x + y*, а не *x + = y.* Это еще один короткий путь в Ruby. Потому что действие переменной, выполняющей операцию над самой собой, настолько распространено в программировании вы можете сократить *x = x + y* до *x + = y*. То же самое работает и для других операций, таких как умножение и деление, где *x *= y и x / = y* также являются допустимыми. Обычный способ увеличить значение переменной на 1 - это *x += 1*, что сокращенно означает *x = x + 1*.

Операторы сравнения и выражения

Программа без логики (*if, unless*) - это просто калькулятор. Компьютеры не просто выполняют отдельные операции с данными. Они также используют логику для определения различных вариантов действий. Основной формой логики является использование операторов сравнения в выражениях для принятия решений.

Рассмотрим систему, которая требует, чтобы пользователь был старше определенного возраста:

```
age = 10
puts "You're too young to use this system" if age < 18
```

Если вы попробуете использовать этот код, то увидите надпись *"You're too young to use this system"*, поскольку код выводит текст на экран только в том случае, если возраст не превышает 18 лет (обратите внимание на символ “меньше”). Давайте сделаем что-нибудь более сложное:

```
age = 24
puts "You're a teenager" if age > 12 && age < 20
```

Этот код не приводит к получению ответа, поскольку лицо в возрасте 24 лет не является подростком. Однако, если бы возраст составлял от 13 до 19 лет включительно, сообщение появилось бы. Это тот случай, когда два небольших выражения (*age > 12 && age < 20*) соединяются символом &&, что означает “и”. Лучший способ понять такие выражения - прочитать их вслух: **“Напечатайте текст, если возраст больше 12 лет, а возраст меньше 20”**.

Чтобы добиться обратного эффекта, вы можете использовать слово " *unless*".:

```
age = 24
puts "You're NOT a teenager" unless age > 12 && age < 20
```

На этот раз вы получите сообщение о том, что вы не подросток, указав, что вам 24 года. Это потому, что ***unless*** означает противоположность ***if*** . Программа показывает сообщение, если возраст **не соответствует** возрасту подростка.

■ Обратите внимание, что еще один интересный прием, предлагаемый Ruby, - это «***between?»*** метод, который возвращает значение *true* или *false*, если объект находится между двумя указанными значениями или равен им. Например, при работе с целыми числами, по крайней мере, *age.between?(13, 19)* эквивалентно *age >= 13 && age <= 19*.

Вы также можете проверить на равенство:

```
age = 24
puts "You're 24!" if age == 24
```

Обратите внимание, что понятие “равно” представлено двумя разными способами из-за двух разных значений. В первой строке вы говорите, что возраст должен быть равен 24, то есть вы хотите, чтобы возраст соответствовал числу 24. Однако во второй строке вы спрашиваете, равен ли возраст “24”. В первом случае вы требуете, а во втором - спрашиваете. Это различие приводит к использованию разных операторов. Следовательно, оператор равенства равен ==, а оператор присваивания - просто =. Список операторов сравнения чисел приведен в таблице 3-2.

Таблица 3-2. Полный список операторов сравнения чисел в Ruby

Значение	**Сравнения**
x > y	больше, чем
x < y	меньше, чем
x == y	равно
x >= Y	больше или равно
x <= y	меньше или равно
x <=> y	Comparison; возвращает 0, если x и y равны, 1, если x больше, и -1, если x меньше
x != y	не равно

Как вы видели ранее, можно сгруппировать несколько выражений в одно, как в следующем примере:

```
puts "You're a teenager" if age > 12 && age < 20
```

&& используется для подтверждения того, что и возраст > 12, и возраст < 20 являются истинной. Однако вы также можете проверить, соответствует ли одно из них или другое верно при использовании ||, например, так:

```
puts "You're either very young or very old" if age > 80 || age < 10
```

■ Обратите внимание, что символ |, используемый в ||, является символом канала, а не буквой l или цифрой 1.

Группировка множественных сравнений также возможна благодаря грамотному использованию круглых скобок:

```
gender = “male”
age = 6
puts "A very young or old man" if gender == "male" && (age < 18 || age > 85)
```

В этом примере проверяется, соответствует ли пол *"male”* , и если возраст меньше 18 или больше 85 лет. Если бы мы не использовали круглые скобки, строка была бы напечатана, даже если бы пол был “женский”, а возраст - старше 85 лет, потому что интерпретатор Ruby рассматривал бы сравнения на индивидуальной основе, а не делал бы первоначальные выводы.
&& зависит от соответствия *age < 18 || age > 85*.

Перебор чисел с помощью блоков и итераторов
Почти все программы требуют многократного повторения определенных операций для достижения результата. Было бы крайне неэффективно (и негибко!) написать программу для подсчета вот так:

```
x = 1
puts x
x += 1
puts x
x += 1
puts x
...
…
```

Что вам нужно сделать в таких ситуациях, так это реализовать цикл — механизм, который заставляет программу использоватьодин и тот же код снова и снова. Вот основной способ реализации цикла:

```
5.times do puts "Test" end
```

```
result: Tecm
        Tecm
        Tecm
        Tecm
        Tecm
```

Сначала вы берете число 5. Затем вызываете метод t*imes*, общий для всех чисел в Ruby. Вместо того, чтобы передавать данные в этот метод, вы передаете ему больше кода: код между *do* и e*nd*. Затем метод *times* использует код пять раз подряд, выводя предыдущие пять строк вывода.
Другой способ записать это - использовать фигурные скобки вместо *do* и e*nd*. Хотя для блоков кода, состоящих из нескольких строк, рекомендуется использовать *do* и *end*, фигурные скобки облегчают чтение кода в одной строке.
Таким образом, этот код работает точно так же:

```
5.times { puts "Test" }
```

С этого момента вы будете использовать этот стиль для отдельных строк кода, но для более длинных блоков кода вы будете использовать *do* и *end.* Это хорошая привычка, которой придерживаются почти все профессиональные разработчики Ruby (хотя из этого правила всегда есть исключения).

В Ruby один из механизмов создания цикла называется итератором. Итератор - это то, что последовательно проходит по списку элементов. В данном случае он выполняет цикл из пяти шагов, в результате чего получается пять строк теста. Для чисел доступны и другие итераторы, такие как следующие:

1.*upto(5) { ...code to loop here... }*
10.downto(5) { ...code to loop here... }
0.step(50, 5) { ...code to loop here... }

Первый пример содержит значения от 1 до 5. Второй пример содержит значения от 10 до 5. Последний пример содержит значения от 0 до 50 с шагом 5, потому что вы используете пошаговый метод для числа 0.

Что неочевидно, так это как получить доступ к числу, которое повторяется на каждом этапе, чтобы вы могли что-то сделать с ним в циклическом коде. Что, если вы захотите распечатать номер текущей итерации? Как вы могли бы разработать программу подсчета с помощью этих итераторов? К счастью, все только что описанные итераторы автоматически передают состояние итерации в циклический код в качестве параметра, который затем можно преобразовать в переменную и использовать, например, так:

1.upto(5) { |number| puts number }

result: 1
2
3
4
5

Самый простой способ понять это заключается в том, что код между { и } (или, возможно, между *do* и *end*, помните?) - это код, на котором выполняется цикл. В начале этого кода указывается число от “1 до 5” передается по “желобу” в переменную с именем *number*. Вы можете визуализировать желоб с помощью столбцов, окружающих *number*. Таким образом параметры передаются в блоки кода, которые не имеют конкретных имен (в отличие от методов классов и объектов, у которых есть имена). В предыдущей строке кода вы просили Ruby посчитать от 1 до 5. Все начинается с 1, которое передается в блок кода и отображается с помощью puts. Это повторяется для чисел от 2 до 5, в результате чего выводится результат.

Обратите внимание, что Ruby (и irb) не волнует, распределяете ли вы свой код по нескольким строкам или нет(обычно — бывают исключения!). Например, этот код работает точно так же, как и предыдущий
пример:

1.upto(5) do |number|
puts number
end

Ключевым моментом, который необходимо осознать здесь, является то, что некоторые методы будут выполнять блоки кода и передавать данные, которые

затем можно записать в переменные. В предыдущем примере метод *upto*, доступный для целых чисел, передает значение каждой итерации в блок кода, и мы “записали” его в переменную *number*.

Числа с плавающей запятой

В главе 2 вы выполняли тест, в котором делили 10 на 3, например, так:

```
puts 10 / 3
```

```
result: 3
```

Результат равен 3, хотя фактический ответ должен быть равен 3,33 в периоде. Причина этого заключается в том, что по умолчанию Ruby рассматривает любые числа без плавающей запятой как целое число. Когда вы говорите "10/3", вы просите Ruby разделить два целых числа, и в результате Ruby выдает целое число. Давайте немного уточним код:

```
puts 10.0 / 3.0
```

```
result: 3.33333333333
```

Теперь вы получаете желаемый результат. Ruby теперь работает с объектами *number* класса *Float* и возвращает значение с плавающей точкой, что обеспечивает ожидаемый уровень точности.

Могут возникнуть ситуации, когда вы не можете контролировать входящие числа, но все равно хотите, чтобы они считывались как числа с плавающей точкой. Рассмотрим ситуацию, когда пользователь вводит два числа, которые нужно разделить, и эти числа требуют точного ответа:

```
x = 10
y = 3
puts x / y
```

```
result: 3
```

Оба входных числа являются целыми, поэтому результатом, как и раньше, будет целое число. К счастью, у целых чисел есть специальный метод, который преобразует их в значения с плавающей запятой на лету. Вы бы просто переписали код следующим образом:

```
x = 10
y = 3
puts x.to_f / y.to_f
```

```
result: 3.333333333335
```

В этой ситуации, когда вы достигаете деления, и x, и y преобразуются в их эквиваленты чисел с плавающей запятой с помощью метода ***to_f*** класса *Integer*. Аналогично, числа с плавающей запятой можно преобразовать обратно в другом направлении, в целые числа, используя ***to_i***:

```
puts 5.7.to_i
```

```
result: 5
```

Мы рассмотрим этот метод, используемый другими способами, в разделе “Преобразование объектов в другие классы” далее в этой главе.

Константы
Ранее вы рассматривали разделение данных и логики с помощью переменных и пришли к выводу, что редко возникает необходимость в том, чтобы данные были прямой частью компьютерной программы. В большинстве случаев это верно, но рассмотрим некоторые значения, которые никогда не меняются — например, значение *pi*. Эти неизменяемые значения называются константами и также могут быть представлены в Ruby именем переменной, начинающимся с заглавной буквы:

Pi = 3.141592

Если вы введете предыдущую строку в irb, а затем попытаетесь изменить значение *Pi*, это позволит вам это сделать, но вы получите предупреждение:

Pi = 3.141592
Pi = 500
(irb): warning: already initialized constant Pi

Ruby предоставляет вам полный контроль над значением констант, но предупреждение выдает четкое сообщение. В будущем Ruby может ужесточить контроль над константами, поэтому уважайте этот стиль использования и старайтесь не переназначать константы в середине программы.
Внимательный читатель, возможно, помнит, что в главе 2 вы называли классы такими именами, как *Dog* и *Cat*, начинающимися с заглавных букв. Это связано с тем, что как только класс определен, он становится постоянной частью программы и, следовательно, тоже действует как константа.

Текст и строки

Если числа - это самый простой тип данных, который может обрабатывать компьютер, то текст - это наша следующая ступенька на лестнице обработки данных. Текст используется повсеместно, особенно при общении с пользователями (напрямую, по электронной почте, по интернету, *Web* или иным образом). В этом разделе вы узнаете, как манипулировать текстом, как вам заблагорассудится.

Строковые литералы
Мы уже использовали строки в некоторых из наших предыдущих примеров кода, например:

puts "Hello, world!"

Строка - это набор текстовых символов (включая цифры, буквы, пробелы и условные обозначения) любой длины. Все строки в Ruby являются объектами класса *String*, как вы можете обнаружить, вызвав метод класса *string* и выведя результат на печать:

puts "Hello, world!".class

result: String

Когда строка введена непосредственно в код (захардкожена, является частью программы), используя кавычки, как и ранее, такая конструкция называется строковым литералом. Это отличается от строки, данные которой поступают из удаленного источника, такого как пользователь, вводящий текст, файл или из Интернета.
Любой текст, предварительно встроенный в программу, является строковым литералом.

Как и числа, строки можно включать в операции, добавлять к ним и сравнивать с ними. Вы также можете присваивать строки переменным:

```
x = "Test"
y = "String"
puts "Success!" if x + y == "TestString"
```

```
result: Succes!
```

Есть несколько других способов включить строковый литерал в программу. Например, вы можете захотеть включить несколько строк текста. Использование кавычек допустимо только для одной строки, но если вы хотите охватить несколько строк, попробуйте это:

```
x = %q{This is a test
of the multi
line capabilities}
```

В этом примере кавычки заменены на *%q*{ и }. Однако необязательно использовать фигурные скобки. Вы можете использовать < и >, (и) или просто два других разделителя по вашему выбору, например ! и !. Этот код работает точно так же:

```
x = %q!This is a test
of the multi
line capabilities!
```

Однако важно помнить, что если вы используете восклицательные знаки в качестве разделителя, то любые восклицательные знаки в тексте, который вы цитируете, приведут к неправильному использованию этого метода. Если символы-разделители присутствуют в вашей строке, ваш строковый литерал завершится раньше, и Ruby сочтет оставшийся текст ошибочным. Разумно выбирайте разделители!
Еще один способ создания длинного строкового литерала - это использование *here document*, концепция, встречающаяся во многих других языках программирования. Это работает аналогично предыдущему примеру, за исключением того, что разделитель может содержать много символов. Вот пример:

```
x = <<END_MY_STRING_PLEASE
This is the string
And a second line
END_MY_STRING_PLEASE
```

В этом случае << обозначает начало строкового литерала, за которым следует разделитель по вашему выбору (В данном случае *END_MY_STRING_PLEASE*). Затем строковый литерал начинается со следующей новой строки и заканчивается, когда разделитель снова повторяется в отдельной строке. При использовании этого метода у вас вряд ли возникнут

проблемы с выбором неправильного разделителя, если вы проявите творческий подход! Обратите внимание, что в качестве разделителя нельзя использовать пробелы ; это должна быть одна группа отображаемых символов.

Строковые выражения
Использование символа **+** позволяет объединить две строки *"Test”* и *"String*", чтобы получить *"testString"*, это означает, что следующее сравнение верно, в результате чего на экран выводится *"Success!"*:

puts "Success!" if "Test" + "String" == "TestString"

Кроме того, вы можете умножить строки (только на **инт**). Например, допустим, вы хотите скопировать строку в пять раз, вот так:

*puts "abc" * 5*

result: abcabcabcabcabc

Вы также можете выполнить “больше” и “меньше” для сравнения:

puts "x" > "y"

result: false

puts "y" > "x"

result: true

■ Примечание. *"x" > "y" и "y" > "x"* - это выражения, которые при использовании оператора сравнения приводят к истинным или ложным результатам.

В этом случае Ruby сравнивает числа, которые представляют символы в строке. Как упоминалось ранее, символы хранятся в памяти вашего компьютера в виде чисел. Каждая буква и символ имеют значение, называемое значением в формате ASCII. Эти значения не особенно важны, но они означают, что таким образом вы можете сравнивать буквы и даже более длинные строки. Если вам интересно узнать, какое значение имеет тот или иной символ, узнайте это следующим образом:

puts "x".ord

result: 120

puts "A".ord

result:65

Метод *ord* класса *String* возвращает целое число, соответствующее позиции этого символа в таблице ASCII, что является международным стандартом представления символов в виде значений.
Вы можете добиться обратного результата, используя метод *chr* класса *String*. Например:

puts 120.chr

result: x

■ Примечание. Более подробное описание набора символов ASCII выходит за рамки данной книги, но в Интернете есть много ресурсов, если вы хотите узнать больше. Одним из отличных ресурсов является *http://en.wikipedia.org/wiki/ASCII.*

Интерполяция

В предыдущих примерах вы выводили результаты своего кода на экран с помощью метода *puts*. Однако в ваших результатах было мало пояснений. Если бы к вам подошел случайный пользователь и воспользовался вашим кодом, он бы не понял, что происходит, поскольку ему не было бы интересно читать ваш исходный код. Поэтому важно, обеспечить ему удобный вывод данных из ваших программ. :

```
x = 10
y = 20
puts "#{x} + #{y} = #{x + y}"
```

result: 10 + 20 = 30

Это математика для детского сада, но результат показывает интересную возможность. Вы можете вставлять выражения (и даже логику) непосредственно в строки. **Этот процесс называется интерполяцией**.

В этой ситуации интерполяция относится к процессу вставки результата выражения в строковый литерал. Способ интерполяции внутри строки заключается в том, чтобы поместить выражение в символы *#{ и }*. Еще более простой пример демонстрирует:

```
puts "100 * 5 = #{100 * 5}"
```

*result: 100 * 5 = 500*

Раздел #{100 * 5} интерполирует результат 100 * 5 (500) в строку в этой позиции, что приводит к показанному результату. Рассмотрим этот код:

```
puts "#{x} + #{y} = #{x + y}"
```

Сначала вы интерполируете значение *x*, затем значение *y,* а затем значение *x* добавляется к *y*. Вы окружаете каждый раздел соответствующими математическими символами, и, вуаля, вы получаете полное математическое уравнение:

result: 10 + 20 = 30

Вы также можете интерполировать другие строки:

```
x = "cat"
puts "The #{x} in the hat"
```

result: The cat in the hat

Или, если вы хотите написать рационально:

*puts "It's a #{"bad " * 5}world"*

result: It's a bad bad bad bad bad world

В данном случае вы интерполируете повторение строки "*bad* " пять раз. Это, безусловно, намного быстрее, чем вводить текст!

Интерполяция также работает в строках, используемых в заданиях:

*my_string = "It's a #{"bad " * 5}world"*
puts my_string

result: It's a bad bad bad bad bad world

Стоит отметить, что вы могли бы достичь тех же результатов, что и в предыдущем случае, поместив выражения за пределы строк, без использования интерполяции. Например:

x = 10
y = 20
puts x.to_s + " + " + y.to_s + " = " + (x + y).to_s
puts "#{x} + #{y} = #{x + y}"

Две строки *puts* приводят к одному и тому же результату. В первом случае используется конкатенация строк (+) для объединения нескольких разных строк вместе. Числа в *x* и *y* преобразуются в строки с помощью метода to_s. Однако во второй строке *puts* используется интерполяция, которая не требует явного преобразования чисел в строки.

Строковые методы

Мы рассмотрели использование строк в выражениях, но со строками можно сделать гораздо больше, чем просто сложить их вместе или умножьте их. В ходе экспериментов, проведенных в главе 2, вы можете использовать несколько различных методов для работы со строкой.

В таблице 3.3 приведен краткий обзор методов работы со строками, рассмотренных в главе 2.

Expression	***Output***
"Test" + "Test"	*TestTest*
"Test".capitalize	*Test*
"Test".downcase	*test*
"Test".chop	*Tes*
"Test".next	*Tesu*
"Test".reverse	*tseT*
"Test".sum	*416*
"Test".swapcase	*tEST*
"Test".upcase	*TEST*
"Test".upcase.reverse	*TSET*
"Test".upcase.reverse.next	*TSEU*

В каждом примере в таблице 3.3 вы используете метод, который предлагает строка, будь то объединение, преобразование в верхний регистр, реверсирование или просто увеличение последней буквы. Вы можете объединить методы в цепочку, как в последнем примере таблицы. Сначала вы создаете строковый литерал "*Test*"; затем вы преобразуете его в верхний регистр, возвращая *TEST*; затем вы изменяете его, возвращая *TSET*; и затем вы увеличиваете последнюю букву в нем, возвращая *TSEU*.

Другим методом, который вы использовали в главе 2, была длина, например, так:

puts "This is a test".length

result: 4

Эти методы полезны, но они не позволяют вам добиться чего-либо особенно впечатляющего с вашими строками. Давайте перейдем непосредственно к работе с самим текстом.

Регулярные выражения и манипуляции со строками

При работе со строками на продвинутом уровне возникает необходимость изучить ***регулярные выражения***. Регулярное выражение - это, по сути, поисковый запрос, и его не следует путать с выражениями, которые мы уже обсуждали в этой главе. Если вы введете *ruby* в свою любимую поисковую систему, то вы ожидаете получить информацию о том, как будет отображаться Ruby. Аналогично, если вашим регулярным выражением является *ruby* и вы запускаете этот запрос, скажем, для длинной строки, вы ожидаете, что будут возвращены любые совпадения. ***Таким образом, регулярное выражение - это строка, которая описывает шаблон для сопоставления элементов в других строках.***

■ Обратите внимание, что в этом разделе приведено лишь краткое введение в регулярные выражения. Регулярные выражения являются важной отраслью компьютерных наук, и их использованию посвящено множество книг и веб-сайтов. Ruby поддерживает большинство стандартных синтаксисов регулярных выражений, поэтому знания о регулярных выражениях, не относящиеся к Ruby, необходимы полученные из других источников данные все еще могут оказаться полезными в Ruby.

Замены

Одна вещь, которую вам часто захочется сделать, это заменить что-то в строке на что-то другое. Возьмем этот пример:

puts "foobar".sub('bar', 'foo')

result: foofoo

В этом примере вы используете метод для строки с именем *sub*, который заменяет первый экземпляр первого параметра *"bar"* на второй параметр *"foo"*, в результате чего получается *foofoo*. *sub* выполняет только одну замену за раз, в первом экземпляре текста, который должен соответствовать, в то время как *gsub* выполняет несколько подстановок одновременно, как показано в этом примере:

puts "this is a test".gsub('i', '')

result: ths s a test

Здесь вы заменили все вхождения буквы "*i"* пустой строкой. Как насчет более сложных шаблонов? Простое сопоставление буквы *"i*" не является настоящим примером регулярного выражения. Например, предположим, что вы хотите заменить первые два символа строки на *"Hello"*:

```
x = "This is a test"
puts x.sub(/^../, 'Hello')
```

result: Hellois is a test

В этом случае выполняется однократная замена на *sub*. Первый параметр, присваиваемый *sub*, является не строкой, а регулярным выражением — косые черты используются для начала и завершения регулярного выражения. В обычном выражением является **’^..“**. Символ ^ - это привязка, означающая, что регулярное выражение будет совпадать с началом любой строки в строке. Каждая из двух точек обозначает “любой символ”. В целом, /^../ означает “любые два символа сразу после начала строки”. Таким образом, вместо *This is a test* будет заменено на *"Hello"*.

Аналогично, если вы хотите изменить последние две буквы, вы можете использовать другую привязку(*anchor*):

```
x = "This is a test"
puts x.sub(/..$/, 'Hello')
```

result: This is a teHello

На этот раз регулярное выражение соответствует двум символам, которые привязаны к концу любой строки в строке.

■ Примечание. Если вы хотите привязать к абсолютному началу или концу строки, вы можете использовать \A и \z соответственно, тогда как ^ и $ привязывают к началу и концу строк в строке.

Итерация с помощью регулярного выражения

Ранее вы использовали итераторы для перемещения по наборам чисел, например, считая от 1 до 10. Что делать, если вы хотите выполнить итерацию по строке и получить доступ к каждому ее разделу отдельно? *scan* - это необходимый вам метод итератора:

```
"xyz".scan(/./) { |letter| puts letter }
```

result:x
y
z

scan соответствует своему названию. Программа сканирует строку в поисках всего, что соответствует переданному ей регулярному выражению. В этом случае вы указали регулярное выражение, которое ищет один символв одно время. Вот почему на выходе вы получаете *x, y u z* по отдельности. Каждая буква вводится в блок, присваивается букве и выводится на экран. Попробуйте этот более сложный пример:

""This is a test".scan(/../) { |x| puts x }

result: Th
is
i
s
a
te
st

На этот раз вы сканируете два символа одновременно. Просто! Однако сканирование всех символов приводит к какому -то странному результату, когда все пробелы перемешаны. Давайте настроим наше регулярное выражение так, чтобы оно соответствовало только буквам и цифрам, вот так:

"This is a test".scan(/\w\w/) { |x| puts x }

result: Th
is
is
te
st

В регулярных выражениях есть специальные символы, которые обозначаются обратной косой чертой и имеют особое значение. *\w* означает “любой буквенно-цифровой символ или знак подчеркивания”. Существует множество других символов, как показано в таблице 3-4.

Character	***Meaning***
^	Привязка к началу строки
$	Привязка к концу строки
\A	Привязка к началу строки(абсолют.)
\z	Привязка к концу строки(абсолют.)
.	Любой символ
\w	Любая буква, цифра или знак подчеркивания
\W	\W Все, что не соответствует \w
\d	Любая цифра
\D	Все, что не совпадает с \d (не цифры)
\s	Пробелы (пробелы, символы табуляции, новые строки и т.д.)
\S	Пробелы (любой видимый символ)

Используя таблицу 3-4, вы можете легко извлечь цифры из строки:

"The car costs $1000 and the cat costs $10".scan(/\d+/) do |x|
puts x
end

result:1000
10

Вы только что получили Ruby для извлечения значения из произвольного текста на английском языке! Метод сканирования использовался по-прежнему, но вы задали ему регулярное выражение, которое использует \d для сопоставления любой цифры, а +, который следует за *\d*, позволяет *\d* сопоставлять как можно больше цифр подряд. Это означает, что оно соответствует как 1000, так и 10, а не только каждой отдельной цифре за раз. Чтобы доказать это, попробуйте следующее:

```
"The car costs $1000 and the cat costs $10".scan(/\d/) do |x|
puts x
end
```

```
result: 1
        0
        0
        0
        1
        0
```

Таким образом, знак (+) после символа в регулярном выражении означает соответствие одному или нескольким символам этого типа. Существуют и другие типы модификаторов, которые приведены в таблице 3-5.

Таблица 3-5. Модификаторы символов регулярного выражения и подвыражений

Modifier	***Description***
*	Сопоставьте ноль или более вхождений предыдущего символа и как можно больше совпадений.
+	Сопоставьте одно или несколько вхождений предыдущего символа и как можно больше совпадений.
*?	Сопоставьте ноль или более вхождений предыдущего символа и как можно меньшее их количество.
+?	Сопоставьте одно или несколько вхождений предыдущего символа и как можно меньшее их количество.
?	Сопоставьте либо один, либо ни один из предыдущих символов.
{x}	Сопоставьте x вхождений предыдущего символа.
{x,y}	Соответствуют как минимум x и не более чем y вхождениям.

Последний важный аспект регулярных выражений, который вам необходимо понять на данном этапе, - это классы символов. Они позволяют выполнять сопоставление с определенным набором символов. Например, вы можете просмотреть все гласные в строке:

```
"This is a test".scan(/[aeiou]/) { |x| puts x }
```

result: i
i
a
e

[aeiou] означает “соответствует любому из *a, e, i, o* или *u*”. Вы также можете указать диапазоны символов внутри квадратных скобок, например:

"This is a test".scan(/[a-m]/) { |x| puts x }

result: h
i
i
a
e

При этом сканировании все строчные буквы между *a* и *m* совпадают.

Регулярные выражения могут быть сложными и запутанными, и им посвящены целые книги большего объема, чем эта. Большинству программистов нужно знать только основы, поскольку более продвинутые методы станут очевидны со временем, но они становятся мощным инструментом, если вы поэкспериментируете с ними и освоите их.

Вы будете использовать и развивать все методы, описанные в этом разделе, в примерах кода на протяжении всей остальной части книги.

Соответствие

Замены и извлечения определенного текста из строк полезно, но иногда вы просто хотите чтобы проверить, совпадает ли определенная строка соответствия с шаблоном на ваш выбор. Возможно, вы захотите установить быстро, содержит ли строка любую из гласных:

puts "String has vowels" if "This is a test" =~ /[aeiou]/

В этом примере **=~** - это другая форма оператора: оператор соответствия. Если строка совпадает с регулярным выражением, следующим за оператором, то выражение возвращает позицию первого совпадения (result: 2 в данном случае —который логически является «*TRUE*», поэтому выполняется условие if). Можно, конечно, поступить наоборот:

puts "String contains no digits" unless "This is a test" =~ /[0-9]/

На этот раз вы говорите, что если диапазон цифр от 0 до 9 не совпадает с тестовой строкой, сообщите пользователю, что в строке нет цифр.

Также можно использовать метод *match*, предоставляемый классом *String*. В то время как =~ возвращает позицию первого совпадения или *nil* в зависимости от того, соответствует ли регулярное выражение строке, *match* предоставляет гораздо больше возможностей. Вот простой пример:

puts "String has vowels" if "This is a test".match(/[aeiou]/)

В регулярных выражениях, если вы заключаете часть выражения в круглые скобки (и), данные, соответствующие этой части регулярного выражения, становятся доступными отдельно от остальных. *match* позволяет вам получить доступ к этим данным:

```
x = "This is a test".match(/(\w+) (\w+)/)
puts x[0]
puts x[1]
puts x[2]

result: This is
        This
        is
```

match возвращает объект *MatchData*, к которому можно обращаться как к массиву. Первый элемент (*x[0]*) содержит данные, соответствующие всему регулярному выражению. Однако каждый последующий элемент содержит то, что было сопоставлено каждой группе совпадений регулярного выражения. В этом примере первое (\w+) соответствует этому, а второе (\w+) соответствует *is*.

■ Сопоставление примечаний может оказаться гораздо более сложным, чем это, но я расскажу о более сложных способах использования в следующей главе, когда вы будете создавать свою первую полноценную программу на Ruby.

Массивы и списки

До сих пор в этой главе вы создавали отдельные экземпляры числовых и строковых объектов и манипулировали ими. Через некоторое время возникает необходимость создавать коллекции этих объектов и работать с ними в виде списка. В Ruby вы можете представлять упорядоченные коллекции объектов с помощью массивов.

Базовые массивы

Вот базовый массив:

```
x = [1, 2, 3, 4]
```

Этот массив состоит из четырех элементов. Каждый элемент является целым числом и отделяется запятыми от соседних элементов. Квадратные скобки используются для обозначения литерала массива.

Доступ к элементам осуществляется по их индексу (положению в массиве). Для доступа к определенному элементу за массивом (или переменной, содержащей массив) следует индекс, заключенный в квадратные скобки. Это называется ссылкой на элемент. Например:

```
x = [1, 2, 3, 4]
puts x[2]

result: 3
```

Как и в большинстве языков программирования, индексация массивов в Ruby начинается с *0*, поэтому первым элементом массива является элемент *0*, а вторым элементом массива является элемент *1* и так далее. В нашем примере, это означает, что *x[2]* обращается к тому, что мы бы назвали третьим элементом массива, который в данном случае является объектом, представляющим число *3*. Чтобы изменить элемент, вы можете просто присвоить ему новое значение или манипулировать им, как вы манипулировали числами и строками ранее в этой главе:

```
x[2] += 1
puts x[2]
```

result: 4

Или:

```
x[2] = "Fish" * 3
puts x[2]
```

result: FishFishFish

Массивы не нужно настраивать с помощью предопределенных записей или размещать элементы вручную. Вы можете создать пустой массив следующим образом:

```
x = []
```

Массив пуст, и попытка обратиться, скажем, к x[5] приводит к возвращению значения nil. Вы можете добавлять элементы в конец массива, помещая в него данные, например, так:

```
x = []
x << "Word"
```

После этого массив содержит один элемент: строку с надписью *"Word"*. В массивах << - это оператор для перемещения элемента в конец массива. Вы также можете использовать метод *push*:

```
x.push("Word")
```

Вы также можете удалять записи из массива одну за другой. Традиционно массивы работают по принципу “последний вошел - первый вышел”, в котором элементы могут быть помещены в конец массива, но также они могут быть извлечены из него (*pop* - процесс извлечения и удаления элементов из конца массива).

```
x = []
x << "Word"
x << "Play"
x << "Fun"
puts x.pop
puts x.pop
puts x.length
```

result: Fun
Play
1

В данном примере помещается (<<) *"Word", "Play" и "Fun"* в массив ***x***, а затем отображаетеся первый «вытолкнутый» элемент на экране. Элементы выводятся из конца массива, поэтому сначала отображается "*Fun*". Затем следует "*Play*". Для наглядности выводится длина оставшегося массива , используя метко названный метод «*length*» (метод

«*size*» тоже работает и дает точно такой же результат), который равен 1, потому что в массиве остаётся только *"Word"* .

Еще одна полезная функция заключается в том, что если массив заполнен строками, вы можете объединить все элементы в одну большую строку, вызвав метод *join* для массива:

```
x = ["Word", "Play", "Fun"]
puts x.join
```

result: WordPlayFun

Метод *join* может принимать необязательный параметр, который помещается между каждым элементом результирующей строки:

```
x = ["Word", "Play", "Fun"]
puts x.join(', ')
```

result: Word, Play, Fun

На этот раз вы объединяете элементы массива, но между каждым набором элементов ставите запятую и пробел. Это приводит к более четкому выводу.

Разбиение строк на массивы

В разделе, посвященном строкам, вы использовали *scan* для перебора содержимого строки в поисках символов, которые соответствуют шаблонам, выраженным в виде регулярных выражений. В *scan* вы использовали блок кода , который принимал каждый набор символов и отображал их на экране. Однако, если вы используете *scan* без блока кода, он возвращает массив всех совпадающих частей строки, например, так:

```
puts "This is a test".scan(/\w/).join(',')
```

result: T,h,i,s,i,s,a,t,e,s,t

Сначала вы определяете строковый литерал, затем просматриваете его в поисках буквенно-цифровых символов (используя /\w/) и , наконец, соединяете элементы возвращаемого массива запятыми.

Что делать, если вы не хотите проверять наличие определенных символов, а вместо этого хотите разделить строку на несколько частей? Вы можете использовать метод *split* и указать ему, что строка должна быть разделена на массив строк по точкам, например, так:

```
puts "Short sentence. Another. No more.".split(/\./).inspect
```

result: ["Short sentence", " Another", " No more"]

Здесь есть пара важных моментов. Во-первых, если бы вы использовали . в регулярном выражении, а не \. , вы бы разделяли каждый символ, а не точки, потому что . представляет собой “любой символ” в регулярном выражении. Поэтому вы должны экранировать его, добавив к нему обратную косую черту (экранирование - это процесс специального обозначения символа, чтобы прояснить его значение). Во-вторых, вместо объединения и распечатывания предложений вы используете метод проверки, чтобы получить более аккуратный результат.

Метод *inspect* является общим почти для всех встроенных классов в Ruby и предоставляет вам текстовое представление объекта. Например, в предыдущем выводе показан

результирующий массив таким же образом, каким вы могли бы создать массив самостоятельно. inspect невероятно полезен при экспериментах и отладке!

split также удобен для разбиения на новые строки или несколько символов одновременно, чтобы получить более четкий результат:

puts "Words with lots of spaces".split(/\s+/).inspect

result: ["Words", "with", "lots", "of", "spaces"]

С Ruby и некоторыми регулярными выражениями вы всегда сможете решить любую проблему с обработкой текста!

Также важно использовать ***p,*** альтернативу использованию inspect. Предыдущий пример также можно было бы записать таким образом:

p "Words with lots of spaces".split(/\s+/)

result: ["Words", "with", "lots", "of", "spaces"]

p - чрезвычайно полезная альтернатива использованию *puts* при работе с выражениями в irb. Она автоматически показывает структуру объектов, возвращаемых следующим за ней выражением. Мы будем широко использовать p в оставшейся части этой главы. Вам почти никогда не понадобится использовать его в рабочем приложении, **но для отладки** и обучения он превосходен, не говоря уже о быстром наборе текста!

Итерация по массивам

Итерация по массивам проста и использует метод *each*. Метод *each* обрабатывает каждый элемент массива и передает его в качестве параметра в указанный вами блок кода. Например:

[1, "test", 2, 3, 4].each { |element| puts element.to_s + "X" }

result: 1X
testX
2X
3X
4X

Хотя каждая итерация выполняется по элементам массива, вы также можете преобразовать массив "на лету", используя метод *collect*:

*[1, 2, 3, 4].collect { |element| элемент * 2 }*

result: [2, 4, 6, 8]

collect выполняет итерацию по массиву элемент за элементом и присваивает этому элементу результат любого выражения в блоке кода. В этом примере значение элемента умножается на 2.

■ Примечание. *map* функционально эквивалентен *collect*. Вы можете увидеть, что и то, и другое используется в этой книге и в другом коде , с которым вы столкнетесь.

Программисты, которые привыкли к менее динамичным и, возможно, не объектно–ориентированным языкам, могут счесть эти методы вполне современными. При необходимости в Ruby можно работать “по старинке”:

```
a = [1, "test", 2, 3, 4]
i = 0

while (i < a.length)
  puts a[i].to_s + "X"
  i += 1
end
```

Это работает аналогично каждому из приведенных ранее примеров, но перебирает массив способом, более знакомым традиционным программистам (из таких языков, как *C, BASIC* или *JavaScript*). Однако, это должно любому сразу станет ясно, почему итераторы, блоки кода и методы, такие как *each u collect*, предпочтительнее использовать в Ruby, поскольку они значительно облегчают чтение и понимание кода.

Другие методы работы с массивами
В массивах есть много интересных методов, о некоторых из которых я расскажу в этом разделе.

Сложение и объединение массивов
Если у вас есть два массива, вы можете быстро объединить их результаты в один:

```
x = [1, 2, 3]
y = ["a", "b", "c"]
z = x + y
p z

result: [1, 2, 3, "a", "b", "c"]
```

■ Обратите внимание, что здесь мы используем вместо *z.inspect*. Вернитесь к разделу ”Разбиение строк на массивы", если вы пропустили объяснение этого ключевого момента.

Вычитание и разница между массивами
Вы также можете вычесть из массива массив. Этот метод удаляет все элементы из основного массива, который находится в обоих массивах:

```
x = [1, 2, 3, 4, 5]
y = [1, 2, 3]
z = x - y
p z

result: [4, 5]
```

Проверяем наличие пустого массива
Если вы собираетесь выполнить итерацию по массиву, возможно, вам захочется проверить, есть ли в нем еще какие-либо элементы. Вы могли бы сделать это, проверив, больше ли значение *array.size* или *array.length*, чем 0, но более популярным сокращением является использование empty?:

```
x = []
puts "x is empty" if x.empty?
```

result: x is empty

Проверка массива на наличие определенного элемента
Метод *include?* возвращает значение *true*, если указанный параметр находится в массиве, и значение *false* в противном случае:

```
x = [1, 2, 3]
p x.include?("x")
p x.include?(3)
```

result: false
true

Доступ к первому и последнему элементам массива
Доступ к первому и последнему элементам массива прост с помощью методов *first u last:*

```
x = [1, 2, 3]
puts x.first
puts x.last
```
result: 1
3

Если вы передадите числовой параметр *first* или *last*, вы получите это количество элементов в начале или в конце массива:

```
x = [1, 2, 3]
puts x.first(2).join("-")
```

result: 1-2

Изменение порядка расположения элементов массива(*Reversing)*
Как и строку, массив можно перевернуть:

```
x = [1, 2, 3]
p x.reverse
```

result: [3, 2, 1]

Хэши

Массивы - это коллекции объектов, как и хэши. Однако хэши имеют другой формат хранения и способ определения каждого объекта в коллекции. Вместо присвоения позиции в списке объектам в хэше присваивается ключ, который указывает на них. Это больше похоже на словарь, чем на список, поскольку здесь нет гарантированного порядка, а есть только простые связи между ключами и значениями. Вот базовый хэш с двумя записями:

dictionary = { 'cat' => 'feline animal', 'dog' => 'canine animal' }

Переменная, хранящая хэш, называется dictionary, и, как вы можете убедиться, она содержит две записи:

puts dictionary.size

result: 2

Одна запись содержит ключ cat и значение *feline animal*, в то время как другая содержит ключ *dog* и значение *canine animal*. Как и в случае с массивами, вы используете квадратные скобки для ссылки на элемент, который хотите извлечь.
Например:

puts dictionary['cat']

result: feline animal

Как вы можете видеть, хэш можно рассматривать как массив, содержащий имена элементов вместо номеров позиций. Вы даже можете изменять значения таким же образом, как и в массиве:

dictionary['cat'] = "fluffy animal"
puts dictionary['cat']

result: fluffy animal

■ Обратите внимание, что это не сразу пригодится вам, но стоит отметить, что и ключи, и значения могут быть объектами любого типа. Таким образом, в качестве ключа можно использовать массив (или даже другой хэш). Это может пригодиться , когда в будущем вам придется иметь дело с более сложными структурами данных.

Основные методы хэширования

Как и в случае с массивами, в хэшировании есть много полезных методов, которые вы рассмотрите в этом разделе.

Перебор элементов хэша

В случае с массивами вы можете использовать метод *each* для перебора каждого элемента массива. То же самое вы можете сделать с хэшами:

x = { "a" => 1, "b" => 2 }
x.each { |key, value| puts "#{key} equals #{value}" }

result: a equals 1
b equals 2

■ Обратите внимание, что в Ruby 1.8 нет гарантии, что элементы будут возвращены в определенном порядке. В Ruby 1.9, однако, порядок, в котором элементы были вставлены в хэш, будет сохранен, и каждый из них будет возвращать их в этом порядке.

Метод each iterator для хэша передает два параметра в блок кода: во-первых, ключ, а во-вторых, значение, связанное с этим ключом. В этом примере вы присваиваете их переменным key и value и используете интерполяцию строк для отображения их содержимого на экране.

Получение ключей

Иногда вас могут не интересовать значения в хэше, но вы хотите получить представление о том, что содержит хэш. Отличный способ сделать это - просмотреть ключи. Ruby предоставляет вам простой способ сразу увидеть ключи в любом хэше, используя метод keys:

```
x = { "a" => 1, "b" => 2, "c" => 3 }
p x.keys
```

return: ["a", "b", "c"]

keys возвращает массив всех ключей в хэше, и, если у вас возникнет желание, *values* также вернет массив всех значений в хэше. Однако, как правило, вы будете искать значения на основе ключа.

Удаление элементов хэша

Удалить элементы хэша с помощью метода *delete* очень просто. Все, что вам нужно сделать, это ввести ключ в качестве параметра, и элемент будет удален:

```
x = { "a" => 1, "b" => 2 }
x.delete("a")
p x
```

result: {"b"=>2}

Условное удаление элементов хэша

Допустим, вы хотите удалить все элементы хэша, значение которых ниже определенного значения. Вот пример того, как это сделать:

```
x = { "a" => 100, "b" => 20 }
x.delete_if { |key, value| value < 25 }
p x
```

result: {"a"=>100}

Хэши внутри хэшей

Внутри хэшей могут быть хэши (или, на самом деле, любые объекты), и даже массивы внутри хэшей, внутри хэшей! Поскольку все является объектом, а хэши и массивы могут содержать любые другие объекты, вы могли бы создавать гигантские древовидные структуры с хэшами и массивами. Вот демонстрация:

```
people = {
  'fred' => {
  'name' => 'Fred Elliott',
  'age' => 63,
  'gender' => 'male',
  'favorite painters' => ['Monet', 'Constable', 'Da Vinci']
  },
  'janet' => {
  'name' => 'Janet S Porter',
  'age' => 55,
  'gender' => 'female'
  }
}

puts people['fred']['age']
puts people['janet']['gender']
p people['janet']
```

result: 63
female
{"name"=>"Janet S Porter", "age"=>55, "gender"=>"female"}

Хотя на первый взгляд структура хэша выглядит немного запутанной, она становится достаточно простой, если разбить ее на разделы. Разделы "Фред" и "Джанет" сами по себе являются простыми хэшами, но они завернутый в другой гигантский хэш, назначенный людям. В коде, который запрашивает гигантский хэш, вы просто связываете поисковые запросы друг с другом, как в случае с *puts people['fred']['age']*. Сначала он получает *people['fred']*, который возвращает хэш Фреда, а затем вы запрашиваете у него *['age']*, что дает результат 63.

Даже к массиву, встроенному в хэш Фреда, легко получить доступ:

```
puts people['fred']['favorite painters'].length
puts people['fred']['favorite painters'].join(", ")
```

result: 3
Monet, Constable, Da Vinci

Эти методы используются более подробно в следующих главах.

Управление потоком

В этой главе вы использовали сравнения вместе с *if* и *unless* для выполнения различных операций в зависимости от обстоятельств. *if* и *unless* хорошо работают в отдельных строках кода, но в сочетании с большими объемами кода, блоками кода, они становятся еще более мощными. В этом разделе вы узнаете, как Ruby позволяет управлять ходом выполнения ваших программ с помощью этих и других конструкций.

if u unless

При первом использовании *if* в этой главе использовалась демонстрация:

```
age = 10
puts "You're too young to use this system" if age < 18
```

Если значение "возраст" меньше 18 лет, строка выводится на экран. Следующий код эквивалентен:

```
age = 10
if age < 18
  puts "You're too young to use this system"
end
```

Выглядит аналогично, но код, который должен выполняться, если выражение истинно, содержится между *if* и выражением *end*, (вместо того, чтобы добавлять выражение *if* в конец отдельной строки кода). Эта конструкция позволяет помещать любое количество строк кода между оператором *if* и *end*:

```
age = 10
if age < 18
  puts "You're too young to use this system"
  puts "So we're going to exit your program now"
  exit
end
```

■ Примечание: Если вы скопируете и вставите предыдущий код непосредственно в irb, вызов *exit* в пятой строке приведет к закрытию irb, поэтому не удивляйтесь этому. Вы также увидите это в следующем примере.

стоит отметить, что *unless* может работать точно так же, потому что *unless* прямо противоположен *if:*

```
age = 10
unless age >= 18
  puts "You're too young to use this system"
  puts "So we're going to exit your program now"
  exit
end
```

Также можно использовать логику вложения, как в этом примере:

```
age = 19
if age < 21
  puts "You can't drink in most of the United States"
    if age >= 18
      puts "But you can in the United Kingdom!"
  end
end
```

if и *unless* также может содержать условие else, используемое для разграничения строк кода, которые вы хотите выполнить , если основное выражение равно *false*:

```
age = 10
if age < 18
  puts "You're too young to use this system"
```

```
else
  puts "You can use this system"
end
```

? , Тернарный Оператор

Тернарный Оператор позволяет выражению содержать мини-оператор *if/else*. Эта конструкция совершенно необязательна для использования, и некоторые разработчики не подозревают о ее существовании. Однако, поскольку она может быть полезна для создания более компактного кода, ее стоит изучить как можно раньше. Давайте рассмотрим пример:

```
age = 10
type = age < 18 ? "child" : "adult"
puts "You are a " + type
```

Вторая строка содержит тернарный оператор. Она начинается с присвоения результата выражения переменной *type*. Выражение имеет значение *age < 18 ? "child" : "adult"*. Структура следующая:

<условие> ? <результат, если условие истинно> : <результат, если условие ложно>

В нашем примере значение *age < 18* возвращает значение *true*, поэтому первый результат, *"child"*, возвращается и присваивается *type*. Однако, если бы значение *age < 18* было неверным, было бы возвращено значение *"adult"*.

Рассмотрим альтернативный вариант:

```
age = 10
type = 'child' if age < 18
type = 'adult' unless age < 18
puts "You are a " + type
```

Двойное сравнение затрудняет чтение. Другой альтернативой является использование многострочного параметра *if/else*:

```
age = 10
if age < 18
  type = 'child'
else
  type = 'adult'
end
puts "You are a " + type
```

Непосредственное преимущество тернарного оператора заключается в его краткости, а поскольку его можно использовать для построения выражений в одной строке, вы можете легко использовать его в вызовах методов или в других выражениях, где, если утверждения были бы недействительны. Рассмотрим еще более простую версию первого примера из этого раздела:

```
age = 10
puts "You are a " + (age < 18 ? "child" : "adult")
```

elsif* и *case

Иногда желательно провести несколько сравнений с одной и той же переменной одновременно. Вы можете сделать это с помощью инструкции *if*, как описано ранее:

```
fruit = "orange"
color = "orange" if fruit == "orange"
color = "green" if fruit == "apple"
color = "yellow" if fruit == "banana"
```

Если вы хотите использовать *else*, чтобы назначить что-то другое, если *fruit* не соответствует апельсину, яблоку или банану, это быстро приведет к путанице, так как вам нужно будет создать блок *if*, чтобы проверить наличие любого из этих слов, а затем выполнить те же сравнения, что и ранее. В качестве альтернативы можно использовать *elsif*, что означает “иначе, если ”:

```
fruit = "orange"
if fruit == "orange"
  color = "orange"
elsif fruit == "apple"
  color = "green"
elsif fruit == "banana"
  color = "yellow"
else
  color = "unknown"
end
```

блоки *elsif* действуют примерно так же, как блоки *else*, за исключением того, что вы можете указать совершенно новое выражение сравнения, которое должно быть выполнено, и если ни одно из них не совпадает, вы можете указать обычный блок *else*, который должен быть выполнен.

Одним из вариантов этого метода является использование блока ***case***. Наш предыдущий пример с блоком *case* становится следующим:

```
fruit = "orange"
case fruit
  when "orange"
  color = "orange"
  when "apple"
  color = "green"
  when "banana"
  color = "yellow"
else
  color = "unknown"
end
```

Этот код аналогичен блоку *if*, за исключением того, что синтаксис намного чище. Блок *case* работает с помощью сначала обрабатывается выражение, а затем выполняется поиск содержащегося в нем блока *when*, который соответствует результату этого выражения. Если соответствующий блок when не найден, то вместо него выполняется блок else в блоке case.

у *case* есть еще один трюк в запасе. Поскольку все выражения Ruby возвращают результат, вы можете сделать предыдущий пример еще короче:

```
fruit = "orange"
color = case fruit
when "orange"
"orange"
when "apple"
"green"
when "banana"
"yellow"
Else
"unknown"
end
```

В этом примере вы используете блок *case*, но результат выполнения любого внутреннего блока присваивается непосредственно *color*.

■ Примечание. Если вы знакомы с синтаксисом *switch/case* в C (или языке, связанном с C), вы можете подумать , что *case/when* эквивалентен Ruby. Это очень похоже, но в Ruby может быть сопоставлен только один “случай”, **так как выполнение не продолжается по списку параметров после того, как было установлено соответствие.**

---

while и until

В предыдущих разделах вы выполняли циклы, используя методы итератора, например, таким образом:

```
1.upto(5) { |number| puts number }
```

```
result: 1
        2
        3
        4
        5
```

Однако можно выполнить цикл кода другими способами. *while u until* позволяют выполнять циклический код на основе результатов сравнения, выполненного в каждом цикле:

```
x = 1
while x < 100
  puts x
  x = x * 2
end
```

```
result: 1
        2
        4
        8
       16
       32
```

В этом примере у вас есть блок *while*, который обозначает часть кода, которая должна повторяться снова и снова, пока выполняется выражение *x < 100*. Таким образом, *x* удваивается цикл за циклом и выводится на экран. Как только *x* становится равным *100* или больше, цикл завершается.

until обеспечивает противоположную функциональность - циклирование до тех пор, пока не будет выполнено определенное условие:

```
x = 1
until x > 99
  puts x
  x = x * 2
end
```

Также можно использовать *while* и *until* в однострочной настройке, как в случае *if* и *unless*:

```
i = 1
i = i * 2 until i > 1000
puts i

result: 1024
```

Значение i удваивается снова и снова, пока результат не превысит 1000, после чего цикл завершается (1024 равно 2 в степени 10).

Блоки кода

Блоки кода использовались в нескольких примерах кода в этой главе. Например:

```
x = [1, 2, 3]
x.each { |y| puts y }

result: 1
        2
        3
```

Каждый метод принимает один следующий блок кода. Блок кода определяется с помощью символов *{* и *}* или, альтернативно, разделителей *do* и *end:*

```
x = [1, 2, 3]
x.each do |y|
  puts y
end
```

Код между *{* и *}* или *do* и *end* представляет собой блок кода — по сути, анонимный, безымянный метод или функция. Этот код передается методу *each*, который затем запускает блок кода для каждого элемента массива.

Вы можете написать собственные методы для обработки блоков кода. Например:

```
def each_vowel(&code_block)
  %w{a e i o u}.each { |vowel| code_block.call(vowel) }
end
```

each_vowel { |vowel| puts vowel }

result: a
e
i
o
u

each_vowel - это метод, который принимает блок кода, обозначенный амперсандом (&) перед именем переменной *code_block* в определении метода. Затем он выполняет итерацию по каждой гласной в массиве букв *%w{a e i o u}* и использует метод вызова *code_block* для выполнения блока кода один раз для каждой гласной, каждый раз передавая переменную *vowel* в качестве параметра.

■ Обратите внимание, что блоки кода, передаваемые таким образом, приводят к созданию объектов, которые имеют множество собственных методов, таких как *call*. Помните, что почти все в Ruby является объектом! (Многие элементы синтаксиса не являются объектами, как и блоки кода в их буквальной форме).

Альтернативным методом является использование метода ***yield***, который автоматически обнаруживает любой переданный блок кода и передает ему управление:

```
def each_vowel
  %w{a e i o u}.each { |vowel| yield vowel }
end

each_vowel { |vowel| puts vowel }
```

Этот пример функционально эквивалентен предыдущему, хотя его назначение менее очевидно, поскольку вы не видите, что в определении функции не используется блок кода. Какой метод вы выберете, зависит от вас.

■ Обратите внимание, что в любой момент времени может быть передан только один блок кода. Невозможно принять два или более блоки кода в качестве параметров метода. Однако сами блоки кода могут не принимать ни одного, ни одного или нескольких параметров.

Также возможно сохранять блоки кода внутри переменных, используя метод лямбда:

```
print_parameter_to_screen = lambda { |x| puts x }
print_parameter_to_screen.call(100)
```

result: 100

Как и при принятии блока кода в метод, вы используете метод вызова лямбда-объекта для его выполнения, а также для передачи любых параметров.

■ Обратите внимание, что термин лямбда-код используется из-за его популярности в других местах и в других языках программирования. Вы конечно, вы можете продолжать называть их блоками кода, и их иногда называют процедурами, хотя есть некоторые различия между процедурами и лямбда-выражениями, которые следует учитывать, когда вы достигнете продвинутого уровня.

Другие полезные строительные блоки

До сих пор в этой главе мы рассматривали основные встроенные классы данных - числа, строки, массивы и хэши. Эти несколько типов объектов могут помочь вам в работе и будут использоваться во всех ваших программах. Более подробно вы познакомитесь с объектами в главе 6, но прежде чем перейти к этому, вам необходимо обратить внимание еще на несколько важных моментов.

Даты и время

Понятие, которое полезно представлять во многих компьютерных программах, - это время в виде дат и времени отсчета. Для работы с этими понятиями в Ruby предусмотрен класс *Time*.

Внутренне *Time* хранит время в виде нескольких микросекунд, начиная с эпохи *UNIX*: 1 января 1970 года 00:00:00 по Гринвичу (*GMT*)/Всемирному координированному времени (*UTC*). Это упрощает сравнение времени с помощью стандартных операторов сравнения, таких как < и >.

Давайте посмотрим, как использовать класс *Time*:

```
puts Time.now
```

```
result: 2015-07-01 00:00:00 +0100
```

Time.now создает экземпляр класса *Time*, для которого установлено текущее время. Однако, поскольку вы пытаетесь вывести это значение на экран, оно преобразуется в предыдущую строку.

Вы можете манипулировать объектами времени, добавляя и вычитая из них числа в секундах. Например:

```
puts Time.now
puts Time.now - 10
puts Time.now + 86400
```

```
result: 2015-07-01 00:00:00 +0100
        2015-06-30 23:59:50 +0100
        2015-07-02 00:00:00 +0100
```

В первом примере вы выводите текущее время, затем текущее время минус 10 секунд, а затем текущее время с добавлением 86 400 секунд (ровно один день). Поскольку в Ruby можно легко манипулировать временем с помощью обычных математических операторов, но поскольку люди предпочитают постоянно работать с минутами, часами и днями, а не с секундами, некоторые разработчики расширяют класс *Fixnum* некоторыми вспомогательными методами, чтобы еще больше упростить манипулирование временем:

```
class Fixnum
  def seconds
```

```
    self
  end
  def minutes
    self * 60
  end
  def hours
    self * 60 * 60
  end
  def days
    self * 60 * 60 * 24
  end
end

puts Time.now
puts Time.now + 10.minutes
puts Time.now + 16.hours
puts Time.now — 7.days

result: 2015-07-01 00:00:00 +0100
        2015-07-01 00:10:00 +0100
        2015-07-01 16:00:00 +0100
        2015-06-24 00:00:00 +0100
```

Не волнуйтесь, если этот код покажется вам запутанным и незнакомым, поскольку мы подробнее рассмотрим этот тип техники в следующих главах. Однако обратите внимание на стиль, используемый в заключительных инструкциях *puts*. С этими помощниками легко манипулировать датами!

Класс *Time* также позволяет создавать временные объекты на основе произвольных дат:

```
year = 2016
month = 1
day = 16
hour = 12
min = 57
sec = 10
msec = 42
Time.local(year, month, day, hour, min, sec, msec)
```

Предыдущий код создает объект *Time* на основе текущего (местного) часового пояса. Все аргументы начиная с месяца, они необязательны и принимают значения по умолчанию 1 или 0. Вы можете указывать месяцы численно (от 1 до 12) или в виде трехбуквенных сокращений их английских названий.

```
Time.gm(year, month, day, hour, min, sec, msec)
```

Предыдущий код создает объект *Time* на основе *GMT/UTC*. Требования к аргументам такие же, как и для *Time.local.*

```
Time.utc(year, month, day, hour, min, sec, msec)
```

Предыдущий код идентичен *Time.gm*, хотя некоторые могут предпочесть название этого метода.

Вы также можете преобразовать объекты *Time* в целое число, представляющее количество секунд, прошедших с начала временной эпохи *UNIX*:

```
Time.gm(2015, 05).to_i

result: 1430438400
```

Аналогично, вы можете преобразовать *epoch_time* обратно в объекты *Time*. Этот метод может быть полезен, если вы хотите хранить даты и время в файле или в формате, где необходимо только одно целое число, а не целое *Time object* :

```
epoch_time = Time.gm(2015, 5).to_i
t = Time.at(epoch_time)
puts t.year, t.month, t.day

result: 2015
        5
        1
```

Помимо демонстрации преобразования времени между объектами *Time* и эпохами, этот код показывает, что объекты *Time* также имеют методы, которые можно использовать для получения определенных фрагментов даты/времени. Список этих методов приведен в таблице 3-6.

Таблица 3-6. Методы объекта Time, используемые для доступа к атрибутам даты и времени

Method	*What the Method Returns*
hour	Число, представляющее время в 24-часовом формате (например, 21 означает 9 часов вечера).
min	Количество минут, прошедших с начала часа.
sec	Количество секунд, прошедших с начала минуты.
usec	Количество микросекунд, прошедших после секунды (в секунду приходится 1 000 000 микросекунд).
day	Номер дня в месяце.
mday	Синоним дневного метода, который считается днем “месяца”.
wday	Номер дня в календаре недели (воскресенье равно 0, суббота равна 6).
yday	Номер дня в году.
month	Номер месяца, в котором указана дата (например, 11 ноября).
year	Год, связанный с этой датой.
zone	Возвращает название часового пояса, связанного со временем.

utc?	Возвращает значение *true* или *false* в зависимости от того, находится ли время/дата в часовом поясе *UTC/GMT* или нет.
gmt?	Синоним *utc?* метод для тех, кто предпочитает использовать термин *GMT.*

Обратите внимание, что эти методы предназначены для извлечения атрибутов из даты или времени и не могут использоваться для их установки. Если вы хотите изменить элементы даты или времени, вам нужно будет либо добавить или вычесть секунды, либо создайте новый объект *Time*, используя *Time.gm* или *Time.local*.

■ Примечание. В главе 16 вы познакомитесь с библиотекой Ruby под названием *Chronicle,* которая позволяет указывать даты и время в естественной англоязычной форме и преобразовывать их в допустимые временные объекты.

Большие числа

Распространенная история об изобретении игры в шахматы вращается вокруг больших чисел. Могущественный король потребовал игру, в которую он мог бы играть в свободное время, и бедный математик придумал для него игру в шахматы. Королю понравилась эта игра, и он предложил математику все, что тот пожелает.
вознаграждение. Математик сказал, что хотел бы, чтобы на его шахматной доске был рассыпан рис. Он хотел , чтобы на первую клетку было одно зернышко, на вторую - два, на третью - четыре и так далее, удваивая количество от клетки к клетке, пока доска не заполнится. Король решил, что математик дурак, когда увидел, как мало зерен требуется, чтобы заполнить первый ряд доски.

Давайте смоделируем эту ситуацию с помощью Ruby, используя итератор и некоторую интерполяцию:

```
rice_on_square = 1
64.times do |square|
  puts "On square #{square + 1} are #{rice_on_square} grain(s)"
  rice_on_square *= 2
end
```

```
result: On square 1 are 1 grain(s)
        On square 2 are 2 grain(s)
        On square 3 are 4 grain(s)
        On square 4 are 8 grain(s)
        On square 5 are 16 grain(s)
        On square 6 are 32 grain(s)
        On square 7 are 64 grain(s)
        On square 8 are 128 grain(s)
       [Results for squares 9 through 61 trimmed for brevity.]
        On square 62 are 2305843009213693952 grain(s)
        On square 63 are 4611686018427387904 grain(s)
        On square 64 are 9223372036854775808 grain(s)
```

К 64-му квадрату вы сможете разместить на каждом из них много триллионов рисовых зерен! История заканчивается тем, что король осознает свое положение и не может

выполнить свое обещание. Однако это доказывает, что Ruby способен работать с чрезвычайно большими числами, и, в отличие от многих других языков программирования, здесь нет неудобных ограничений.

Другие языки часто имеют ограничения на размер чисел, которые могут быть представлены. Обычно это 32 двоичных бита, что приводит к ограничению значений примерно до 4,2 миллиардов в языках, поддерживающих 32-разрядный формат целых чисел. Большинство операционных систем и компьютерных архитектур также имеют схожие ограничения. Ruby, с другой стороны, легко преобразует числа, которые компьютер может обрабатывать изначально (то есть с легкостью) , в те, которые требуют дополнительной работы. Он делает это с помощью разных классов, один из которых называется *Fixnum* и представляет легко управляемые меньшие числа, а другой, удачно названный *Bignum*, представляет “большие” числа. В большинстве систем границей является число 4611686018427387903 — вы можете найти его, поэкспериментировав в irb:

puts 4611686018427387903.class

result: Fixnum

puts 4611686018427387904.class

result: Bignum

Если вы не получите точно такие же результаты, не волнуйтесь. Ruby обработает значения *Bignum и Fixnum* за вас, и вы сможете выполнять арифметические и другие операции без каких-либо проблем. Результаты могут отличаться в зависимост от архитектуры вашей системы, но поскольку эти изменения полностью обрабатываются Ruby, беспокоиться не о чем.

Диапазоны

Иногда бывает полезно иметь возможность сохранять концепцию списка, а не его фактическое содержимое. Например, если вы хотите представить все буквы от A до Z, вы можете начать создавать массив, например, так:

x = ['A', 'B', 'C', 'D', 'E' .. и так далее..]

Однако было бы неплохо просто сохранить концепцию “всего от *A* до Z”. Это можно сделать с помощью диапазона. Диапазон представляется следующим образом:

('A'..'Z')

Класс *range* предлагает простой способ преобразования диапазона в массив с помощью t*o_a*. Этот однострочный пример демонстрирует:

('A'..('A'..'Z').to_a.each { |letter| print letter }

Выражение компактно, но выполняет свою работу. Оно преобразует диапазон от *"A*" до *"Z"* в массив из 26 элементов, каждый из которых содержит букву алфавита. Затем оно выполняет итерацию по каждому элементу, используя каждый, который вы впервые использовали в в предыдущем разделе, посвященном массивам, и передает значение в *letter*, которое затем выводится на экран.

■ Примечание. Помните, что, поскольку вы использовали *print*, а не *puts*, буквы печатаются одна за другой в одной строке, тогда как puts при каждом использовании начинает новую строку.

Несмотря на то, что работа с массивами, возможно, более очевидна, класс *range* имеет свой собственный метод *each*, поэтому, хотя в нем нет массива, предыдущий пример можно было бы переписать следующим образом:

('A'..'Z').each { |letter| print letter }

В класс *range* встроены и другие методы. Также может быть полезно проверить, включено ли что -либо в набор объектов, заданных в *range*. Например, для вашего диапазона *('A'..'Z')* вы можете проверить, находится ли *R* в пределах этого диапазона, используя метод *include?* например, так:

('A'..'Z').include?('R')

result: true

Но, *r* не будет включена. Т.к она в нижнем регистре :

('A'..'Z').include?('r')

result: false

Вы также можете использовать диапазоны в качестве индексов массива для одновременного выбора нескольких элементов:
a = [2, 4, 6, 8, 10, 12]
p a[1..3]

result: [4, 6, 8]

Аналогично, вы можете использовать их для установки нескольких элементов одновременно (и исходя из текущего содержимого ***a***):

a[1..3] = ["a", "b", "c"]
p a

result: [2, "a", "b", "c", 10, 12]

Вы можете использовать диапазоны с объектами, принадлежащими ко многим различным классам, включая те, которые вы создаете сами.

Символы
Символы - это абстрактные ссылки, представленные, как правило, короткой строкой, перед которой ставится двоеточие. В качестве примеров можно привести *:blue*, *:good* и *:name*. К сожалению, для обозначения символов не существует простого в освоении способа, поэтому вам понадобится нужно прочитать весь этот раздел — возможно, даже не один раз, — чтобы усвоить его. Мне, конечно, потребовалось некоторое время, чтобы ознакомиться с

ними, но когда я начинал работать с Ruby, оказалось, что но они так широко используются рубинистами, что это того стоит!

Среди основных языков символы достаточно уникальны для Ruby (хотя в *Lisp* и *Erlang* есть схожие концепции) и, как правило, сбивают с толку большинство новых пользователей, поэтому давайте сразу перейдем к наглядному примеру:

```
current_situation = :good
puts "Everything is fine" if current_situation == :good
puts "PANIC!" if current_situation == :bad

result: Everything is fine
```

В этом примере *:good u :bad* являются символами. Символы не содержат значений или объектов, как переменные. Вместо этого они используются в качестве согласованного имени в коде. Например, в предыдущем коде вы могли бы легко заменить символы строками, например, так:

```
current_situation = "good"
puts "Everything is fine" if current_situation == "good"
puts "PANIC!" if current_situation == "bad"
```

Это дает тот же результат, но не так эффективно. В этом примере каждое упоминание “хорошего” и “плохого” создает новый объект, хранящийся отдельно в памяти, в то время как символы являются отдельными ссылочными значениями, которые инициализируется только один раз. В первом примере кода существуют только *:good* и *:bad*, в то время как во втором примере вы получаете полные строки *"good"*, "хороший" и "плохой", занимающие память.

Символы также во многих ситуациях приводят к более чистому коду. Часто для присвоения имени параметрам метода используются символы. Использование изменяющихся данных в виде строк и фиксированной информации в виде символов упрощает чтение кода.

Возможно, вы захотите рассматривать символы как буквенные константы, которые не имеют значения, но имя которых — это самый важный фактор. Если вы присвоите переменной символ :*good* и в будущем сравните эту переменную с символом *:good*, вы получите совпадение. Это делает символы полезными в ситуациях, когда вы хотите сохранить не фактическое значение, а концепцию или параметр.

Символы особенно полезны при создании хэшей, когда требуется проводить различие между ключами и значениями. Например:

```
s = { :key => 'value' }
```

Этот метод также может быть полезен, когда есть спецификация или последовательность использования ключевых имен:

```
person1 = { :name => "Fred", :age => 20, :gender => :male }
person2 = { :name => "Laura", :age => 23, :gender => :female }
```

Многие методы, предоставляемые классами Ruby, используют этот стиль для передачи информации в этот метод (и часто для возврата значений). Вы увидите примеры такой конструкции в этой книге.

Думайте о символах как о менее гибких строках в смирительной рубашке, которые используются в качестве идентификаторов. Если это все еще не произойдет чтобы это стало для вас понятным, обратите внимание на то, где далее в книге мы используем символы, и вернитесь к этому разделу.

Преобразование объектов в другие классы

Числа, строки, символы и другие типы данных - это просто объекты, принадлежащие различным классам. Числа относятся к классам *Fixnum, Bignum, Float* и/или *Integer*. Строки являются объектами класса *String*, символы - объектами класса *Symbol* и так далее.

В большинстве случаев вы можете преобразовать объекты из одного класса в другой, так что число может превратиться в строку, а строка - в число. Учесть следующее:

```
puts "12" + "10"
puts 12 + 10

result: “1210“
         22
```

Первая строка объединяет две строки, которые содержат числа, в результате чего получается *“1210“*. Вторая строка складывает два числа вместе, в результате чего получается 22.

Однако возможно преобразование этих объектов в представления в разных классах:

```
puts "12".to_i + "10".to_i
puts 12.to_s + 10.to_s

result: 22
        “1210“
```

Значения были преобразованы с помощью методов ***to_*.** Класс *String* предоставляет методы *to_i* и *to_f* для преобразования строки в объект класса *Integer* или *Float* соответственно. Класс *String* также предлагает *to_sym*, который преобразует строку в символ. Символы преобразуются в обратном порядке с помощью метода *to_s* для преобразования их в строки.

Аналогично, классы *number* поддерживают *to_s* для преобразования в текстовые представления, а также *to_i* и *to_f* для преобразования в целые числа и числа с плавающей точкой и между ними.

Резюме

В этой главе вы рассмотрели ключевые компоненты всех компьютерных программ - данные, выражения и логику — и узнали, как реализовать их с помощью Ruby. Темы этой главы служат важной основой для всех остальных глав этой книги, поскольку почти каждая будущая строка вашего кода на Ruby будет содержатьвыражение, итератор или какой-либо логический элемент.

■ Примечание. Важно помнить, что из-за обширности Ruby я не пытался описать здесь каждую комбинацию классов и методов. В Ruby существует более одного способа сделать что-либо, и мы рассмотрели сначала рассмотрим самые простые маршруты, а затем перейдем к более продвинутым техникам, описанным в книге.

Вы еще не исчерпали возможности использования различных типов данных в Ruby. Объекты и классы, как описано в Главе 2, на самом деле, тоже являются типами данных, хотя может показаться, что это не так. В Главе 6 вы будете непосредственно манипулировать объектами и классами аналогично тому, как вы манипулировали числами и строками в этой главе, и общая картина станет ясна.

Прежде чем перейти к Главе 4, где вы будете разрабатывать полноценную, но базовую программу на Ruby, давайте поразмыслим над тем, что мы рассмотрели до сих пор:

- Переменная: заполнитель, который может содержать объект (или ссылаться на него) — от чисел до текста, массивов и объектов вашего собственного создания. (Переменные были рассмотрены в главе 2, но эта глава расширила ваши знания о них).

- Оператор: то, что используется в выражении для манипулирования объектами, такими как + (плюс), - (минус), * (умножение) и / (деление). Вы также можете использовать операторы для выполнения сравнений, например, с помощью <, > и &&.

- *Integer*: целое число, например 5 или 923737.

- Число с плавающей запятой: число с десятичной частью, например 1,0 или 3,141592.

- Символ: отдельная буква, цифра, пробел или типографский символ (знак препинания и т.п.).

- Строка: набор символов, таких как "Привет, мир!" или "Ruby - это круто".В Ruby мы представляем строки, заключая их в кавычки, например, *"Hello"* или "Привет".

- Константа: переменная с фиксированным значением. Имена постоянных переменных начинаются с заглавной буквы.

- Итератор: специальный метод, такой как *each, upto* или *times*, который выполняет пошаговую обработку списка элемент за элементом. Этот процесс называется итерацией, а *each, upto u times* являются методами-итераторами.

- Интерполяция: преобразование выражений в строки.

- Массив: Коллекция объектов или значений в определенном регулярном порядке.

- Хэш: Коллекция объектов или значений, связанных с ключами. Ключ может использоваться для поиска соответствующего значения в хэше, но элементы внутри хэша не имеют определенного порядка. Это таблица поиска, очень похожая на индекс книги или словаря.

- Регулярное выражение: способ описания шаблонов в тексте, которые можно сопоставлять и сравнивать с другими. Если вы хотите поиграться с ними и их работой, посетите сайт *http://rubular.com/*, где вы найдете удобный инструмент.

- Управление потоком: процесс управления тем, какие разделы кода следует выполнять, исходя из определенных условий и состояний.

- Блок кода: фрагмент кода, часто используемый в качестве аргумента метода-итератора, который не имеет отдельного имени и сам по себе не является методом, но который может быть вызван и обработан методом, получающим его в качестве аргумента. Блоки кода также могут быть сохранены в переменных как объектах класса *Proc* (или как лямбда-выражениях).

• Диапазон: представление всего диапазона значений между начальной и конечной точками.

• Символ: уникальная ссылка, определяемая строкой, перед которой ставится двоеточие (например, *:blue* или *:name*). Символы не содержат значений, как переменные, но могут использоваться для обеспечения согласованности ссылок в коде. Их можно рассматривать как идентификаторы или константы, которые стоят особняком в том, что они абстрактно представляют. Теперь пришло время собрать воедино некоторые из этих базовых элементов и разработать полностью работоспособную программу, что вы и сделаете в Главе 4.