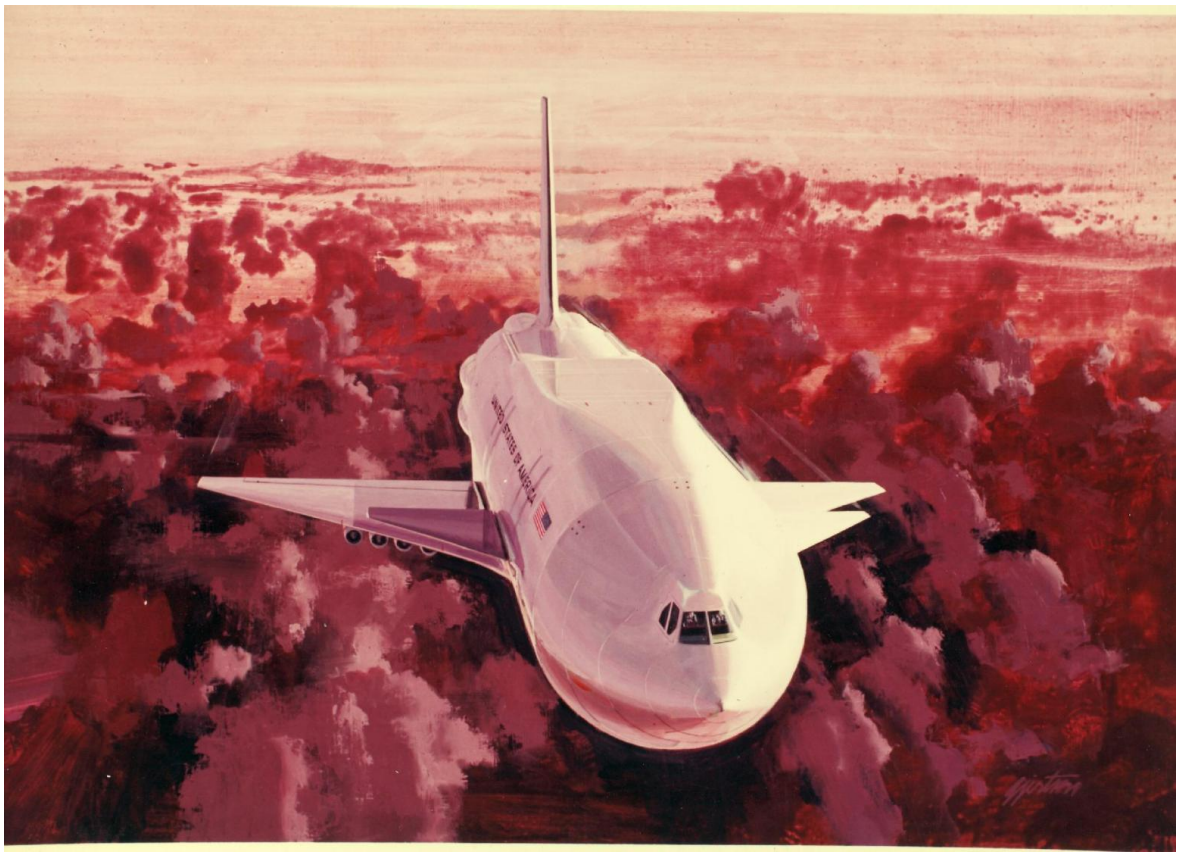
Книга по Ruby!

Шаг за логическим шагом

Николас Джонсон

Версия документа: 0.9.0 - Бета

Последнее обновление: 2017



Почему вы должны заботиться о Ruby

Здравствуйте и добро пожаловать на это супер захватывающее небольшое введение в Ruby, язык, цель которого - заставить разработчиков снова полюбить программирование.

Ruby - это выразительный, открытый язык, который позволяет вам делать очень многое практически в кратчайшие сроки.

На этом языке работает значительная часть технологических стартапов в мире. Почему? Потому что он позволяет небольшим командам разработчиков crack создавать невероятные приложения, на которые при использовании традиционных методов ушли бы месяцы или даже годы. Это язык, который обеспечивает MVP и упрощает итерации.

- **В Интернете** такие фреймворки, как Rails, Sinatra и многие другие, позволяют создавать веб-приложения за невероятно короткие промежутки времени.
- **В мобильном пространстве** Ruby Motion позволяет создавать мобильные приложения с использованием собственных компонентов.
- **На рабочем столе** Ruby позволяет создавать настольные приложения, использующие собственный пользовательский интерфейс вашей операционной системы.
- **В масштабе предприятия** JRuby работает на виртуальной машине Java, предоставляя ей готовый доступ к другим языкам на основе JVM, таким как Java и Scala.

И что это даст вам? Ruby - дружелюбное сообщество. Люди хотят помогать вам, люди хотят платить вперед. И если у вас все получится, вы сможете зарабатывать очень значительные деньги. Хороший разработчик Ruby в Лондоне будет зарабатывать около 500 фунтов стерлингов в день. Это язык, который стоит вашего времени.

Поразите своих клиентов своей потрясающей производительностью. Беритесь за побочные проекты с амбициями и готовностью.

Вы уже в восторге?

Вот некоторые из замечательных особенностей Ruby:

- Очень высокий уровень - немного - это долгий путь.
- Очень легкий, чистый, гибкий, некоторые сказали бы, поэтичный синтаксис.
- Действительно полностью объектно-ориентированный. Целые числа являются объектами. Даже методы являются объектами.
- Самоанализ встроен, а не привязан. Мощное метапрограммирование позволяет вам писать код для написания кода.
- Перегрузка операторов очень проста. Определите свои собственные типы и выполняйте с ними математические расчеты.
- Вы еще не написали метод? С помощью methodMissing Ruby может написать его за вас, пока ваш код выполняется.
- Интерпретируется, поэтому никакого цикла сборки, компилирования, отладки для ускорения вашего потока, просто пишите и обновляйте.
- Потрясающие фреймворки: Rails, Sinatra и многие другие.
- Огромная библиотека драгоценных камней, к которым можно обратиться. Вам редко приходится изобретать велосипед.
- Невероятно дружелюбное сообщество.
- Супер очаровательный.

... И еще больше приятных мелочей, от которых вы не сможете оторвать глаз.

Теперь вы взволнованы? Вы собираетесь изучать язык, разработанный специально для того, чтобы делать программистов счастливыми, который не заставляет вас прыгать через обручи, который

вызывает у вас уважение, которого вы заслуживаете, и который, если вы овладеете им, может сделать вас богатым. Добро пожаловать в Ruby!

Добро пожаловать в Ruby!

Для начала вам понадобятся некоторые основы. Я собираюсь предположить, что у вас есть копия Ruby, терминал и текстовый редактор с некоторым описанием. Вам понадобится доступ к irb, интерактивному интерпретатору ruby.

Следующие вещи должны быть правдой:

1. Вы можете создать командную строку или терминал с некоторым описанием, в идеале bash или аналогичный.
2. Если вы введете ruby в командной строке, это должно сработать и не выдавать ошибку
3. Если вы вводите irb в командной строке, это тоже должно сработать (введите exit, чтобы снова выйти)

Если для вас это не все верно, пожалуйста, обратитесь ко мне.

Привет, IRB

IRB - это интерактивный интерпретатор ruby. Если у вас есть доступ к командной строке и копии Ruby, у вас будет IRB. Это тестовый стенд. Он позволяет быстро опробовать идеи. Здесь мы собираемся использовать его для написания нашей первой программы на Ruby.

В командной строке введите

```
irb;
```

как перейти на интерактивный Ruby. Получилось? Хорошо. Теперь введите:

```
puts "Hello Ruby you charming little thing"
```

Теперь нажмите enter. Посмотрите, что вы там сделали? Puts означает "поместить строку". Это просто добавляет строку к текущему выходному потоку, который здесь является терминалом. Обратите также внимание, чего вы не сделали. Вы не вводили точки с запятой и фигурные скобки.

Теперь давайте сделаем еще один шаг вперед

В IRB введите:

```
puts "Hello Ruby " * 5
```

Видите, что получилось? Умножение работает со строками. На первый взгляд это может показаться странным, но на самом деле это согласуется с тем, как Ruby рассматривает операторы как методы. Мы вернемся к этому позже.

IRB чрезвычайно полезен для пробы чего-либо. Поскольку Ruby настолько выразителен, вы можете сделать огромное количество всего в одной строке. Ruby нравится, когда вы пишете чистый, лаконичный и, прежде всего, выразительный код, и IRB может помочь вам в этом. Вы можете выполнить многие упражнения из этой книги, используя IRB.

Упражнение IRB

На протяжении всей этой книги вы найдете множество забавных и увлекательных упражнений. Я бы посоветовал вам попробовать их, но будьте прагматичны. Если что-то действительно простое и очевидное, вы можете пропустить это, я не буду возражать. Вот простой способ для начала:

Напишите строку кода на IRB, которая напечатает наше “Hello Ruby”. Используйте puts для вывода строки.

Вы можете создать простой цикл, используя метод times:

```
10.times { puts 'cheese and crackers' }
```

Заставьте irb печатать Hello World 50 раз, 1000 раз, 100000 раз.

Упражнение в редактировании

В этом упражнении мы собираемся создать простое приложение ruby и запустить его из командной строки. Цель - просто убедиться, что наша среда работает и мы можем ее использовать.

1. Запустите свой редактор (Sublime, Brackets, Vim, Notepad и т.д.) И создайте файл с именем hello_world.rb
2. Создайте приложение hello world в редакторе и вызовите его из командной строки.

Дополнительные упражнения в редакторе

В этом упражнении мы создадим простую программу, которая считывает данные из командной строки.

1. Получите значение из командной строки, используя a = gets
2. Напишите небольшую программу, которая спрашивает пользователя, нравится ли ему сыр.
3. Если они ответят "да", дайте им немного гауды. Если они ответят "нет", сделайте им строгий выговор.

Оператор if в Ruby выглядит следующим образом:

```
if a
  puts('a was true')
else
  puts('a was false')
end
```

Переменные и константы

Давайте поговорим о переменных. Я предполагаю, что вы знаете, что такое переменная. По сравнению с некоторыми языками, Ruby очень свободен в использовании переменных, он значительно сокращает ваши расходы и по большей части предполагает, что вы знаете, что делаете.

Переменные вводятся с помощью duck

Переменные в Ruby имеют **утиный тип**. Если содержимое переменной может крякать, как утка, ей будет разрешено плавать в пруду. Интерпретатор не будет выполнять за вас никакой предварительной проверки типов. Это может расстроить людей, знакомых с .Net, C ++ или Java, и это одна из причин, по которой Ruby подходит небольшим командам разработчиков crack, поскольку вы не можете принудительно использовать интерфейс и не можете легко запретить людям передавать глупые параметры вашим методам.

Вам просто нужно довериться своему базовому интеллекту. Страшно?

Это также одна из причин, по которой Ruby настолько невероятно производителен, поскольку вам не нужно взаимодействовать с системой типов. Предполагается полиморфизм. Вам не нужно выполнять какую-либо работу, чтобы включить его. С точки зрения производительности, это огромная победа, при условии, что вы можете доверять себе и своим коллегам.

Объявление переменных

Переменные появляются при их первом объявлении. Определять их не нужно.

Например:

```
hi = "Hello Ruby"
some_big_number = 1000000
```

Это довольно разумно. Переменная может содержать все, что вам нравится, и ту же самую переменную можно переназначить для хранения чего-то совершенно другого:

```
a = 10
a = "red"
```

Немного рекомендаций: соглашения об именовании

В Ruby существуют соглашения об именовании, регулирующие имена переменных. Эти соглашения не применяются принудительно, но вы должны придерживаться их, если хотите понравиться людям, поскольку Ruby чувствителен к регистру.

Имена переменных

Имена переменных всегда начинаются со строчной буквы. По соглашению все они написаны строчными буквами с необязательными символами подчеркивания (змеиный регистр), например:

```
number_of_people
user_name
height_of_the_eiffel_tower
```

Константы

Константы начинаются с заглавной буквы и по соглашению пишутся ЗАГЛАВНОЙ БУКВОЙ_SNAKE_CASE:

```
MAX_NUMBER_OF_PEOPLE = 20
NUMBER_OF_DAYS_IN_A_LEAP_YEAR = 364
```

Обратите внимание, что константы на самом деле не являются постоянными, вы можете переопределить их, если вам действительно, действительно нужно. Вы получите предупреждение, но оно не сломается. Ruby такой, он предполагает, что вы умны. Да, Ruby - это язык, который вызывает у вас уважение, которого вы ЗАСЛУЖИВАЕТЕ.

Хитрые уловки

Есть множество маленьких хитростей, которые вы можете проделать с переменными, которые полезны и могут очень помочь при попытке казаться умными.

Цепочка присваиваний

Задания могут быть объединены в цепочки, сохраняя ввод текста, например:

```
x = y = z = 4
puts x + y + z
  => 12
```

Это работает, потому что результатом присваивания является определяемое значение, поэтому результат z = 4 равен 4.

Параллельное назначение

Ruby также поддерживает параллельное присваивание, позволяя вам назначать несколько разных переменных в одной строке, например:

```
a,b = 5,6

# a => 5
# b => 6
```

Вы можете использовать это для обмена значениями двух переменных в одной строке:

```
a,b = b,a

# a => 6
# b => 5
```

Целые числа (фиксированные числа)

Все в Ruby является объектом, включая целые числа, или фиксированные числа, как они известны в Ruby. Следовательно, у целых чисел есть методы.

```
x = 1
x.to_s
  => '1'
```

Основы математики

Вы можете выполнять любую математику, какую захотите, с целыми числами. Все работает так, как вы и ожидали:

```
1 + 2
  => 3

5 - 1
  => 4

3 * 4
  => 12

4 / 2
  => 2
```

У вас также есть общие математические сокращения, которые вы найдете в других языках. Опять же, они работают так, как вы и ожидали:

```
x = 2
  => 2

x += 5
  => 7

x *= 2
  => 14
```

Используйте ** для возведения в степень, если у вас возникнет такая склонность:

```
2 ** 2
  => 4
2 ** 3
  => 8
2 ** 4
  => 16
```

Note: There is no Incrementation operator

A surprise here. Ruby has no pre/post increment/decrement operator. For instance, x++ or x– will fail to parse. Use x += 1 instead. There are some relatively technical and quite good reasons for this to do with consistency.

Матцу это не нравится, потому что приращение ++ - единственный оператор с подразумеваемым вторым значением, поэтому у нас его нет. Вместо этого используйте a += 1.

Зацикливание с целыми числами

Помните, что все в Ruby является объектом? Это позволяет нам делать некоторые достаточно умные вещи. Например, мы можем выполнять цикл, например так:

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

hello
hello
hello
hello
hello
```

Это может показаться странным. Небольшой фрагмент кода, заключенный в фигурные скобки {помещает “hello”}, называется блоком. Блок - это встроенная функция. На самом деле мы передаем этот блок (или функцию) методу Fixnum times, который затем выполняет его 5 раз. поначалу это может показаться немного странным, но на самом деле это действительно довольно хорошо и вскоре станет второй натурой, как мы увидим, когда дойдем до раздела о блоках.

Сравнение

Как и следовало ожидать, для сравнения используются знаки больше или меньше. == тесты на равенство.

```
1 > 2
  => false

2 >= 2
```

```
  => true

1 < 2
  => true

1 == 1
  => true
```

The Spaceship Operator

Чрезвычайно полезный из них заключается в том, что оператор spaceship возвращает 1, 0 или-1 в зависимости от того, больше ли первый операнд, такой же или меньше второго. Это превосходно, поскольку такой тип сравнения лежит в основе всех известных алгоритмов сортировки.

Чтобы сделать любой класс сортируемым, все, что вам нужно сделать, это реализовать в нем оператор spaceship . Указание пользовательскому классу, как реагировать на оператора, называется перегрузкой оператора. Подробнее о перегрузке оператора позже.

```
4 <=> 3 => 1

4 <=> 4 => 0

2 <=> 10 => -1
```

Большие целые числа

Символы подчеркивания могут быть включены в целые числа, чтобы сделать их более разборчивыми. Мы делаем это не часто, но это забавный трюк, который стоит знать:

```
x = 100_000_000
y= 100000000
puts x == y
  => true
```

Плавает

В Ruby также есть числа с плавающей запятой. Числа с плавающей запятой полезны для работы с очень большими числами или числами высокой точности.

Объявите значение с плавающей точкой, просто добавив десятичную точку, вот так:

```
a = 0.5
  => 0.5

a = 1.0
  => 1.0
```

You can convert integers to floats using the to_f method like so:

```
15.to_f
  => 15.0
```

Integers are not implicitly converted to floats, so:

```
3 / 2
  => 1
```

вместо 1.5

Однако, если один из операндов уже является float , вывод будет float, так что:

```
3.0 / 2
  => 1.5

3 / 2.0
  => 1.5
```

Бесконечность

Поплавки обладают довольно удобной способностью расширяться до бесконечности, вот так:

```
1.0 / 0
  => Infinity

(1.0 / 0).infinite?
  => 1

(-1.0 / 0).infinite?
  => -1
```

Если вам нужно сказать, что что-то, скажем, стек, может содержать неограниченное количество значений, вы можете найти применение для infinity .

Упражнение - немного посчитайте

1. Получите значение из командной строки, используя a = gets
2. Напишите небольшую программу, которая спрашивает пользователя, сколько ему лет. Вам нужно будет использовать years.to_i
3. Выведите количество недель, в течение которых они прожили.
4. Предположим, что продолжительность жизни составляет 79 лет. Выведите количество недель, которые им остались.

Упражнение - повторите сами

Напишите однострочную программу, которая выводит на экран “Hello World” 250 раз

Упражнение - показатели

Используйте times для написания программы, которая выводит степени, равные 2, вплоть до некоторого максимального значения, например:

```
1
2
4
8
16
32
...
```

Строки

Ruby обладает необычайно богатым и подробным инструментарием для работы со строками. Процесс создания строк аналогичен другим языкам

```
string_1 = "Hello Ruby"
string_2 = 'Hello Everyone'
puts string_1
  => Hello Ruby
puts string_2
  => Hello Everyone
```

Вы можете заключать свою строку в одинарные или двойные кавычки.

Строки - это объекты

Строки являются реальными объектами и имеют методы:

```
s = "Hello Ruby"
puts s.reverse
  => "ybuR olleH"
```

Разрешены строковые литералы

Как и следовало ожидать, вы можете создавать строки и вызывать методы для них напрямую.

```
"Hello Ruby".upcase
  => "HELLO RUBY"
```

Встраивание переменных прямо в строки

Используйте двойные кавычки и синтаксис #{}, если вы хотите, чтобы Ruby искал переменные, встроенные в строку:

```
name = "Derrick"
puts "Hello, my name is \#{name}"
  => "Hello, my name us Derrick"
```

Приятный, простой, встроенный, читаемый.

Экранирование с помощью \

Вы можете включить escape-символы в свою строку с обратной косой чертой:

```
"Ruby\'s great! \\n Oh yes it is!"
  => "Ruby's great"
  => "Oh yes it is!"
```

“Одинарные” и "двойные" строки, заключенные в кавычки

Вопрос, который мне постоянно задают, заключается в том, следует ли нам предпочесть одинарные или двойные кавычки.

Двойные кавычки будут искать переменные и управляющие символы, поэтому технически они немного медленнее.

По этой причине вы должны использовать одинарные кавычки, за исключением случаев, когда вам нужна встроенная переменная или escape-символ.

Конкатенация строк

Сложите две строки вместе, используя простую арифметику:

```
"Hello " + "Ruby"
  => "Hello Ruby"
"Hello " << "Everybody"
  => "Hello Everybody"
"Hello " * 5
  => "Hello Hello Hello Hello Hello"
```

Скорее всего, вы захотите выполнить объединение в массиве, что-то более похожее на это:

```
["Hello", "Ruby"].join(" ");
```

Преобразование в другие типы

Строки могут быть преобразованы во множество других типов с помощью методов приведения. Существует множество встроенных методов, но не стесняйтесь писать и свои собственные. Метод “to_i” преобразует в целое число.

```
"5".to_i
  => 5
"99 Red Balloons".to_i
  => 99
```

Преобразование из других типов

Строки могут быть созданы из любого другого типа с помощью метода to_s. Например, если у вас есть целое число, и вам нужно, чтобы оно было строкой, сделайте это следующим образом:

```
5.to_s
  => "5"
```

-# ## Упражнение - Строковый API

-# String API богат и глубок

Упражнение - Работа со строками

Ознакомьтесь с вышесказанным и попробуйте следующее

1. Создайте строку и присвоите ее переменной.
2. Переверните строку, используя обратный метод.
3. Постоянно переворачивайте строку, используя метод reverse! . Методы, которые заканчиваются на ! разрушительны, они влияют на исходную переменную.
4. Используйте gets для заполнения переменной, содержащей ваше имя. Теперь используйте синтаксис #{} для создания строки приветствия, которая включает эту переменную.

Упражнение GSub

Строковый метод gsub дает нам глобальную подстановку. Читайте об этом здесь:

http://ruby-doc.org/core-2.1.4/String.html#method-i-gsub

У меня проблема, когда я обычно набираю a вместо e. Для меня это ужасная проблема, но вы можете помочь. Напишите код для замены всех экземпляров буквы ‘a’ в строке буквой ‘e’.

Упражнение - String API

Ознакомьтесь со строковым API здесь:

http://ruby-doc.org/core-2.2.2/String.html

В моем приложении у меня есть строка, подобная этой:

```
email = "   dave@davely.com ";
```

Мне нужно избавиться от этого пробела. Пожалуйста, очистите его для меня.

Теперь у меня есть строка, подобная этой:

```
first_name = "  dave";
```

Мне нужно, чтобы это было “Дейв”.

Пожалуйста, приведите его в порядок, чтобы мы могли сохранить в базе данных.

Упражнение - Кастинг

1. Напишите код, который получает номер и имя из командной строки, скажем “Hal” и “123”.
2. Теперь выведите “Прости, Дейв, я не могу этого сделать” 123 раза

Упражнение - strftime

Класс Time позволит вам легко форматировать объект Time в виде строки, используя strftime.

Получите текущее время с помощью Time.now

Теперь ознакомьтесь с API strftime здесь http://ruby-doc.org/core-2.2.2/Time.html#method-i-strftime

Теперь выведите время в таком формате:

“20 января 1946 года в 12:45”

Функции

Функции объявляются с помощью ключевого слова def:

```
def greeting
  puts "Hello Ruby"
end

greeting()
  => Hello Ruby
```

Прием параметров

Функции могут принимать параметры, как вы и ожидали. Мы передаем их следующим образом:

```
def greet(name)
  puts "hello #\{name}"
end
```

```
greet("dave")
  => "hello dave"
```

Необязательные фигурные скобки

При вызове функции фигурные скобки необязательны.

```
greet "dave"
  => "hello dave"
```

Это действительно хороший синтаксис, и он становится самостоятельным, когда мы начинаем писать методы.

Возвращает значение

Функции могут возвращать значение. Мы передаем значение обратно, используя оператор return, примерно так:

```
def say_hello_to(name)
  return "hello #\{name}"
end

puts say_hello_to "dave"
  => "hello dave"
```

get_greeting_for “dave” **вычисляет** строку “hello dave". Эта строка принимается puts, который затем выводит ее на экран.

Необязательные операторы возврата

Оператор return также необязателен. Если он опущен, функция вернет последнее вычисленное выражение, поэтому:

```
def get_greeting_for(name)
  "hello #\{name}"
end

puts get_greeting_for "dave"
  => "hello dave"
```

Это чистый и полезный синтаксис для коротких методов, таких как геттеры.

Значения по умолчанию

Мы можем установить значение аргумента по умолчанию, поэтому, если значение не передано, наша функция все равно будет работать:

```ruby
def get_greeting_for(name="anonymous")
  return "hello #\{name}"
end

puts get_greeting_for
  => "hello anonymous"
```

Обратите внимание, что если у нас есть несколько аргументов, и некоторые из них отсутствуют, они будут заполнены слева направо, поэтому последние примут значения по умолчанию.

Итог

- Функции в Ruby создаются с использованием ключевого слова def (сокращение от define).
- Функции, которые существуют в объекте, обычно называются методами.
- Функции и методы те же, за исключением того, что они принадлежат объекту.
- Объекты создаются из классов с использованием метода .new

Упражнение - Знакомство

Напишите простую функцию, которая приветствует пользователя по имени. Она должна получать имя и возвращать строку.

Если он вызывается без параметров, он должен говорить “Привет, аноним”.

Упражнение - простая функция

Напишите функцию, которая получает значение и выводит строку, содержащую все числа до этого значения включительно.

Интегрируйте это в приложение командной строки.

Управление потоком

Ruby любит, когда вы указываете ему, что делать.

if/elsif/else/end

Инструкции If присутствуют. Они работают так, как вы ожидаете. Обратите внимание, что фигурные скобки не требуются.

```ruby
bacon = true
fish = false
if fish
  puts 'I like fish'
elsif bacon
  puts 'I like bacon'
else
```

```
  puts "I don't like fish or bacon"
end

  => 'I like bacon'
```

если не

Ключевое слово unless противоположно ключевому слову if. Оно эквивалентно !if (не if). Это может сделать ваш код более читабельным.

```
bacon = false
unless bacon
  puts 'fish'
else  puts 'bacon'
end

  => "fish"
```

Модификаторы операторов

Ruby любит, когда вы пишете кратко. Ключевые слова if и unless также могут быть размещены после строки кода, которую вы можете захотеть выполнить, а можете и не захотеть. При использовании таким образом они называются модификаторами операторов:

```
user = "dave"

puts "Hello \#{user}" if user
puts 'please log in' unless user
```

Тернарный оператор

Как и большинство языков, Ruby включает в себя троичный оператор. Он работает так, как вы и ожидали. Если первая часть принимает значение true, выдается первый результат, в противном случае выдается второй результат:

```
bacon = true
puts bacon ? 'bacon' : 'fish'
  => "bacon"
```

Это эквивалентно:

```
bacon = true
if bacon
  puts 'bacon'
else
  puts 'fish'
end
```

```
  => "bacon"
```

Логические значения, nil и undefined

Логические значения в Ruby логичны и понятны

Что считается false?

В Ruby только Nil и False имеют значение false. Если оно существует, оно является true. Это включает нули, пустые строки и т.д. Это потому, что 0 является объектом в Ruby, как и пустая строка ""

Мы можем протестировать это с помощью короткой функции, которая определяет, принимает ли параметр значение true или false, например:

```
def true?(value)
  if (value)
    true
  else
    false
  end
end

true?(false)     # => false
true?(nil)       # => false
true?(0)         # => true
true?("")        # => true
true?(true)      # => true
true?(15)        # => true
true?([0,1,2])   # => true
true?('a'..'z') # => true
true?("pears")   # => true
true?(:bananas) # => true
```

Переменной присваивается значение nil, если переменная была объявлена, но ни на что не указывает (помните, что Ruby полностью объектно-ориентирован, поэтому все переменные являются указателями на объекты)

Ноль

Переменной присваивается значение nil, если она была объявлена, но не содержит значения. Nil существует как тип и имеет методы. Например:

```
a = nil
a.to_s
  => ""
```

```
a.nil?
  => true
```

Nil - это очень полезная вещь, которую можно возвращать. Это означает “нет значения”.

Не определено

Переменная не определена, если она не была объявлена. Вы можете проверить это с помощью оператора defined? .

```
a = 1
defined? a
  => "local-variable"
defined? b
  => nil
```

Булева алгебра

Вы можете выполнять все стандартные действия, используя булеву алгебру.

```
true && true
  => true
true || false
  => true
12 == 12
  => true
```

все они поддерживаются.

Специальное использование логических ИЛИ ||

Есть полезные вещи, которые можно сделать с помощью команды OR || . Вторая часть вычисляется, только если первая часть возвращает значение false (nil или false равняется false), а возвращаемое значение является последним вычисленным значением. Rails использует это, позволяя вам делать такие аккуратные вещи, как это:

```
name = nil;
user_name = name || "Anonymous Coward";
```

Здесь у нас есть значение по умолчанию. Если name равно nil, вместо него будет использоваться anonymous coward.

Упражнение - Student Simulator

Простой для начала. У меня есть логическая переменная с именем hungry:

```
hungry = true;
```

1. Если hungry - это true, выведите “Make toast”.
2. Если это не так, выведите слово: “Go to sleep”.

Упражнение - Выбор имени кота

Я хочу выбрать имя для своей кошки, но по личным и идеологическим причинам меня интересуют только имена кошек, начинающиеся на букву R.

1. Создайте функцию, которая получает имя cat.
2. Если cat начинается с буквы “R”, верните имя, в противном случае верните nil. (Помните, что вам не нужно явно возвращать nil, просто не возвращайте.)

Вы можете получить первую букву строки, используя синтаксис квадратных скобок:

```
"hello"[0]
# => "h"
```

Дополнительное расширение

Используйте цикл while для перебора функции до тех пор, пока не будет предложено приемлемое имя. Цикл while в ruby выглядит следующим образом:

```
x = 0
while x < 5
  x += 1
  puts x
end
```

Дальнейшее расширение

Поймайте случай, когда я случайно ввел “Ruby the Cat" вместо “ruby the Cat". Самый простой способ сделать это - использовать нижний регистр.

http://ruby-doc.org/core-2.1.0/String.html#method-i-downcase

Операторы Case

Конечно, в Ruby также есть операторы case. Вы почти никогда не увидите их в дикой природе, поскольку операторы case немного, ну, 1990-х годов, но если вы решите, что они вам нужны, вот как они работают.

```
refreshment_type_required = "biscuit"
suggest_you_eat = case refreshment_type_required
  when "pastry" : "cinnamon danish whirl"
  when "hot drink" : "mocha with sprinkles"
  when "biscuit" : "packet of bourbon creams"
  else "glass of water"
end
```

Оператор case автоматически прерывается, если вы нажмете на соответствующий термин, вам не нужно указывать ему прерываться, как в некоторых других языках.

Блоки качаются

Блок - это безымянная функция, лямбда-выражение, которое принимается функцией и которое может указывать ей, что делать.

Блоки - вот где действительно начинается самое интересное. Блок в Ruby - это своего рода безымянная функция, которая может быть передана методу с использованием специального синтаксиса. Он может быть сколь угодно длинным или сложным и занимать много строк. Вот простой пример:

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

Ruby Ruby
Ruby Ruby
Ruby Ruby
Ruby Ruby
Ruby Ruby
```

Блок здесь представляет собой бит кода: помещает “Ruby Ruby”. Этот фрагмент кода передается методу times. В Ruby мы используем блоки для всего, особенно при создании циклов, но также и множеством других способов. Пойдем со мной, когда мы войдем в мир циклов…

Почему это круто?

Это супер здорово, потому что означает, что число 5 само по себе знает, как выполнять итерации до самого себя. У нас идеальная инкапсуляция. Нам не нужен цикл for. Нам не нужно делать никаких предположений о реализации. Все это скрыто под поверхностью.

Передача параметров в блок

Мы можем заставить нашу функцию передавать параметры в свой блок. Мы делаем это, используя синтаксис | | “chute” . Например:

```
5.times{|i| puts i}

0
1
2
3
4
```

Здесь метод times передает блоку число, которое доступно в переменной i . Это один из многих способов, которыми мы выполняем итерацию в Ruby.

Блоки лучше циклов

Одним из наиболее распространенных и удобных для программистов применений блоков является циклирование. Многие объекты, в первую очередь массивы и хэши, принимают блок, а затем применяют этот блок к каждому из своих элементов по очереди. Весь циклический код инкапсулирован, что означает, что нам не нужно беспокоиться о внутренней структуре массива.

```
people = ["jim","harry","terrence","martha"]
people.each { |person| puts person }
```

```
=> "jim"
"harry"
"terrence"
"martha"
```

Как мы видели ранее, метод Fixnum.times работает аналогичным образом:

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

```
=> Ruby
Ruby
Ruby
Ruby
Ruby
```

Если бы мы хотели выполнить итерацию по некоторым конкретным числам, мы могли бы использовать метод upto для создания перечислимого объекта, а затем выполнить итерацию по нему. Обратите внимание:

```
5.upto(10) {|i| puts i}
```

```
=> 5
6
7
8
9
10
```

Мы также могли бы выполнять итерации по объекту Range. вскоре подробнее о диапазонах:

```
(6..8).each {|i| puts i}
```

```
=> 6
7
8
```

Наиболее распространенное использование цикла - итерация по массиву - полностью описано, и фактически, при написании кода на Ruby мы почти никогда не пишем циклические конструкции, к которым вы могли бы привыкнуть. Они описаны в блоках

each_with_index

Если по какой-то причине нам нужно получить индекс массива, мы можем сделать это также с помощью Array.each_with_index .

Этот метод принимает блок с двумя параметрами, значением и индексом. Мы можем использовать его следующим образом:

```
people = ["jim","harry","terrence","martha"]
people.each_with_index { |person, i|   puts "person: #\{i} is called #\{person}" }
  => person: 0 is called jim
  person: 1 is called harry
  person: 2 is called terrence
  person: 3 is called martha
```

Коротко и завитушками

Мы можем объявить блок двумя способами. Мы можем использовать синтаксис фигурных скобок или синтаксис do / end.

Разница между ними заключается в том, что синтаксис фигурных скобок может занимать только одну строку кода, тогда как синтаксис do / end может занимать столько кода, сколько вам нравится, даже целый шаблон веб-страницы, если это необходимо.

Вот пример синтаксиса фигурных скобок:

```
favourable_pets = ["kittens","puppies","hamsters"]

favourable_pets.each_with_index { |pet, i| puts pet; puts i }
  => kittens
  puppies
  hamsters
```

А вот пример синтаксиса do / end. Обратите внимание, что код более растянут. Это более читабельно для больших блоков кода.

```
favourable_pets.each_with_index do |pet, i|
  puts pet
  puts i
end

  => kittens
  0
  puppies
  1
```

```
hamsters
2
```

Упражнение - Gsub принимает блок

Метод String.gsub найдет и заменит подстроки в тексте. Это ужасно полезно, но иногда нам нужно больше, нам нужно находить строки в тексте и манипулировать ими.

Допустим, у вас есть строка, содержащая URL-адреса, возможно, взятые из Twitter. Вы могли бы заменить все URL-адреса следующим образом:

```
tweet_text = "hello http://www.google.com hi"

tweet_text.gsub(/http:\\/\\/[^ ]*/, "A URL was here.")

  => "hello A URL was here. hi"
```

Это нормально, но что, если бы мы захотели заменить URL действующей ссылкой. Метод gsub необязательно примет блок. Блок получает параметр, содержащий совпадение. Затем блок должен возвращать значение, которое используется для замены соответствия.

1. Создайте строку, содержащую один или несколько URL-адресов. Теперь напишите код, используя gsub, чтобы создать гиперссылки на все URL-адреса.
2. Предположим, что ваша строка - это твит. Теперь напишите код для создания гиперссылок на все хэш-теги. Хэштеги начинаются с # и заканчиваются пробелом или переводом строки.
3. Наконец, напишите код для создания гиперссылок на все имена пользователей. Имена пользователей начинаются с @.

Карта - переворачивание букв

Мы можем разделить строку на массив, используя функцию split, вот так:

‘hello world’.split(’ ’) => [‘привет’, ‘мир’]

Используйте string.split и метод Array#map, чтобы взять предложение и поменять местами буквы всех слов, вот так.

```
"Hello there" # => "olleH ereht"
```

Сначала разделить, затем сопоставить с обратным.

Для получения бонусных баллов используйте заглавную букву при написании:

```
"Olleh ereht"
```

Inject - добавление всех чисел

Inject - это невероятно удобная функция, которая позволит вам вставлять блок между каждым элементом массива. Блок, в свою очередь, получит следующий элемент в массиве и результат

предыдущего вызова блока.

У меня есть массив чисел, подобный этому:

```
[4,9,6,3]
```

1. Используйте inject для сложения всех чисел в массиве.
2. Используйте inject для умножения всех чисел в массиве.

Файл - сохранение в файловой системе

При подаче файла также используется блок. Мы вызываем File.open и передаем ему блок. Внутри блока у нас есть доступ к объекту file.

```
File.open("path/to/file", 'wb') do |file|
  file.write('hey there!', :ruby)
end
```

Напишите код, который сохраняет случайную строку в файл.

RSpec

Существует множество тестовых наборов для Ruby. RSpec - самый популярный. Он вдохновил многих имитаторов, таких как Jasmine для JavaScript и PHPSpec для PHP.

Получение RSpec

Установите rspec с помощью

```
gem install rspec
```

или, если вы предпочитаете, добавьте это в свой Gemfile и bundle.

Теперь у вас есть новая команда терминала rspec. Тип:

```
rspec;
```

в терминале протестируйте вашу установку.

Написание спецификации

RSpec очень прост в настройке. Создайте папку с именем spec в том же каталоге, что и ваш код. Именно там будут находиться ваши спецификации.

В папке spec создайте файл с именем test_spec.rb. Поместите в него следующий код:

```
describe "an example spec" do
  it "passes" do
    expect(true).to be(true)
  end
end
```

запустите свои спецификации с помощью:

```
rspec spec/*
```

Вы должны увидеть сообщение, подобное этому, сообщающее о том, что спецификации переданы:

```
Finished in 0.00105 seconds (files took 0.11919 seconds to load)
1 example, 0 failures
```

Тестирование некоторого реального кода

Давайте напишем функцию будильника, которая будит нас по утрам. Когда мы вызываем ее, она выдает нам сообщение: “Бип-бип-бип”. Это поможет нам проснуться.

```
def wake_me_up
  "Beep beep beep"
end
```

Сохраните это в файле с именем clock.rb

Теперь давайте напишем спецификацию для этого. Сначала нам нужно импортировать наши часы, затем мы вызываем код, затем говорим, каким должен быть результат:

```
require_relative '../clock'

describe "alarm clock" do
  it "beeps" do
    expect(wake_me_up).to eq('Beep beep beep')
  end
end
```

Запустите это и убедитесь, что это работает

be vs eq

Обратите внимание, что выше мы использовали для eq, а не для be. Тесты Be для одного и того же объекта. Тесты Eq на равенство объектов.

Когда вы создаете две строки, эти строки не являются одним и тем же объектом, но они содержат одинаковые символы. Произойдет сбой, эквалайзер пройдет.

Сначала протестируйте

Все это хорошо, но как насчет разработки на основе тестирования (TDD). В этой методологии мы сначала пишем тест, наблюдаем, как он завершается неудачей, затем пишем код, чтобы он прошел.

Иногда мне хочется спать, и я хотел бы, чтобы меня будили с большей силой. Я бы хотел, чтобы мой будильник принимал необязательный параметр и будил меня с большей силой.

Сначала мы пишем спецификацию

```
require_relative '../clock'

describe "alarm clock" do
  it "beeps" do
    expect(wake_me_up).to eq('Beep beep beep')
  end
  it "beeps louder" do
    expect(wake_me_up(6)).to eq('Beep beep beep beep beep beep')
  end
end
```

Теперь мы запускаем спецификацию и видим красный цвет:

```
1) alarm clock beeps
   Failure/Error: expect(wake_me_up(6)).to eq('Beep beep beep beep beep beep')
   ArgumentError:
     wrong number of arguments (1 for 0)
   # ./clock.rb:5:in `wake_me_up'
  # ./spec/clock_spec.rb:16:in `block (2 levels) in <top (required)>'

Finished in 0.00161 seconds (files took 0.14819 seconds to load)
2 examples, 1 failure

Failed examples:

rspec ./spec/clock_spec.rb:15 # alarm clock beeps
```

Обратите внимание, что он точно сообщает нам, в чем и где проблема. Давайте напишем код, чтобы наш тест прошел успешно.

```
def wake_me_up(i = 3)
  ("beep " * i).strip.capitalize
```

```
end
```

Запустите тест еще раз и добейтесь успеха, у нас есть проходной тест.

Многие разработчики говорят, что живут ради этих зеленых моментов. Вы можете любить тестирование так же сильно, как это, а можете и не любить, но независимо от этого, хороший набор пройденных тестов может дать вам спокойствие в выходные и помочь вам лучше спать по ночам, зная, что вы проделали хорошую работу.

Упражнение - Напишите несколько тестов

Ознакомьтесь с документацией по сопоставлению RSpec здесь:

https://www.relishapp.com/rspec/rspec-expectations/docs/built-in-matchers

В разделе о блоках вы написали код для изменения местами отдельных слов в строке, но не самой строки, так что:

```
"Hello World";
```

становится:

```
"olleH dlroW";
```

1. Извлеките этот код в функцию и напишите для него спецификацию.
2. Проделайте то же самое для упражнений gsub и inject.

Массивы

Как и большинство языков, Ruby позволяет создавать массивы для хранения нескольких значений. Массивы в Ruby одномерны, хотя вы можете объявить массив массивов, если почувствуете необходимость.

Создание массивов

Создайте массив, используя квадратные скобки, следующим образом:

```
cakes = ["florentine", "lemon drizzle", "jaffa"]
```

Доступ к массиву осуществляется с помощью квадратных скобок:

```
cakes[0]
  => "florentine"
```

Нулевая индексация

Массивы индексируются нулем, поэтому 0 является первым элементом.

Полиморфный

Как и следовало ожидать от Ruby, массивы могут содержать элементы любого типа. Это разрешено:

```
["hello", 1, 0.5, false]
```

Манипулирование массивами

Манипулировать массивами можно огромным разнообразием способов, например:

Добавление

```
[1,2,3] + [4,5,6]
  => [1, 2, 3, 4, 5, 6]
```

Вычитание

```
[1,2,3] - [1,3]
  => [2]
```

Добавление

```
[1,2,3] << 4
  => [1, 2, 3, 4]
```

Умножение

```
[1,2,3] * 2
  => [1, 2, 3, 1, 2, 3]
```

Удобный dandy array API

Мы также можем использовать множество методов создания массивов, таких как:

```
[1,2,3].reverse
  => [3, 2, 1]
```

```
[1,2,3].include? 2
  => true
```

Возможно, вы заметили, что в названии этого метода есть знак вопроса? Это соглашение Ruby. Методы с вопросительным знаком возвращают значение true или false.

```
[4,9,1].sort
  => [1, 4, 9]
```

Разбиение массива

Параллельное присваивание работает при извлечении значений из массива.

```
array_of_numbers = [1,2,3,4]
a,b = array_of_numbers
a
  => 1
b
  => 2
```

Мы также можем извлечь первый элемент и вернуть остальную часть массива, если захотим:

```
arr = [1,2,3]
a,*arr = arr
a
  => 1
arr
  => [2, 3]
```

Это хитрый трюк, который стоит знать, поскольку он производит впечатление на людей и заставляет вас выглядеть умным.

Создание массива с помощью метода to_a

Метод to_a позволяет преобразовать множество объектов в массивы. Например, массив может быть создан из диапазона следующим образом

```
(1..10).to_a
  => [1,2,3,4,5,6,7,8,9,10]
```

Диапазоны

Диапазоны - это полезные объекты, которые можно использовать для представления последовательности. Диапазоны определяются с помощью синтаксиса .. или .... Например:

```
1..10
```

представляет собой последовательность чисел от 1 до 10.

```
1...10
```

представляет диапазон, который исключает высокое значение. В данном случае числа от 1 до 9

Диапазоны в памяти

Диапазоны компактны. Каждое значение в диапазоне не хранится в памяти, поэтому, например, диапазон:

```
1..100000000
```

... занимает тот же объем памяти, что и диапазон:

```
1..2
```

Преобразование в массивы

Диапазоны могут быть преобразованы в массивы с помощью to_a следующим образом

```
(1..10).to_a
  => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Обратите внимание, что здесь нам нужно заключить диапазон в фигурные скобки, чтобы устранить неоднозначность точек.

Массивы символов

Мы также можем объявлять массивы символов следующим образом:

```
'a'..'f'
```

Проверка того, содержит ли диапазон значение.

Мы можем использовать оператор equality ===, чтобы проверить, содержит ли диапазон значение. Например:

```
('a'..'f') === 'e'
  => true

('a'..'f') === 'z'
  => false
```

Упражнение - Перебор массива

Мы можем передать блок в массив, который будет выполняться для каждого элемента в этом массиве. Например:

```
['fork','knife','spoon'].each {|table_item| puts table_item}
```

Нашему коду не нужно знать внутренние детали массива, он просто передает блок и позволяет массиву разобраться самому.

1. вот массив строк. Передайте блок в Array. каждый выводит их один за другим.

```
['cats','hats','mats','caveats']
```

2. Метод each_with_index принимает блок с двумя параметрами, значением и индексом. Используйте его, чтобы распечатать строки в массиве, которым предшествует их индекс в массиве, следующим образом:

```
1. cats
2. hats
3. mats
4. caveats
```

3. Повторите упражнение выше, но теперь распечатайте только каждую строку с индексом нечетного числа.

4. Изучите метод remove. Измените свой код, чтобы выводить только строки, содержащие букву ‘c’

5. Функции, которые ничего не возвращают, сложно протестировать. Измените свой код так, чтобы вместо вывода на экран он возвращал строку. Используйте RSpec для его тестирования.

Упражнение - Присоединиться

Используйте метод join для объединения массива с помощью запятых, у вас должно получиться что-то вроде этого:

```
Fluffy, Hammy, Petunia
```

Манипулирование массивами

Измените порядок следования массива в обратном порядке, выведите что-то вроде этого:

```
Petunia, Hammy, Fluffy
```

Добавьте два массива друг к другу

Создайте массив dogs. добавьте его к первому. используя оператор plus, затем выведите оба массива вместе, вот так:

```
Fluffy
Hammy
Petunia
```

Упражнение - Карта

Используйте Map для заглавной буквы каждого элемента в массиве перед выводом, вот так:

```
FLUFFY
HAMMY
PETUNIA
```

Теперь выделите каждый элемент массива заглавными буквами и измените его местами наоборот.

```
PETUNIA
HAMMY
FLUFFY
```

Упражнение - Сортировка и реверсирование

Используйте sort для сортировки массива в алфавитном порядке следующим образом:

```
BARKY
FLUFFY
GOMEZ
HAMMY
PETUNIA
WOOFY
```

И отсортируйте массив в обратном порядке:

```
WOOFY
PETUNIA
HAMMY
GOMEZ
FLUFFY
BARKY
```

Сортировка массива с помощью блока

По умолчанию array.sort сортирует любой массив значений, используя метод <=> spaceship для значения. Этот метод, как мы видели, возвращает -1, 0 или 1 в зависимости от того, является ли значение меньшим, эквивалентным или больше другого значения. Класс String реализует метод <=>

путем сравнения двух значений в алфавитном порядке, поэтому по умолчанию array.sort сортирует массив строк в алфавитном порядке следующим образом:

```
array = ["James", "Derek", "Stuart", "Thomas"]
puts array.sort
  => Derek
  James
  Stuart
  Thomas
```

Если мы хотим переопределить это, есть два простых способа сделать это. Сначала мы можем переопределить оператор spaceship (скоро появится). В качестве альтернативы мы можем передать Array.sort блок, который можно использовать вместо него. Например, следующий код сортирует массив в порядке второй буквы:

```
array = ["James", "Derek", "Stuart", "Thomas"]
puts array.sort {|a,b| a[1] <=> b[1]}
  => James
  Derek
  Thomas
  Stuart
```

Красиво и просто.

Упражнение - Сортировка по длине строки.

Вы можете получить длину строки“ используя ”string".длина.

```
BARKY
WOOFY
HAMMY
GOMEZ
FLUFFY
PETUNIA
```

Изменение каждого элемента массива с помощью map

Map - это безумно полезная функция массива, которая позволяет изменять каждый элемент массива с помощью блока.

Допустим, у вас есть массив строк:

```
['kittens', 'puppies', 'hamsters']
```

Допустим, вы хотите преобразовать это в массив символов, вы могли бы сделать что-то вроде

```
['kittens', 'puppies', 'hamsters'].map {|i| i.to_sym}
```

Упражнение - Случайные буквенно-цифровые символы с картой

Используйте метод Array #map для генерации случайной строки. Вы можете сгенерировать случайную букву, создав массив допустимых символов следующим образом:

```
n = (0..9).to_a + ('a'..'z').to_a + ('A'..'Z').to_a + %w(_ -)
```

Затем вы можете выбрать случайный вариант следующим образом:

```
n.sample
```

Теперь используйте map для генерации случайной строки из 128 символов.

Хэши и символы

Хэши имеют большое значение в Ruby. Мы часто используем их, особенно при передаче битов данных. Символы - это крошечные легковесные объекты-заполнители Ruby. Они часто используются в сочетании с хэшами. В этом разделе мы рассмотрим хэши, символы и некоторые из их распространенных применений.

Символы - это не магические руны

Новичкам в Ruby символы часто кажутся волшебными рунами. На самом деле это просто небольшие объекты, которые имеют некоторые специальные функции и синтаксис, окружающие их, чтобы сделать их немного проще в использовании и немного меньше в памяти компьютера.

Синтаксис символов

Символы определяются с помощью оператора двоеточия или метода to_sym:

```
:price
:length
:outrageousness
"My Price".to_sym
```

Особенности символов

- Символы - это крошечные объекты
- Символы не имеют значения, они являются заполнителями, а не переменными.

- Всегда может быть только один символ с определенным именем, им управляет Ruby. Из-за этого они не требуют больших затрат процессора или памяти.
- Символы потенциально могут представлять собой утечку памяти, если вы создаете их много.

Может быть только один

Символ будет существовать в памяти только один раз, независимо от того, сколько раз он использовался. Например, если вы создадите два символа в разных местах, оба с именем, например :name, будет создан только один объект. Этот объект будет сохраняться до тех пор, пока работает интерпретатор Ruby.

Использование символов

Символы чаще всего используются в качестве заполнителей в хэшах. Мы уже использовали символы в Rails, например, хэш Rails params связывает любые значения любых параметров, переданных пользователем (из формы или URL-адреса), с символом, представляющим имя этого значения.

Хэши

Хэши - это объекты, которые связывают списки произвольных объектов друг с другом. Например:

```
animals = Hash.new
animals[:tall] = "giraffe"
animals[:minute] = "kitten"
puts animals.inspect
```

Сокращенное обозначение хэш-функции

```
animals = {:tall => "giraffe", :minute => "kitten"}
puts animals[: minute]
=> kitten
```

Установка значений по умолчанию в хэше

Хэш вернет nil, если не сможет найти ни одного подходящего ключа. Вы можете установить значение по умолчанию, которое хэш вернет в этом случае, если вы того пожелаете.

```
animals = Hash.new("monkey")
puts animals[:funny]
=> "monkey"
```

Вы также можете установить это значение, используя метод Hash.default

```
animals.default = "star mole"
puts animals[:odd]=> star mole
```

В качестве ключа можно использовать любые объекты

В качестве ключа можно использовать любой объект. Здесь мы используем число 45 в качестве ключа и сохраняем корневой каталог локальной файловой системы в качестве значения:

```
animals[45] = Dir.new '/'
puts animals.inspect # {45 => #<Dir:0x284a394>
```

Все это разрешено.

Следует иметь в виду одно предостережение: если вы используете Ruby версии 1.8.6 или ниже, вы не можете использовать хэш в качестве ключа в другом хэше. Эта проблема была решена в Ruby версии 1.8.7 и выше.

Упражнение - хэши и символы

Создайте хэш для хранения списка чувств и продуктов питания. Он должен выглядеть примерно так, только длиннее:

```
food_hash = {
  :happy => "ice cream",
  :pensive => "witchetty grub"
}
```

Напишите функцию, которая позволяет пользователю вводить feeling и получать от нее соответствующее значение food (совет: используйте to_sym для преобразования введенных пользователем данных в символ).

Функции, которые получают хэш

Функция может получать хэш значений. Это чрезвычайно полезно, и мы делаем это постоянно.

```
def get_greeting_for(args={})
  name = args[:name] || "anonymous"
  return "hello #\{name}"
end

puts get_greeting_for :name => "Fat Tony"
=> "hello Fat Tony"

puts get_greeting_for
=> "hello anonymous"
```

Здесь наша функция получает параметр, который мы назвали args . Значением параметра args по умолчанию является пустой хэш. Любые пары ключ-значение, которые мы передадим, перейдут в args и

могут быть извлечены.

Во второй строке мы делаем это:

```
name = args[:name] || "anonymous"
```

Здесь мы устанавливаем значение name равным либо значению, хранящемуся в аргументах под ключом :name, либо, если это значение равно нулю (и, следовательно, false), мы устанавливаем его в значение anonymous. Это чрезвычайно полезно, поскольку мы можем создавать функции, которые принимают несколько аргументов в любом порядке с любыми значениями по умолчанию, которые имеют смысл.

Вы должны привыкнуть писать настраиваемые, расширяемые методы, которые получают хэш. Это настоящий rubyism.

Упражнение - получение хэша

Расширьте вашу функцию поиска еды, чтобы она могла получать хэш. Вы должны иметь возможность вызывать ее следующим образом:

```
food mood: :pensive
```

Напишите RSpec

Напишите RSpec, чтобы убедиться, что он действительно работает.

Обзор

- Символы - это одноэлементные строки. Они очень легкие и могут быть использованы везде, где вам нужен уникальный объект, например, в хэше.
- Хэши - это словарные объекты, иногда называемые хэш-таблицами. Они связывают ключи со значениями.
- Вы можете использовать любой объект в качестве ключа и хранить любой объект как значение.
- Мы обычно используем символы в качестве ключей в хэшах
- Существует специальный синтаксис, который позволяет нам передавать хэш функции.

Сложное упражнение - шифр подстановки

Шифр подстановки - это шифр, в котором каждая буква заменяется другой, поэтому "a" может соответствовать "z", а "b" - "y".

Здесь мы создадим шифр подстановки в нескольких словах.

Мы можем использовать zip для объединения двух массивов

```
[1,2,3].zip [4,5,6]
  => [[1,4],[2,5],[3,6]]
```

мы можем составить массив символов из диапазона, подобного этому:

```
('a'..'z').to_a
```

Мы можем создать хэш из массивов массивов, вот так:

```
Hash[[[1,4],[2,5],[3,6]]]
  => {1 => 4, 2 => 5, 3 => 6}
```

Вы можете вращать массив с помощью rotate, вот так

```
[1,2,3].rotate 1
  => [2,3,1]
```

Используйте эти методы для создания хэша шифра подстановки, что-то вроде этого:

```
{'a' => 'b', 'b' => 'c', c => 'd' ...}
```

1. Теперь напишите функцию, которая может принимать строку.
2. Используйте split, чтобы разбить строку на массив.
3. Используйте map с блоком, чтобы переписать каждый элемент, чтобы a стало z и т. Д…
4. Теперь используйте join, чтобы превратить массив обратно в строку.

Если вы все сделали правильно, вы должны быть в состоянии передать строку обратно, и все вернется к тому, как было.

Для получения бонусных баллов напишите тело функции в двух строках кода.

Упражнение - Получение хэша

Измените свой шифр подстановки, чтобы он мог получать хэш значений.

Я хочу иметь возможность называть это следующим образом

```
substitution_cypher "Hello Ruby", rotate: 3
```

Классы и объекты

Ruby объектно-ориентирован. Многие языки утверждают, что они объектно-ориентированные, но большинство из них не дотягивают в той или иной области. Когда Ruby говорит, что он объектно-ориентированный, это действительно так.

До сих пор мы использовали множество объектов Ruby, когда делали такие вещи, как:

```
"Abracadabra".reverse.downcase
```

и

```
Time.now
```

Давайте теперь посмотрим, как мы можем определять наши собственные объекты.

Классы

В Ruby есть объектная модель, основанная на классах. Это означает, что мы определяем классы, а затем используем их для удаления любого количества объектов, сколько захотим. Конечно, поскольку это Ruby, классы являются объектами и имеют собственные методы, но мы вернемся к этому достаточно скоро.

Мы определяем класс следующим образом:

```
class Pet
end
```

Здесь мы определили очень простой класс, называемый Pet . По соглашению классы в Ruby всегда начинаются с заглавной буквы. Теперь давайте создадим pet:

```
flopsy = Pet.new
```

Это здорово, мы создали новый экземпляр класса Pet. Flopsy - это экземпляр Pet.

Обратите внимание, что все объекты получают новый метод бесплатно. Он наследуется от суперкласса Object. Подробнее о наследовании чуть позже.

Условности в Ruby

Поскольку Ruby - такой упрощенный язык, соглашения становятся очень важными. snake_case для переменных, CapitalLetters для классов.

Эти вещи не предусмотрены языком, но они помогут вам написать более последовательный, разборчивый и совместимый код.

Поиск класса объекта

Мы можем пойти наоборот, чтобы найти класс экземпляра, подобного этому

```
flopsy.class
  => Pet
```

Ruby не уклоняется от самоанализа, он встроен, как мы увидим позже.

Методы - предоставление возможностей Flopsy

Все это очень мило, но флопси не очень интересна, она не умеет ходить по канату, или играть в шахматы, или вообще делать что-либо особенное. Чтобы сделать Флопси более интересным, нам нужен метод:

```
class Pet
  def play_chess
    puts "Now playing chess"
  end
end
```

Смотрите сейчас. Мы добавили во flopsy метод play chess. Теперь мы можем писать:

```
flopsy.play_chess
```

... и она это сделает, в некотором роде. В конце концов, она всего лишь домашний питомец.

Соглашения об именовании

Есть несколько вещей, которые следует иметь в виду при присвоении имен методам в Ruby, если вы хотите выглядеть невозмутимым в обществе детей.

Во-первых, используйте регистр snake для всех имен функций, вот так.

```
each_with_index
```

Во-вторых, если ваш метод возвращает логическое значение и является вопросом, сформулируйте его как таковой. Используйте знак вопроса, например:

```
['toast','jam','honey'].include? 'ham'
person.has_name?
password.valid?
```

В-третьих, если ваш метод изменяет исходный объект на месте, а не возвращает новый объект, и поэтому является деструктивным, укажите это восклицательным знаком, вот так:

```
['toast',['jam','honey']].flatten!
  => ['toast','jam','honey']
```

Переменные экземпляра - дают flopsy некоторые сверхспособности

Flopsy все еще немного скучновата. Было бы здорово иметь возможность хранить некоторые данные о ней, возможно, присвоить ей какие-то пользовательские атрибуты.

В Ruby мы сохраняем переменную экземпляра, используя синтаксис @ . Переменные экземпляра - это @ variables . Все переменные экземпляра являются частными, поэтому, чтобы получить к ним доступ, нам нужно написать методы, называемые получателями и установщиками, для доступа к ним. Давайте посмотрим сейчас:

```
class Pet

  def super_powers=(powers)
    @super_powers = powers
  end

  def super_powers
    @super_powers
  end

end
```

Здесь мы предоставили flopsy два метода: getter и setter. Первый - это setter. Метод super_powers= получает параметр и сохраняет его в переменной экземпляра с именем @super_powers.

Второй - это средство получения. Он просто возвращает переменную экземпляра @super_powers, которая была установлена ранее.

Теперь мы можем настроить супермощность flopsy следующим образом:

```
flopsy.super_powers = "Flight"
```

и извлекать его следующим образом:

```
flopsy.super_powers
  => "Flight"
```

Обратите внимание, что нам не нужно нигде объявлять переменную @super_powers. Мы можем просто установить ее, и все в порядке.

Это здорово, потому что...

Геттеры и сеттеры предоставляют нам чистый способ предоставления интерфейса для нашего объекта. Это изолирует нас от деталей реализации. Мы вольны хранить данные любым способом, каким пожелаем, в виде переменной, файла, базы данных, зашифрованного хэша или комбинации других переменных.

Вот как работает active record при использовании Rails. Значения могут быть получены из базы данных, как если бы мы обращались к атрибутам объекта.

Как и в Flopsy, граница между атрибутами и методами гораздо более размытая, чем в большинстве языков. Отчасти это происходит из-за необязательных круглых скобок Ruby, которые создают впечатление, что мы обращаемся к атрибутам, хотя на самом деле мы всегда обращаемся к методам.

Мы можем иметь атрибуты только для чтения, только создав средство получения, и атрибуты только для записи, только создав средство установки. Примером атрибута, доступного только для записи, может быть пароль, который может быть установлен, а затем зашифрован односторонним хэшем и никогда больше не будет прочитан.

Метод attr

Поскольку переменные класса являются настолько распространенными, ruby определяет ярлыки для их создания и связанные с ними методы получения и установки. Метод attr создает атрибут и связанный с ним метод получения. Если второй параметр равен true, то также создается установщик. Методы attr_reader и attr_writer создают получатели и установщики независимо.

Инициализация flopsy с помощью метода initialize.

Метод initialize вызывается новым методом, и именно сюда мы помещаем любой код, необходимый нам для инициализации объекта.

Когда напарница флопси мопси была впервые создана, у нее вообще не было никаких способностей, заметьте:

```
mopsy = Pet.new
mopsy.super_powers
  => nil
```

Бедный мопси. Мы можем исправить эту ситуацию, предоставив мопси базовую сверхспособность при ее инициализации. Давайте сделаем это сейчас.

```
class Pet
  def initialize(args = {})
    @super_powers = args[:power] || "Ability to eat toast really really quickly"
  end
end
```

Теперь, когда мы воссоздаем mopsy, она поставляется уже готовой

```
mopsy = Pet.new
mopsy.super_powers
  => "Ability to eat toast really really quickly"
```

Мы также можем сделать это:

```
mopsy = Pet.new :power => "none worth mentioning"
mopsy.super_powers
  => "none worth mentioning"
```

Если вам нужно просмотреть список всех атрибутов mopsy, вы можете сделать это с помощью inspect следующим образом:

```
mopsy.inspect
  => "#<Pet:0x102f87fe8 @super_powers=\"Ability to eat toast really really quickly\">"
```

Итог

- Функции в Ruby создаются с использованием ключевого слова def (сокращение от define).
- Функции, которые существуют в объекте, обычно называются методами.
- Функции и методы те же, за исключением того, что они принадлежат объекту.
- Объекты создаются из классов с использованием метода .new

Упражнение - Создайте свой собственный fluffster

Все является объектом, поэтому важно, чтобы мы как можно больше практиковались в их создании. В этой серии упражнений мы создадим класс, реализуем некоторые методы, перегрузим некоторые операторы и создадим некоторые виртуальные атрибуты.

Определите свой класс

1. Определите класс либо для смертоносного военного корабля, либо для пушистого животного, в зависимости от вашего сегодняшнего настроения.
2. Добавьте методы, чтобы вы могли задать имя для своего боевого корабля / fluffster.
3. Добавьте методы для получения и установки возраста в миллисекундах.
4. Напишите метод, который извлекает возраст в днях. Вы должны использовать тот же атрибут age, что и в 3, просто измените его перед возвратом.
5. Напишите метод, который устанавливает возраст в днях. Опять же, используйте тот же атрибут age, просто немного поиграйте с ним, прежде чем устанавливать.
6. Напишите геттер, который возвращает возраст в неделях. (возраст в днях / 7)
7. Напишите методы получения и настройки, которые возвращают строку, представляющую стандартное действие, например “мило извиваться” или “взрывать объекты с помощью экстравагантной огневой мощи”.

Упражнение - Получение хэша

Расширьте созданный вами ранее класс warship / flufster, чтобы он мог получать хэш начальных значений.

Напишите метод инициализатора, который принимает хэш и соответствующим образом устанавливает значения по умолчанию. Вы должны уметь вызывать его следующим образом:

```
Fluffster.new age: 2, name: "Floppy"
```

Упражнение - виртуальные атрибуты только для чтения

1. Добавьте атрибуты позиции x и y с геттерами, но без установщиков. Инициализируйте атрибуты позиции равными нулю.
2. Добавьте методы перемещения на север, юг, восток и запад. При вызове этих методов они должны изменять позиции x и y.
3. Добавьте метод distance_from_home, который вычисляет текущее расстояние от дома.
4. Добавить at_home? метод, который возвращает true, если человек дома, т.е. если его координаты x и y равны нулю.

Виртуальные атрибуты особенно полезны для таких вещей, как пароли, где вы на самом деле не хотите нигде хранить пароль или разрешать его извлечение.

Обезьяньи исправления для большей справедливости

Поскольку Ruby является интерпретируемым языком, объекты открыты и могут быть изменены во время выполнения. Классы могут быть повторно открыты в любое время.

Мы можем предоставить mopsy новые методы, даже после того, как она уже создана. Соблюдайте:

```
class Pet
  def play_chess
    puts "now playing chess"
  end
end

class Pet
  def shoot_fire
    puts "activating primary weapon"
  end
end

mospy.shoot_fire
  => activating primary weapon
```

Mopsy все еще может играть в шахматы. Класс Pet был добавлен, а не перезаписан

```
mopsy.play_chess
  => Now playing chess
```

Изменение существующего класса

Как мы упоминали ранее, существующие классы могут быть расширены. Это включает встроенные классы Ruby. Это функция, которая может быть использована как во благо, так и во зло:

Навсегда

```
class String
  def put_times(n)
    for i in (1..n)
      puts self
    end
  end
end

"Marmalade Toast".put_times 5
```

На зло

```
class Fixnum
  def *(num)
    self + num
  end
end

puts 5*4
  => 9
```

Да, Ruby позволяет вам это делать. Будьте осторожны и делайте что-то, и ваш код будет читаться как жидкий солнечный свет.

Обезьянье исправление

Повторное открытие кода таким способом часто называют обезьяньим исправлением. Мы можем изменять или расширять любой существующий класс во время выполнения, даже встроенный в такие классы, как строки и массивы. Это можно использовать с большим эффектом, например, Rails Fixnum date extensions, которые позволяют вводить такие вещи, как:

```
Date.today + 5.days
```

Здесь Fixnum был исправлен с помощью функции, которая позволяет ему работать с датами и временем. Это приятный синтаксис, хотя некоторых людей он раздражает, поскольку, по-видимому, нарушает инкапсуляцию.

Обезьянье исправление - это весело, но используйте его с осторожностью, иначе в итоге у вас получится скрученный беспорядок на полу.

Метапрограммирование

Monkey patching - это первый шаг к метапрограммированию - написанию кода, который записывает код. Подробнее об этом скоро.

Упражнение - секунды

Расширьте Fixnum с помощью метода .seconds, который возвращает число * 1000

Теперь вы можете вызвать Time.now + 60.seconds, чтобы получить время за одну минуту.

Для получения бонусных баллов также создавайте методы минут, часов, дней и недель. Теперь вы можете вызывать Time.now + 1.week .

Упражнение - Зеленые бутылки

Расширьте класс FixNum методом green_bottles, который возвращает текст популярной песни.

Я хочу иметь возможность сказать:

5.green_bottles

и вернуться к:

```
"5 green bottles sitting on the wall\n
4 green bottles ..."
```

Если вы используете Macintosh, сделайте звук погромче и попробуйте это. Обратите внимание на обратные кнопки, которые вы можете найти над клавишей alt:

```
`say #\{5.green_bottles}`
```

Бонус

Для получения бонусных баллов заставьте его принимать блок, который получает песню построчно. Я хочу иметь возможность вызывать:

```
5.green_bottles {|song_line| puts song_line}
```

Exercise - Every Other ()

Extend the Array class with a method that iterates over every other element. Call it like this:

```
names.every_other {|name| puts name}
```

Вам нужно будет явно получить блок, отфильтровать массив, затем передать его методу each.

Перегрузка оператора

Я упоминал, что в Ruby все является объектом? Это распространяется на операторы, такие как +, -, * и / . Операторы в Ruby на самом деле являются методами, и мы можем определять и переопределять

их, вот так:

```
class Pet
  def +(pet)
    p = Pet.new
    p.super_powers = self.super_powers + " and also " + pet.super_powers
    return p
  end
end
```

Здесь мы определили метод plus, который получает другого питомца. Это просто создает нового питомца с объединенными сверхспособностями двух его родителей и возвращает его. Понаблюдайте за потомством Мопси и Флопси:

```
cottontail = mopsy + flopsy
cottontail.super_powers
  => "Ability to hop really really quickly and also Flight"
```

Упражнение по переопределению оператора +

В этом разделе вы расширите свой класс warship / flufster с некоторой перегрузкой оператора.

Реализуйте оператор +. Пусть он вернет новый объект с объединенными именами и стандартными действиями.

Сортировка вашего пользовательского класса с помощью <, > и <=> (космический корабль).

На этот раз немного более полезное.

1. Определите метод в вашем классе, который определяет мощность атаки.
2. Теперь определите оператор <, который возвращает true, если объект имеет меньшую мощность атаки.
3. Следуйте инструкциям операторов > и ==.

Теперь о космическом корабле

1. Реализуйте оператор spaceship в своем классе. Оператор spaceship возвращает либо -1, 0, либо 1 в зависимости от того, меньше, совпадает или больше второго первого операнда.
2. Теперь создайте массив объектов на основе вашего класса и отсортируйте их.
3. Передайте массив.отсортируйте блок и отсортируйте массив на основе разных критериев.

Теперь вы можете вызвать метод .sort для массива объектов, и он будет отсортирован по мощности атаки.

Дальнейшее упражнение (если вы закончите первым)

Реализуйте оператор *.

Пусть он возвращает массив, содержащий несколько экземпляров объекта. (либо копии, либо несколько переменных, указывающих на один и тот же объект, свободный выбор.) Вы можете

дублировать объект, используя метод .dup .

Наследование

Rails поддерживает наследование одного объекта. Это означает, что класс может иметь один родительский класс и будет наследовать все методы и атрибуты, принадлежащие этому классу. Мы определяем отношения наследования с помощью оператора <.

Допустим, мы хотели бы определить определенный тип домашнего животного, скажем, маленького и пушистого котенка. Давайте создадим класс kitten, который может наследоваться от нашего класса Pet:

```
class Kitten < Pet
  def play_tennis
    puts "I am now playing tennis"
  end
end
```

Теперь у нашего котенка есть все атрибуты домашнего животного. Он может стрелять огнем из своих глаз и неплохо играть в шахматы, но вдобавок он также может играть в теннис:

```
tiger = Kitten.new
tiger.play_tennis
  => I am now playing tennis
tiger.shoot_fire
  => now shooting fire
```

Упражнение - Подклассы

Расширьте свое предыдущее упражнение "Флаффстер / военный корабль". Создайте подкласс военного корабля, возможно, фрегата, с другими способностями.

Собственные классы (статические методы)

В Ruby все методы существуют внутри класса. Когда вы создаете объект, методы для этого объекта существуют внутри его класса. Методы могут быть общедоступными, частными или защищенными, но понятия **статического метода** не существует. Вместо этого у нас есть одноэлементные классы, обычно называемые собственными классами.

Никаких статических методов

Статические методы - это методы класса. они принадлежат классу, а не экземпляру. Это нарушило бы простую объектную структуру Ruby, поскольку классы являются экземплярами class Class, добавление методов в Class сделало бы их доступными повсюду, а это не то, чего мы хотим.

Синглтоны

Вместо этого Ruby позволяет нам определять безымянный одноэлементный класс, который находится в дереве наследования непосредственно над любым объектом. Давайте сделаем это сейчас и создадим статический метод.

```
class Kitten
  class << Kitten
    def max_size
      8
    end
  end
end
```

Синтаксис класса << Kitten открывает собственный класс и помещает в него метод max_size. Затем мы можем получить к нему доступ следующим образом

```
puts Kitten.max_size
```

Обратите внимание, что здесь мы говорим о классе Kitten как об объекте.

Сокращение

Мы используем синтаксис class << self для явного открытия собственного класса объекта. Мы можем выполнить то же самое, используя сокращенный синтаксис:

```
def Kitten.max_size
  8
end
```

Это добавляет метод к собственному классу Kitten. Мы также можем добавить метод к собственному классу любого другого объекта, например:

```
fluffy = Kitten.new
popsy = Kitten.new

def popsy.deploy_wheels
  @wheels = :deployed
end

def popsy.launch_scouter
```

```
    @scouter = :launched
  end
```

Здесь мы добавили метод в собственный класс popsy, позволяющий ей развертывать диски.

Собственные классы в иерархии наследования

Собственный класс находится непосредственно над объектом в иерархии наследования, под классом объекта. Это удобное место для размещения методов, которые мы хотим применить непосредственно к объекту, а не к каждому экземпляру этого объекта. Это кажется техническим, но как только вы его получите, на самом деле это довольно приятно.

Упражнение - Добавление метода непосредственно к объекту

В irb (или в файле ruby) создайте 3 экземпляра вашего класса warship / pet. Добавьте к каждому из них свой метод. Убедитесь, что его может вызывать только экземпляр, к которому вы добавили метод.

Вы пишете в собственный класс. Кажется естественным, не так ли?

Модули (миксины)

Модули могут использоваться для добавления функциональности многократного использования в класс. Иногда их называют миксинами. Модуль состоит из целого набора методов. Импортируя его в класс, мы получаем доступ ко всем этим методам. Это удобный способ обойти ограничения наследования одного объекта, поскольку мы можем импортировать столько модулей, сколько захотим.

Определение общей функциональности

Давайте научим флопси готовить омлет. Тогда она сможет помогать на кухне. Было бы неплохо, если бы наш омлет допускал несколько вариантов, поэтому давайте разрешим и это.

```
module CookOmelette
  def cook_omelette(args={})
    number_of_eggs = args[:number_of_eggs] || args[:eggs] || 2
    cheese = args[:cheese] ? "cheese" : nil
    ham = args[:ham] ? "ham" : nil
    mushrooms = args[:mushrooms] ? "mushrooms" : nil
    ingredients = [cheese,ham,mushrooms].delete_if{ |ingredient| ingredient
    ingredients = ingredients.join(" & ")
    "#\{ ingredients } omelette with #\{number_of_eggs} eggs".strip
  end
end
```

Теперь включите mixin в класс Pet.

```
class Pet
  include CookOmelette
end
```

Теперь все наши питомцы могут готовить вкусные омлеты. Соблюдайте:

```
mopsy.cook_omelette
  => "omelette with 2 eggs"
mopsy.cook_omelette :ham => true, :cheese => true, :eggs => 4
  => "cheese & ham omelette with 4 eggs"
```

Наследование модулей

Включение миксина в класс добавляет эти методы в этот класс, как если бы они были определены внутри этого класса.

Методы, добавленные в класс модулем, наследуются подклассами этого класса. Например, при включении CookOmelette mixin в класс Pet подкласс Kitten и все его экземпляры также получают этот метод.

Упражнение - Извлечение общей функциональности

Это упражнение расширяет возможности смертоносного боевого корабля класса "пушистый котенок" .

1. Теперь давайте создадим модуль. Извлеките методы age и age_in_weeks в модуль, который можно включить в другое место. Назовите модуль разумно. Теперь удалите их из вашего класса и вместо этого импортируйте модуль. Теперь у вас есть возможность указать возраст чего угодно и разумно запрашивать его возраст.
2. Напишите стереомодуль, который позволит вашему классу воспроизводить несколько классных случайных звуков (на самом деле струнных). Добавьте его в свой класс.

Расширять вместо Включать

Include и Extend позволяют нам брать методы из модуля и добавлять их в объект. Хотя они работают немного иначе друг от друга. Давайте посмотрим…

Extend добавляет методы непосредственно к объекту

Extend добавляет методы к объекту. Он расширяет этот объект, добавляя к нему новые функции.

```
class Hamster
end

module PetSkills
  def snuggle;end
end

Hamster.extend PetSkills
```

Если вы расширяете класс, вы создаете метод класса.

```
h = Hamster.new
Hamster.methods.include? :snuggle
# => true
h.methods.include? :snuggle
# => false
```

Если вы расширяете экземпляр класса, вы создаете метод экземпляра, но только для этого экземпляра. Вы можете расширить любой объект подобным образом.

```
h.extend PetSkills
h.methods.include? :snuggle
# => true

i = Hamster.new;
i.methods.include? :snuggle
# => false
```

Вы можете вызвать extend для любого объекта, чтобы добавить методы только к этому объекту.

Include добавляет методы экземпляра в класс

Include использует более традиционный подход. Если вы включаете модуль в класс, методы в модуле будут добавлены как методы экземпляра и будут доступны для всех экземпляров этого класса.

```
class Gerbil
  include PetSkills
end

g = Gerbil.new
Gerbil.methods.include? :snuggle
# => false
g.methods.include? :snuggle
# => true
```

Итог

Extend добавит методы к объекту, и только к этому объекту. Если мы расширяем класс, мы получаем методы класса.

Include будет включать методы из модуля в класс, эти методы становятся методами экземпляра для объектов этого типа.

Обработка исключений

Обработка исключений в Ruby очень похожа на другие языки.

Создание исключения

Создать исключение в Ruby тривиально просто. Мы используем raise.

```
raise "A Error Occurred"
```

Это вызовет исключение RuntimeException по умолчанию.

Создание определенного исключения

Мы также можем создавать исключения определенного типа:

```
value = "Hi there"
raise TypeError, 'Expected a Fixnum' if value.class != Fixnum
```

Спасение исключений

Мы можем легко восстанавливать исключения. Поместите код, который может вызвать исключение, в блок begin, rescue end. В случае возникновения исключения управление будет передано в раздел rescue.

```
begin
  raise "A problem occurred"
rescue => e
  puts "Something bad happened"
  puts e.message
end
```

Сохранение определенных исключений

Мы можем сохранять различные типы исключений

```
value = "Hi there"

begin
  raise TypeError, 'Expected a Fixnum' if value.class != Fixnum
```

```
    raise "A problem occurred"
  rescue TypeError => e
    puts "A Type Error Occurred"
    puts e.message
  rescue => e
    puts "an unspecified error occurred"
  end
```

Иерархия исключений Ruby

Вот встроенные исключения, доступные в Ruby:

```
Exception;
NoMemoryError;
ScriptError;
LoadError;
NotImplementedError;
SyntaxError;
SignalException;
Interrupt;
StandardError;
ArgumentError;
IOError;
EOFError;
IndexError;
LocalJumpError;
NameError;
NoMethodError;
RangeError;
FloatDomainError;
RegexpError;
RuntimeError;
SecurityError;
SystemCallError;
SystemStackError;
ThreadError;
TypeError;
ZeroDivisionError;
SystemExit;
fatal;
```

Определение собственного исключения

Вы можете определить свои собственные исключения следующим образом:

```
class MyNewError < StandardError
end
```

Затем вы можете создать свое новое исключение по своему усмотрению.

Упражнения на исключение

Попробуйте эти упражнения, чтобы получить представление об обработке исключений в Ruby.

Вывод ошибки аргумента

Расширьте свой класс kitten со вчерашнего дня. Давайте предположим, что вашему котенку нужен возраст (0 не подходит), Вызовет ошибку аргумента, если возраст не задан в инициализаторе

Выдает ошибку типа

Ваш возраст котят должен быть фиксированным числом. Проверьте это, если это не так, выдайте ошибку типа

Ошибка при делении на ноль

Может ли ваш котенок заниматься математикой? Если нет, напишите сейчас функцию divide, которая принимает два значения и делит их. Поймайте ошибку деления на ноль и, если она возникнет, верните nil.

Написание методов, которые принимают блоки

Итак, как метод передает управление блоку? Ну, мы используем ключевое слово yield . Например, следующая (очень простая и не очень полезная) функция принимает блок кода, а затем просто запускает его один раз:

```
def do_once
  yield
end

do_once {puts "hello"}
  => "hello"
do_once {10+10}
  => 20
```

Вы можете вызывать yield столько раз, сколько захотите. Вот как работает метод Array.each, перебирая массив и вызывая yield для каждого элемента, передавая элемент в качестве параметра. Вот глупый пример:

```
def do_thrice
  yield
  yield
  yield
end
```

```
do_thrice {puts "hello"}
  => "hello hello hello"

do_thrice {puts "hello".upcase}
  => "HELLO HELLO HELLO"
```

Передача параметров в блок

Блок - это безымянная функция, и вы можете легко передать ей параметры. Следующий пример принимает массив имен и для каждого из них отправляет блоку приветствие. Блок в этом случае просто выводит приветствие на экран.

```
def greet(names)
  for name in names
    yield("Hi There #\{name}!")
  end
end

greet(["suzie","james","martha"]) { |greeting| puts greeting }
  => Hi There suzie!
  Hi There james!
  Hi There martha!
```

Итог

Мы можем сообщить нашей функции о получении блока двумя способами:

1. Явно, простым получением параметра блока
2. Неявно, вызывая yield внутри нашего блока

Мы можем проверить, был ли передан блок, используя метод block_given? .

Повторная реализация упражнений по методу Fixnum#times

Это вызывает блок заданное количество раз. Это позволяет вам писать код, подобный этому:

```
12.times_over { puts "starfish" }
```

Recreate this method in your own words.

Расширьте его, чтобы вы могли называть его следующим образом:

```
12.times_over_with_index { |i| puts "\#{i} - starfish" }
```

Повторно реализуйте метод Array#each_with_index.

Этот метод вызывает свой блок один раз для каждого элемента в массиве. Вы вызываете его следующим образом:

```
[1,2,'cats'].each_with_index { |el, i| puts "\#{i} - \#{el}" }
```

Дальнейшее упражнение - Реальный блочный код

Ознакомьтесь с методом Rails link_to. Как и многие вспомогательные программы, он получает необязательный блок.

Ознакомьтесь с базой кода и посмотрите, сможете ли вы понять, как это делается:

https://github.com/rails/rails/blob/433ad334fa46623f4b218d3bb34e3f63d5481c18/actionview/lib/action_view/hel

Создание драгоценного камня

Gem - это zip-файл, содержащий код, как правило, хотя и не всегда Ruby-код, плюс немного метаданных, помогающих RubyGems и Bundler удобно управлять им. Драгоценные камни содержат файл манифеста gemspec, который содержит метаданные, и обычно каталог lib, содержащий сам код.

Поскольку gem содержит код Ruby, он может делать все, что вам нравится. Он может исправлять Rails. Он может удалять новые модули и классы. Он может расширять существующие объекты и т.д.

Gem может храниться локально или на хостинге gem, например rubygems.org. Если вы опубликуете свой gem на rubygems, он будет доступен для скачивания и использования любому пользователю. Если вы сохраните его в своем проекте, вы все равно можете использовать bundler, вы просто указываете путь.

http://asciicasts.com/episodes/245-new-gem-with-bundler

В этом разделе мы собираемся создать простой gem с помощью Bundler.

Создание gem с помощью Bundler

Мы собираемся создать гем summarise, который расширит класс string методами для создания summarize и проверки того, можно ли суммировать строку, вот так:

```
class String
  def summarise(l=200)
    i = 0
    self.split.map{ |word| word if (i += word.length) < l}.compact.join(' ')
  end

  def summarisable?(length=200)
    return self.summarise(length) != self
  end
end
```

прежде всего создайте gem:

```
bundle gem summarise
```

Теперь у нас есть новый каталог summary, содержащий каталог lib для кода, файл gemspec для метаданных и несколько других файлов.

У нас также есть пустой репозиторий git, инициализированный для нас.

Gemspec

Файл gemspec - это сердце вашего драгоценного камня, он содержит все метаданные о вашем драгоценном камне. Сгенерированный вами gemspec будет выглядеть примерно так:

```
# coding: utf-8
lib = File.expand_path('../lib', __FILE__)
$LOAD_PATH.unshift(lib) unless $LOAD_PATH.include?(lib)
require 'summarise/version'

Gem::Specification.new do |spec|
  spec.name          = "summarise"
  spec.version       = Summarise::VERSION
  spec.authors       = ["Nicholas Johnson"]
  spec.email         = ["email@domain.com"]
  spec.description   = %q{TODO: Write a gem description}
  spec.summary       = %q{TODO: Write a gem summary}
  spec.homepage      = ""
  spec.license       = "MIT"

  spec.files         = `git ls-files`.split($/)
  spec.executables   = spec.files.grep(%r{^bin/}) { |f| File.basename(f) }
  spec.test_files    = spec.files.grep(%r{^(test|spec|features)/})
  spec.require_paths = ["lib"]

  spec.add_development_dependency "bundler", "~> 1.3"
  spec.add_development_dependency "rake"
end
```

Обратите внимание, как задается spec.files, получая файлы, включенные в данный момент в git.

Также обратите внимание, что версия берется из константы Summary::VERSION. Это определено в lib/summarise/version.rb. Версию можно обновить, отредактировав этот файл.

Содержимое Gem

Теперь мы определяем gem в каталоге lib.

библиотека/sumarise.rb

Этот файл просто импортирует остальную часть кода.

```
require "summarise/version"
require "summarise/string_extensions"
require "summarise/string"

module Summarise
end
```

библиотека/summarise/string_extensions.rb

Здесь объявляются методы, которые мы хотим добавить.

```
module Summarise
  module StringExtensions
    def summarise(l=200)
      i = 0
      self.split.map{ |word| word if (i += word.length) < l}.compact.join(' ')
    end

    def summarisable?(length=200)
      return self.summarise(length) != self
    end
  end
end
```

библиотека/summarise/string.rb

Это расширяет класс String.

```
class String
  include Summarise::StringExtensions
end
```

Создание gem

Поскольку gemspec использует Git для определения того, какие файлы включать, мы должны сначала зафиксировать ваши обновленные файлы.

```
git add .
git commit -a -m "first commit"
```

Создайте gem с помощью команды gem build:

```
gem build summarise.gemspec
```

Вы создадите файл с названием что-то вроде: summarise-0.0.1.gem

Локальные частные жемчужины

Если вы хотите сохранить свой gem приватным, вы можете развернуть его непосредственно в своем каталоге vendor / gems, например:

Сначала распакуйте его в vendor / gems:

```
gem unpack summarise-0.0.1.gem --target /path_to_rails_app/vendor/gems/.
```

Теперь объявите это в вашем Gemfile:

```
gem 'summarise', :path => "#\{File.expand_path(__FILE__)}/../vendor/gems/summarise-0.0.1"
```

Наконец установите его в Gemfile.lock

```
bundle install
```

Это хороший способ разработки gem, поскольку вы можете развернуть его локально и работать над ним на месте.

Загрузка вашего драгоценного камня в RubyGems.org

Если вы хотите поделиться своим гемом с сообществом, вы также можете отправить его по адресу RubyGems.org

Вам понадобится учетная запись RubyGems:

https://rubygems.org/users/new

Теперь просто нажмите на упакованный gem:

```
gem push summarise-0.0.1.gem
```

Упражнение - создание драгоценного камня

Создайте случайную строку gem. Я хочу иметь возможность вызывать что-то вроде:

String.random(6)

чтобы получить обратно случайную буквенно-цифровую строку.

Отправить

Send позволяет вызывать метод, обозначенный символом.

В моем приложении есть классы для людей и событий. Каждый класс имеет разные атрибуты.

```
class Person
  attr_accessor :name
end

class Event
  attr_accessor :title
end
```

У меня также есть модуль, который настраивает объекты.

```
module Setup
  def init(name_or_title)

  end
end
```

Упражнение - Пошлите немного любви

Включите модуль Setup в классы Person и Item. Теперь напишите метод init. используйте response_to? и отправьте для инициализации имени или заголовка.

Начните с массива [:name,:title], затем выполните итерацию по нему, проверяя, реагирует ли объект на методы, затем вызовите метод, когда получите результат.

Определение метода

Мы используем метод define для создания метода "на лету", задающего строку в качестве имени.

```
class A
  define_method :c do
    puts "Hey!"
  end
end

A.new.b

A.new.c(1,2)
```

Упражнение - Зеленые хомячки

Хомяки бывают красного, зеленого, синего и оранжевого цветов.

Класс hamster выглядит следующим образом:

```
class Hamster
  attr_accessor :colour
end
```

пока мы просто предположим, что разрешены только эти цвета.

Учитывая экземпляр hamster, я хотел бы иметь возможность вызывать что-то вроде:

```
hammy = Hamster.new
hammy.colour = :red
hammy.is_red?
```

и заставить его возвращать true или false

Бонусные баллы

Вместо того, чтобы заранее определять ограниченный набор методов, используйте method_missing, чтобы перехватывать, когда метод не определен, а затем определять его на лету. Если я вызову is_taupe ? method_missing должен уловить это, определить метод Hanster #is_taupe? затем вызовите его.

Упражнение - сушка кода

Приведенный ниже реальный код полон дубликатов. Как вы могли бы использовать define_method, чтобы очистить его?

```
class Widget
  def product
    product = Product.find_by_slug(object_slug)
    if !product
      product = Product.first
    end
    product
  end

  def poem
    poem = Poem.find_by_slug(object_slug)
    if !poem
      poem = Poem.first
    end
    poem
  end
end
```

Если вы не хотите создавать целый экземпляр Rails, вы можете использовать простой каркас, подобный этому:

```
class Product
  class << self
```

```
    def find_by_slug
      return "success"
    end
  end
end
class Poem < Product; end
```

Отсутствует метод

Попытка вызвать метод, которого не существует в Ruby, является скорее исключением, чем языковой ошибкой. Мы можем перехватить это, или мы можем просто реализовать функцию method_missing, которая будет вызвана, когда не будет найден соответствующий метод.

```
class A

  def ary
    [:a,:b,:c]
  end

  def method_missing(method, *args)
    puts ary.include?(method)
  end
end

a = A.new

a.b
a.d
```

Упражнение - Создайте космический корабль.

Предполагая, что вы построили мощный военный корабль, дайте вашему классу набор таких способностей, как эта: [: warp,:photon_torpedos,: holodeck] и т.д.

Теперь мы собираемся использовать method_missing, который позволит нам запрашивать наш космический корабль. Мы сможем вызывать такие методы, как enterprise.has_holodeck? и voyager.can_warp? и получить обратно значение true или false.

Вот регулярное выражение, которое должно помочь

```
FEATURE_REGEX = /^(?:has|can)_(\w*)\?$/
if find = method.to_s.match(FEATURE_REGEX)
  feature = find[1]
end
```

Пример из реального мира

Для примера из реального мира прочитайте и поймите кодовую базу string_enquirer здесь:

https://github.com/rails/rails/blob/d71d5ba71fadf4219c466c0332f78f6e325bcc6c/activesupport/lib/active_suppo

Instance_Eval

Мы используем eval экземпляра для оценки блока в контексте объекта Ruby.

```
class A
  def initialize
    @b = 123
  end
end
puts A.new.instance_eval { puts @b }
```

Мы можем использовать это для создания DSL (языка, специфичного для домена, например:)

```
class Review
  attr_accessor :stars, :title, :content
  def initialize &block
    self.stars = 0
    instance_eval &block
  end

  def set_title t
    self.title = t
  end
end

r = Review.new do |review|
  set_title "new Macbook"
end
```

Упражнение - Оценка экземпляра

Используйте instance_eval для определения DSL для создания веб-страниц.

```
Page.new do
  set_title "My page"
  set_content "Page Content"
end
```

Расширение

Расширьте свой DSL, чтобы вы также могли создавать подстраницы, например:

```
Page.new do
  set_title "My homepage"
  set_content "Page Content"
  sub_page do
    set_title "My sub page"
    set_content "Page Content"
  end
end
```