Стандартные классы: числа, строки, перечисления, регулярные выражения. Набросок

Андрей Васильев

2017

Зачем нужны стандартные типы

Ruby - объектно-ориентированный язык программирования, в котором даже стандартные объекты предоставляют большой набор методов для решения повседневных задач

- Числа
- Строки
- Перечисления

Числа, обзор

Ruby поддерживает следующие виды чисел:

- Целые числа
- Вещественные числа
- Рациональные числа
- Комплексные цисла

Целые числа

Интерпретатор Ruby поддерживает работу с числами бесконечного размера.

- Работа с числами, входящими в размерность платформы, ведётся на уровне встроенных чиселок
- При выходе за эти размеры числа автоматически преобразуются к бесконечным
- В версии Ruby 2.4 вы не можете программно определить какая реализация используется, в предыдущих версиях было 2 отдельных класса Fixnum и Bignum

```ruby
num = 9000
6.times do
  puts "#{num.class}: #{num}"
  num *= num
end
```

Рациональные и комплексные числа

Вещественные числа плохо подходят для работы с бесконечными дробями, например с 1/3

- Для создания рациональных чисел используйте метод `Kernel#Rational`:

```
Rational(3, 5) * Rational(4, 3)
Rational("1/2") * Rational("3/5")
```

- А также с помощью метода `String#to_r`

Комплексные числа

Данные числа полезны при работе с математикой. Создавать их также можно:

- С помощью метода `Kernel#Complex`:

```
Complex(1, 2) * Complex(3, 4)
Complex("1+2i") * Complex("3+6i")
```

- С помощью метода `String#to_c`

Взаимодействие чисел

При выполнении математических операций над числами в Ruby реализованы два принципа:

- Если взаимодействующие числа относятся к одному типу (например, целые), то и результат операции тоже будет относиться к этому типу
- Если они принадлежат разным типам, то результат будет принадлежать классу большей общности: Целые < Рациональные < Вещественные < Комплексные. Вещественные “больше” рациональных, т.к. не существует надёжных методов для такой трансформации.
- Эта информация также относится и к операции деления.
- Библиотека `mathn` немного изменяет поведение стандартных типов и операций, чтобы больше соответствовать представлениям “математиков” о способах описания чисел.

Описание циклов с помощью чисел

Числа также предоставляют удобные итераторы для выполнения операций определённое количество раз

- `Integer#times` позволяет выполнить операцию несколько раз, `3.times {...}`
- `Integer#upto(limit)` позволяет пройти по промежутку, включая его окончание `6.upto(13) {...}`
- `Integer#downto(limit)` позволяет пройти по промежутку вниз, включая его окончание `50.downto(48) {...}`
- `Numeric#step(limit=nil, step=1)` позволяет пройти по промежутку используя конкретный шаг `30.step(50, 5) {...}`
- Эти методы также возвращают нумераторы

Строки

Строки в Ruby являются последовательностями символов, хотя отдельного типа «символ» в Ruby не существует. Для созадния строк можно использовать класс String и литералы

- Литералы с одной кавычкой не обрабатываются
  - \' - заменяется на одинарную кавычку
  - \\ - заменяется на обратную косую
- Литералы из двойных кавычек являются шаблонами
  - В шаблонах можно указывать `#{expr}`
- Литералы обычные: `%q{this is also string}`
- Литералы шаблонные
  - `%Q!Template string!`
  - `%{Template string}`
  - `%/Template string/`

Многострочные литералы

Достаточно часто необходимо подготовить шаблоны

- Краткое описание работы приложения
- Шаблоны различных визуальных элементов
- Шаблоны SQL-запросов

Для создания таких строк можно воспользоваться:

- Конкатенацией однострочных литералов
- Экранированием окончания строки
- Использованием %( )-синтаксиса
- Использование HERE-документов

Here-документы

Обычный документ сохраняет пробелы в начале строк

```
string = <<-END
  This is the test body
  going on multiple strings
END
```

В Ruby 2.3 появилась возможность игнорировать пробелы

```
long_string = <<~DOC
  All the spaces will be removed
  from this string
DOC
```

Данные литералы являются шаблонными

Кодировка строк

Для каждой строки определена кодировка. Кодировка определяет соответствие между битами и символами

Кодировка определяется следующими факторами:

- Версей Ruby. Начиная с 2.0 по умолчанию используется Unicode
- Указанием кодировки в файле: `#encoding: utf-8`
- Указанием кодировки при запуске Ruby

Для проверки кодировки строки используйте атрибут `String#encoding`

Для преобразования строки в другую кодировку используйте метод `String#encode`

Методы класса String

Мы не сможем рассмотреть все методы данного класса, так как их очень много, выделим следующие:

- `#chomp` - убрать символы окончания строки
- `#strip` - убрать пробелы с начала и с конца строки
- `#downcase`, `#upcase` - верхний регистр, нижний регистр
- `#squeeze(str)` - удалить все повторения символов
- `#split(reg)` - разделить строку по регулярному выражению
- `#scan(reg)` - найти подстроки, удовлетворяющие регулярному выражению, и сформировать из них массив
- `#sub(reg, str)` - заменить все подстроки, соответствующие регулярному выражению, на строку

Пример решения задачи

```
/jazz/j00132.mp3 | 3:45 | Fats Waller | Ain't Mis
/jazz/j00319.mp3 | 2:58 | Louis Armstrong | Wonde
/bgrass/bg0732.mp3| 4:09 | Strength in Numbers |
```

Диапазоны

Последовательности достаточно часто встречаются при разработке, например от января к декабрю. В Ruby есть специальный тип Range, диапазон, который используется для:

- Описания последовательности значений.
- Описания условий.
- Описания интервалов.

Диапазоны как последовательности

Любой диапазон описывается двумя значениями: начальным и конечным. Две точки между ними (..) включает конечное значение, три точки (...) исключают:

```
1..10
'a'..'z'
5...15
'A'...'Z'
```

Последовательности можно преобразовывать к другим типам:

- `to_a` - преобразование в массив
- `to_enum` - преобразование в нумератор

Итераторы диапазонов

Вместо преобразования зачастую лучше воспользоваться предоставляемыми итераторами, а также методами модуля Enumerable, который подключён к данному классу

- `bsearch {}` - бинарный поиск по диапазону
- `each {}` - проход по элементам диапазона
- `max {}` - нахождение максимального элемента
- `min {}` - нахождение минимального элемента
- `step(n=1) {}` - пройтись по диапазону с шагом

А также ряд полезных методов:

- `begin`, `end` - начало и конец диапазона
- `last(n)`, `first(n)` - последние и первые элементы
- `include?(obj)` - входит ли объект в диапазон
- `exclude_end?` - входит ли последний объект в диапазон

Собственные объекты в диапазоне

Вы можете использовать произвольные объекты в диапазоне. Для этого необходимо, чтобы объект реализовывал метод сравнения <=>, а также метод для создания следующего элемента последовательности succ.

```
class PowerOfTwo
  attr_reader :value
  def initialize(value)
    @value = value
  end
  def <=>(other)
    @value <=> other.value
  end
  def succ
    PowerOfTwo.new(@value + @value)
  end
end
```

Диапазоны как условия

Диапазоны могут быть использованы в условных операторах, при этом условие будет верным, если начало и окончание промежутка будут верными.

Пример ниже печатает все строки, которые начинаются на слово `start` и заканчивают словом `end`.

```
while line = gets
  puts line if line =~ /start/ .. line =~ /end/
end
```

Нагляднее было бы использование логического И

Диапазоны как интервалы

Достаточно часто диапазоны используются для проверки того: принадлежит ли объект данному диапазону или нет. Для этого используется оператор ===:

```
(1..10) === 5 # => true
(1..10) === 15 # => false
(1..10) === 3.14 # => true
('a'..'z') === 'e' # => true
('a'..'e') === 'z' # => false
```

Case-выражения

Диапазоны зачастую выступают в роли условий Case-выражений

```
car_age = gets.to_f # let's assume it's 9.5
case car_age
when 0...1
  puts "Mmm.. new car smell"
when 1...3
  puts "Nice and new"
when 3...10
  puts "Reliable but slightly dinged"
when 10...30
  puts "Clunker"
else
  puts "Vintage gem"
end
```

Ещё чуть-чуть про Case-выражения

- В примере использовались исключающие промежутки - это упрощает чтение кода
- Это также позволяет корректно обрабатывать ситуацию с нецелыми числами, которые проверяются в выражении. Например, 2.5 корректно будет обработано.

Регулярные выражения

Регулярные выражения представляют собой очень мощный инструмент по работе с текстом. Если вы знаете их внутреннее устройство, то найдёте им замечательное применение в задачах обработки текста. В противном случае вам следует их изучить перед применением.

Регулярные выражения хорошо решают следующие задачи:

- Проверить, что строка соответствует указанному шаблону
- Извлечь из строки подстроки, которые соответствуют частям регулярного выражения
- Изменить строку, заменив подстроки, соответствующие регулярному выражению

Описание регулярных выражений

Обычно регулярные выражения описываются с помощью литералов

- `/cat/` соотносится со строками `"dog and cat"`, `"catch the fish"`, но не со строками `"Cat"` или `"c.a.t"`
- `/t a b/` соотносится со строкой `"hit a ball"`, но не со строкой `"table"`

Есть ряд символов, которые являются специальными и которые необходимо экранировать обратной косой, если вы хотите сравнить свою строку с этими символами: `.`, `|`, `(`, `)`, `[`, `]`, `{`, `}`, `+`, `\`, `^`, `$`, `*`, `?`. Примеры употребления:

- `/\*/` - нахождение звёздочки
- `/\\/` - нахождение обратной косой черты

Литералы регулярных выражений являются шаблонными строками

Сравнение строк с регулярными выражениями

Для сравнения строк с регулярными выражениями используется оператор =~. Он возвращает число, соответствующее совпадению или `nil`, если совпадений нет.

```
/cat/ =~ "dog and cat" # => 8
/cat/ =~ "catch" # => 0
/cat/ =~ "Cat" # => nil
```

Данное сравнение можно использовать в условных операторах

```
str = "cat and dog"
if str =~ /cat/
  puts "There's a cat here somewhere"
end
```

Сравнение строк, продолжение

Есть оператор, который выполняет обратную задачу - проверяет, что строка не совпала с выражением: !~

```
File.foreach("testfile").with_index do |line, in|
  puts " #{in}:  #{line} " if line !~ /on/
end
```

Операторы =~ и !~ позволяют указывать регулярное выражение как слева от оператора, так и справа от него.

Замена подстрок

Метод `String#sub` позволяет заменить первую встречу регулярного выражения на строку

```
str = "Dog and Cat"
new_str = str.sub(/Cat/, "Gerbil")
new_str # => "Dog and Gerbil"
```

Метод `String#gsub` позволяет заменить все встречи регулярного выражения на строку

```
str = "Dog and Cat"
new_str1 = str.sub(/a/, "*" )
new_str2 = str.gsub(/a/, "*" )
new_str1 # => Dog *nd Cat
new_str2 # => Dog *nd C*t
```

Сравнение с классами символов

Базовые классы символов, знакомые из других языков программирования, сравнивают только лишь символы из набора ASCII.

- `/./` - Любой символ за исключением новой строки
- `/./m` - Любой символ, многострочный
- `/\w/` - Символ слова (`[a-zA-Z0-9_]`)
- `/\W/` - Не символ слова (`[^a-zA-Z0-9_]`)
- `/\d/` - Числовой символ (`[0-9]`)
- `/\D/` - Не числовой символ `[^0-9]`)
- `/\h/` - Хекс-число (`[0-9a-fA-F]`)
- `/\H/` - Не хекс-число (`[^0-9a-fA-F]`)
- `/\s/` - Символ-промежуток (`/[ \t\r\n\f\v]/`)
- `/\S/` - Не символ-промежуток (`/[^ \t\r\n\f\v]/`)

Классы символов из Unicode

В свежих версиях Ruby существует поддержка классов символов из Unicode. Ссылка на класс `/\p{Class}/`:

- `/\p{Alnum}/` - Прописные и числовые символы
- `/\p{Alpha}/` - Символы алфавита
- `/\p{Blank}/` - Пробелы и табуляция
- `/\p{Cntrl}/` - Контрольные символы
- `/\p{Digit}/` - Цифры
- `/\p{Graph}/` - Печатные, но непустые символы
- `/\p{Lower}/` - Символы алфавита в нижнем регистре
- `/\p{Print}/` - Любые печатные символы
- `/\p{Punct}/` - Символы пунктуации
- `/\p{Space}/` - Символы-пробелы
- `/\p{Upper}/` - Символы алфавита в верхнем регистре
- `/\p{XDigit}/` - Символы хекс-чисел
- `/\p{Word}/` - Символ, который может быть частью слова