Я предпочитаю обеспечить много путей, если это возможно, но поощрять или вести пользователей, чтобы выбрать лучший путь, если это возможно

Юкихиро Мацумото

Руби: руководство по стилю оформления:

https://github.com/arbox/ruby-style-guide/blob/master/README-ruRU.md - по-русски
https://github.com/styleguide/ruby
https://github.com/airbnb/ruby

Комментарии к домашкам…

Переменные:

- local – только внутри метора, или блока
- @instance – привязаны к объекту\инстансу. У каждого свои. Шарятся между методами. По-умолчанию – nil. (В отличие от локальных)
- @@class – привязаны к классу, шаряться между объектами этого же класса. Должны быть проинициализированы перед использованием
- $global – привязаны к программе, шаряться между классами. По-умолчанию – nil
- CONS_TANT – внутри класса ведут себя как классовые, снаружи – как глобальные, но при этом “константы”. Вне класса – доступ по ::
- self – объект-получатель\текущий
- __FILE__
- __LINE__

… defaults & predefined

(ENV, RUBY_VERSION, STDIN, $LOAD_PATH, $, …...)

Ещё немного об условиях:

Когда elsif-ов становится слишком много, или нужно неявное сравнение, используется конструкция case.

- использует === в сравнениях

- выводит последнее исполнение

case < переменная или значение (объект на входе)>

when сравнение1 [, сравнение2 и тд через запятую]* [then]

< code >

when сравнение3 [, сравнение4]* [then]

< code >

…

[else < code >]

end

- результат высчитывается сравнение === объект на входе, не наоборот, сверху вниз

- как только истина – выполняет код

- если ничего нет – заходит в else (если он есть)

- если совсем ничего нет – то nil

- [then] - для онлайнеров

Пример:

number = 13

case number

when 10 then puts ‘number is 10’

when 20, 13

puts ‘number is 20 or 13’

else

puts ‘uknown’

end

=> number is 20 or 13

Важно заметить возможную ошибку:

number = 10

```
case number
when number > 4
puts ‘more than 4’
when number > 6, number > 8 then puts ‘more than 6’
else
puts ‘unknown’
end
# => unknown
```

Подумайте почему.

И ещё:

```
1 === Fixnum # => false
Fixnum === 1 # => true
```

Так происходит потому, что метод === на объекте самого класса Fixnum переопределен, а на объекте экземпляра класса Fixnum, единице - нет.

Немного об объектах, прежде чем идти далее.

Класс - это тип, описывающий устройство объектов.
Объект - некая сущность в памяти, появляющаяся при создании экземпляра класса (через #new).
Свойства объекта - его атрибуты.

В дальнейшем для удобства чтения будем употреблять термин *экземпляр* (instance) для *объекта* и термин *тип* для *класса*.

Так как каждый описываемый класс имет различные свойства и выполняемый код, то их нужно где-то хранить. Ruby хранит данные объектах. Поэтому у объектов, описывающих какой-то класс, так же должен быть свой тип. Отсюда тип (класс) объектов описывающих сами же классы - есть *Class*.

```
puts 1.class # => Fixnum
puts 1.class.class # => Class
```

pits 1.class.class.class # => Class
...
В данном примере и 1 и Fixnum - объекты, но 1 есть экземпляр типа (класса) Fixnum, а Fixnum в свою очередь - экземпляр типа Class.

Также все объекты тем или иным образом выведены из класса Object (он в свою очередь от BasicObject, начала всех начал).
Поэтому даже экземпляры пустого класса имеют минимальный набор методов:
class Test; end
test = Test.new
puts test.methods.inspect

В данном случае происходит автоматическое наследование Test от Object, что равносильно следующему:
class Test < Object; end

В Ruby все данные являются объектами, в отличие от многих других языков, где существуют примитивные типы (это означает, что и числа, и строки, и даже “ничто” в виде nil - всё это объекты определённого класса). Каждая функция — метод.

Мы ещё вернёмся подробней к объектной модели Ruby, а пока нужно запомнить, что всё (почти) - есть объект. И пару методов:
вывод типа объекта:
1.class # => Fixnum
вывод всех методов объекта (без учёта наследования):
1.methods
вывод уникального идентификатора объекта:
1.object_id

Объекты в парадигме ООП общаются между собой через сообщения, но иногда это для удобства называют вызовом метода. Пример:
‘ruby’.capitalize # => Ruby
но можно и так: ‘ruby’.send(:capitalize) # => Ruby

Находясь внутри irb объектом является специальный объект main. Поэтому каждый метод имеет место, где он определён, даже если вызывается не явно.
Как #puts например, который определён на классе IO.

Тема 3. Типы данных: Fixnum(Bignum), Float, Boolean, String, Symbol, Array, Hash, Range, Nil, Enumerable. Итераторы.

Мы уже начали что-то рассматривать, углубимся!

Репрезентация иерархии типов в Ruby:
http://dl.dropbox.com/u/306877/bigsoft/types.png

Числа:

Автоматический переход от типа к типу:
Fixnum < 2^30
Bignum > 2^30
Complex
Float

```
puts 2**3 # => 8 (Fixnum)
puts 2**-3 # => 1 / 8 (Rational) (начиная с Ruby 2)
```

Проблема с округлением у Float:

```
[243] pry(main)> 0.1 + 0.2 == 0.3
=> false
```

Решение
```
[242] pry(main)> (Rational(1,10) + Rational(2,10)).to_f == 0.3
=> true
```

```
12.4 * 1.05
=> 13.020000000000001
```

Решение:
```
require 'bigdecimal'
```

=> true
(BigDecimal('12.4') * BigDecimal('1.05')).to_f
=> 13.02

Важные методы:
Стоит отметить, что каждый возвращает не последнее значение, а сам вызывающий объект!

.times - повторение заданное количество раз
3.times { puts ‘1’ }
1
1
1
=> 3

.upto - повторение до заданного числа, по-возрастанию
2.upto(4) { puts ‘1’ } # от 2х до 4х: 2, 3, 4 - 3 раза
1
1
1
=> 2

.downto - повторение до заданного числа, по-убыванию, аналогичен .upto но вниз
.step(end, size) - проход от начального числа до числа = end, с шагом = size:
2.step(6, 2) { puts 1 }
1
1
1
=> 2

Важно, что всё они принимают аргументом блок (конструкцию в {}), который умеет аккумулировать текущее значение. Сравните с предыдущими вариантами:
2.step(6, 2) { |num| puts num }
2 # начинается с 2, как и вызывали
4
6

=> 2

3.times { |index| puts index }
0 # с нуля начинается
1
2
=> 3

2.upto(4) { |f| puts f }
2 # начинается с 2, как и вызывали
3
4
=> 2

Методы приведения типов:
.to_f - к Float
.to_c - к Complex
.to_i - к Integer (Fixnum/Bignum)
.to_s - к String (например полезно когда нужно добавить в конец строки)

** - возведение в степень:
2**3
=> 8
% - остаток от деления:
8 % 5
=> 3
.div - целочисленное
.abs - модуль:
-3.abs
=> 3

Самостоятельное изучение:
.between?
.odd?
.even?
.next

Логические:

true (TrueClass)
false (FalseClass)

Мы уже обсуждали, что в Ruby ложь - только nil и false

Строки:

Строки — это произвольной длины наборы символов, которые заключены в одинарные или двойные кавычки. Пример строк:
“this is a string”
‘this is a string in single quotes’

В Ruby:
- не ограничены размером
- динамичны (можно расширять, уменьшать..)
- любой объект можно преобразовать в строку (с помощью #to_s)
- можно и “ (двойные кавычки, позволяют интерполировать значение через #{}, и преобразовывают управляющие символы \n, \t ..) и ' (одинарные)
- переменные указывают на объект
- очень богатое API (Ruby хорошо работает с текстом и имеет множество библиотек для этого)

Инструкции для показа:

```
puts "String"
puts 'String'
puts "Time now is #{Time.now}"

puts 'Hello\t\tworld'
puts "Hello\t\tworld"
```

```
puts "Eat.Sleep.Repeat " * 2
puts "a" < "z"
puts "Z".ord
puts 89.chr

puts "Conca" + "tenation"
puts "Conca" << "tenation"

str1 = "A"
str2 = "B"
puts str1 + " and " + str2 + " are cool letters!"
puts "#{str1} and #{str2} are cool letters!"

puts "2 + 2 = 4".sub("2", "1")
puts "2 + 2 = 4".gsub("2", "1")

puts "ruby".capitalize
puts "RuBy".downcase
puts "RuBy".upcase
puts "ybur".reverse.capitalize

puts "LongString".size
puts "LongString".length

puts "LongString".chop.inspect
puts "LongString\na".chomp.inspect
puts "LongString\n".chomp.inspect
puts "LongString".chomp("ing")
puts "Ruby".swapcase
puts 'LongString'.scan('xx')
puts 'LongString'.scan('St')
puts 'LongString'.count('gn')


string = "abcdefg"
puts string[0]
```

```
puts string[100]
puts string[-1]
puts string[2, 3]
puts string[2..4]
puts string[2...4]
puts string.slice(2...4).methods.sort
puts string.index('b')
puts string.include? "abcd"
puts string.insert(4, 'XXX')

puts string.delete 'ae'
puts string.delete 'a-d'
"hello".each_char { |c| puts c }
"hello\nWorld".each_line {|c| puts c}

puts "string".empty?
puts " ".empty?
puts "".empty?

p string.split
puts '-3/2'.to_c

a = "tom"
puts "#{a}, #{a.object_id}"
a = "jerry"
puts "#{a}, #{a.object_id}"

a = "tom"
puts "#{a}, #{a.object_id}"
a.gsub!("tom", "jerry")
puts "#{a}, #{a.object_id}"
```

Переменные указывают на объект по ссылке, не по значению:

```
girlfriend = "Даша"
```

goes_on_a_visit = girlfriend
puts girlfriend
=> Даша

goes_on_a_visit[0] = "М” # меняем первую (номер ноль) букву у переменной-строки
puts girlfriend
=> Маша # WAT?

Значение, возвращаемое первой переменной, изменилось потому, что в Ruby переменные содержат лишь ссылку на то, что вы им присваиваете.
Сами данные (объект) лежат где-то в другом месте. Ввиду этого естественно, что при прямом изменении самого объекта, на который указывает переменная, все другие переменные, указывающие на этот объект, будут возвращать изменённое значение (также будут изменяться).
Чтобы girlfriend наша осталась "Даша", надо в переменную занести её клон:
girlfriend = "Даша"
goes_on_a_visit = girlfriend.clone
goes_on_a_visit[0] = "М” # Но изменили мы лишь клон. Дома в сохранности сидит настоящая:
puts girlfriend
=> Даша

Можно создавать копии объектов ещё методом .dup. Разница между ними будет понятна позже.
Для безвредного присваивания новых значений переменным их редко приходится клонировать, ибо большинство методов делают это и так. Даже если вы просто присвоите переменной новое значение, Ruby создаст объект с новым значением и поместит в (уже существующую) переменную ссылку на тот объект:

goes_on_a_visit = "Аристарх" # Создаётся новый объект, переменная переводится на него
p girlfriend
=> "Даша"

%Q

Альтернатива строке в двойных кавычках. Полезно, когда в строке имеются собственные кавычки - вместо экранирования обратным слешем можно использовать %Q:

```
what_tom_said = “Hello”
%Q(Mark said: "Tom said: "#{what_tom_said}"")
=> Mark said: "Tom said: "Hello""
```

Скобки здесь “(…)” могут быть заменены на любой не алфавитный символ:

```
%Q!Mark said: "Tom said: "#{what_tom_said}""!
%Q[Mark said: "Tom said: "#{what_tom_said}""]
%Q+Mark said: "Tom said: "#{what_tom_said}""+
```

Так же можно использовать:

```
>> %/Mark said: "Tom said: "#{what_tom_said}""/
=> Mark said: "Tom said: "Hello""
```

%q

Используется как альтернатива для строки в одинарных кавычках. Синтаксис похож на %Q, но уже без подстановки выражений и экранирования:

```
%q(Mark said: 'Tom said: '#{what_tom_said} ' ')
=> Mark said: 'Tom said: '#{what_tom_said} ' '
```

<<HEREDOC’s

Мы знаем, что Ruby позволяет создавать строковые литералы, которые занимают несколько физических строк:

```
str = "Welcome to
Ruby!"
```

В результате такого объявления переменная str будет хранить следущую строку: “Welcome to\nRuby!”, где \n является специальным символом новой строки.

Существует также следующий стиль (более аккуратный и желаемый) для создания многострочных строковых литералов:

```
str = <<STR
This is a
multiline
```

string
STR
Здесь, вместо STR — идентификатора конца строки может быть использован любой другой слово, например error_message, wellcome_message и так далее.

```
str = <<END
#{"Hello Chuck!"}
END
=> "Hello Chuck!\n"
str = <<"END"
#{"Hello Chuck!"}
END
=> "Hello Chuck!\n"
# обратите внимание на кавычки в которые заключен END
str = <<'END'
#{"Hello Chuck!"}
END
#=> "\#{\"Hello Chuck!\"}\n"
```

**Экранирование - через **

```
[524] pry(main)> a = 1
=> 1
[525] pry(main)> b = 2
=> 2
[526] pry(main)> "a + b = \#{a + b}"
=> "a + b = \#{a + b}"
[527] pry(main)> "a + b = #{a + b}"
=> "a + b = 3"
```

Показать:

```
[186] pry(main)> f = 'f'
=> "f"
[187] pry(main)> f.object_id
=> 14461460
[190] pry(main)> f << '1'
```

=> "f1"
[191] pry(main)> f.object_id
=> 14461460
[192] pry(main)> f = 'last'
=> "last"
[193] pry(main)> f.object_id
=> 13609880

Символы:

- храняться всё время выполнения программы, независимо от контекста
- уникальны (один и тот же объект) (показать имена методов)
- не изменяемы
- это не String, хоть чем-то и похожи
- GarbageCollector научится с ними работать только в последней версии руби (2.2)

Symbol.all_symbols

#to_s - данный метод используется для преобразования объекта в строку, если быть совсем точным, то объект преобразовываться в другой тип не может, просто создается другой, аналогичный объект — строкового типа, пример:
:symbol.to_s #=> "symbol"
#to_sym — сей метод предназначен для получения соответствующего строке символа, пример:
"string".to_sym #=> :string
Еще один малоизвестный способ преобразовать строку в символ — просто поставить перед строкой двоеточие, данный способ, вполне может заменить метод to_s в тех случаях, когда используется непосредственно для строки, а не для переменной ссылающейся на нее, примеры:
symbol = "string".to_sym #=> :string
symbol.object_id #=> 54088
symbol2 = :"string" #=> :string
symbol2.object_id # => 54088

Массивы:

Коллекцией называется любой набор элементов. Коллекция представляет собой объект определенного типа (экземпляр определенного класса), который состоит из других объектов. Если коллекция содержит в себе другие коллекции, то такая коллекция называется ветвещейся колекциейБ или деревом, или root-коллекцией (корневой коллекцией), так как из нее происходит ветвление. Двумя основными коллекциями в Ruby являются массивы и хэши.

Данные представленные в массиве имеют парную структуру ключ - значение, в которой ключ служит для идентификации элемента в массиве и может быть только целым числом. Стоит заметить, что ключей (индексов) у каждого элемента массива имеется целых два: левосторонний и правосторонний, как и у строк.

Так как Ruby язык высокого уровня, в котором в отличие от, например, C не нужно заранее определять размерность массива и тип хранимых в нем данных.

- доступ по уникальному числовому ключу
- индексация с 0
- не ограничены (лишь оперативной памятью)
- упорядочены
- может хранить данные разных типов
- поэлементное сравнение
- includes Enumerable

Инициализация:

[ ] - лучше всего (пробел не нужен)
Array.new(size = 0, default = nil) - объектный подход
%w{} - разберём отдельно
Array[]
Array.[](args*)

Примеры:

[] - пустой массив:
[1, 2, 3]
[‘hello’, ‘my’, ‘cruel’, ‘world’]
[1, ‘cat’, 2, ‘dog’] - смешанные типы
[1, [1, 2], ‘dog’, [‘duck’], nil] - и даже так

array = [1, [1, 2], ‘dog’, [‘duck’], nil, true]
array[0]
=> 1
array[-1]
=> true
array[100] # доступ к несуществующему элементу
=> nil
array[10] = ‘test’
=> ‘test’
array.inspect
=> [1, [1, 2], "dog", ["duck"], nil, true, **nil, nil, nil, nil, "test**"]
array[0] = false
=> false
array.inspect
=> [**false**, [1, 2], "dog", ["duck"], nil, true, nil, nil, nil, nil, "test"]
array[2] = [3, 4]
array.inspect
=> [false, [1, 2], **[3, 4]**, ["duck"], nil, true, nil, nil, nil, nil, "test"]

[array[1, 3]
=> [[1, 2], [3, 4], ["duck"]]
array[2..3]
=> [[3, 4], ["duck"]]
array[2...3]
=> [[3, 4]]

Полезные методы:

<< (.push) - добавление в конец (object_id не меняется)
.pop - удалить из конца и вывести
.shift - удалить сначала и вывести
.unshift - добавление в начало

- - приклеить в конец (по-элементно, сам не меняется, показать)

* - повторить несколько раз

- A - B, в результате этой операции возвращается массив уникальных элементов A

.concat (+=) - добавить в конец и изменить (спросить в чём отличие (object_id))
.compact - удаляет все nil
.flatten(level) - понизить уровень (рекурсивно)
.uniq - только уникальные
.each {|e| puts e} - итерация по каждому элементу, вывод
.map/collect
.select / reject
.inject
.with_index
.join() - соеденить все элементы массивы в строку
.sample(3, random: rng) - произвольная выборка (без повторов если в самом массиве их нет)
.sort (если элементы сравнимы) и .sort_by
.reverse - обратный порядок
.size - количество элементов
.count - количество элементов (может принимать блок)
.first - первый элемент
.last - последний элемент
.values_at(1, 2, 3) - значения по индексам (может принимать несколько индексов)
.index(1) - индекс первого найденного
.delete(1) - удаление по значению
.delete_at(1) - удаление по индексу
.insert(index, value) - вставка значения на место индекса со сдвигом
.shuffle - перетусовать порядок элементов
.include?(value) - проверка на включение

.methods (as usual :))

Сравнение по-элементно (порядок важен).
Операции как над множествами: &, |, ⇔, ...
%w vs %W
a = Array.new(10) {| elem| elem.odd? ? elem**2 : elem**3 }
#=> [0, 1, 8, 9, 64, 25, 216, 49, 512, 81]

Задание:

- Дан 10-ти элементный массив. В одну строку удалить 5 последних элементов.
- Посчитать сумму 5-значного массива

Массивы часто используются для обработки строк и даже хэшей:
.join / .split / .sort

```
[157] pry(main)> a = 1
=> 1
[158] pry(main)> b = 2
=> 2
[159] pry(main)> c = [a,b]
=> [1, 2]
[160] pry(main)> c
=> [1, 2]
[161] pry(main)> a += 1
=> 2
[162] pry(main)> a
=> 2
[163] pry(main)> c
=> [1, 2]
```

```
[164] pry(main)> a = '1'
```

```
=> "1"
[165] pry(main)> c = [a, b]
=> ["1", 2]
[166] pry(main)> a << '1'
=> "11"
[167] pry(main)> c
=> ["11", 2]
```

Хэши:

- они же ассоциативные массивы (почему), они же словари
- не упорядочены
- саморасширяемы, по-умолчанию - nil
- может хранить данные разных типов
- похожи на массивы

Ключами могут быть сложные объекты, но лучше всего использовать символы.
Почему?
Показать пример символов и памяти.

Инициализация:

```
{}
{a: 1}
{:a => 1}
Hash.new
Hash[]
```

Синтаксис создания хэша с Hash.new позволяет принимать блок кода, который выполняется при обращении к несуществующщему ключу, создает его и соответствующее ему значение по правилам определенным в блоке кода, пример:

```
hash = Hash.new { |h,k| h[k] = k }
=> {}
```

hash[:a]
=> :a
hash
=> { :a => :a }
Другими словами, в такой способ можно создать значение по умолчанию.
Синтаксис Hash[] позволяет создавать хэш из предоставляемого массива. Такой синтаксис создания хэша, как бы просит у переданного массива расчитаться на 1й — 2й, где 1й — ключ, 2й — значение, пример:
hash = Hash[1, 2, 3, 4, 5, 6]
=> { 1=>2, 3=>4, 5=>6 }
hash = Hash[:a, 1, :b, 2, :array, [1, 2, 3, 4]]
=> { :a=>1, :b=>2, :array=>[1, 2, 3, 4] }
Полезные методы:

.each { |key, value| … } - итерация по парам ключ-значение
.keys / . values - массив ключей / массив значений
.to_a - конвертация в массив
.merge(hash2) - объединие с другим хэшем
.default - данный метод позволяет установить хэшу значение по умолчанию. В массивах значение по умолчанию всегда nil, а в хэшах его можно очень просто установить, пример:
h= Hash.new
=> {}
h[:a]
=> nil
h.default = 'ruby'
h[:a]
=> "ruby"
h[:b]
=> "ruby"
.invert — данный метод позволяет преобразовать структуру хэша так, что ключи и значения меняются местами, пример:
h = { a: 100, b: 200, c: 300 }
h.invert
=> { 100=>:a, 200=>:b, 300=>:c }

Хотя массивы и позволяют хранить элементы различных типов, это не является самым хорошим стилем. Вам следует использовать массивы для хранения более-менее одинаковых элементов или вложенных структур, которые в свою очередь так же могут быть массивами или хэшами.

Массив -> Хэш -> Массив
Для преобразования массива в хэш достаточно воспользоваться специальным синтаксисом создания хэша: Hash[], который разбивает массив на пары ключ — значение, примеры:

```
array = [1, 2, 3, 4]
hash = Hash[*array]
=> { 1=>2, 3=>4 }
```

Обратите внимание на символ * — это оператор носит название «splat» или «asterisk» и используется для преобразования массива в набор аргументов, пример:

```
array
=> [1, 2, 3, 4]
[array]
=> [[1, 2, 3, 4]]
[*array]
=> [1, 2, 3, 4]
```

Для преобразования хэша в массив имеется специальный метод to_a, пример:

```
hash
=> { 1=>2, 3=>4 }
hash.to_a
=> [[1, 2], [3, 4]]
```

Ещё массив пар массивов можно привести к хэшу с помощью метода .to_h

```
ar = [[:a, 1], [b: 2]]
ar.to_h
=> { :a=>1, :b=>2 }
```

Немного о строковых ключах:

```
h.keys.last
```

```
=> "f"
h.keys.last << '1'
=> RuntimeError: can't modify frozen String


[34] pry(main)> hash = {:a => 1}
=> {:a=>1}
[35] pry(main)> b = :a
=> :a
[36] pry(main)> b.object_id
=> 361768
[37] pry(main)> hash.keys.first.object_id
=> 361768


[272] pry(main)> Symbol.all_symbols.size
=> 5210
[273] pry(main)> a = :tort
=> :tort
[274] pry(main)> Symbol.all_symbols.size
=> 5211
[275] pry(main)> h = {:tort => 1}
=> {:tort=>1}
[276] pry(main)> Symbol.all_symbols.size
=> 5211
```

Диапазоны:

- они же интервалы: набор значений с началом и концом
- class == Range
- достаточно уникальный тип данных, не в каждом языке есть

Инициализация:

(1..10)
(1...10)
('a'..'z') # и даже так

Полезные методы:

.min / .max
.include? / .member?
.to_a

Между #min и #first, а также #max и #last существует маленькое, но очень важное различие, незнание которого может привести к ошибкам в коде! Это различие проявляется тогда, когда используются обратные диапазоны, пример:

(1..100).first
=> 1
(1..100).min
=> 1
(100..1).min
=> nil
(100..1).first
=> 100
(1..100).max
=> 100
(1..100).last
=> 100
(100..1).max
=> nil
(100..1).last
=> 1
Как видите, #min и #max при обратных диапазонах возвращают значение nil, этим можно пользоваться, когда вам необходимо узнать, диапазон прямой или обратный, во всех остальных случаях я бы рекомендовал использовать методы #last и #first, если

разумеется нет ограничения на то, каким должен быть диапазон (обязательно ли он должен быть прямым?).

Итерация по диапазону:

#each — сей метод позволяет вам проходить по каждому элементу входящему в диапазон и выполнять с ним определенные действия, например:

```
 (1..10).each { |e|  print(e, ', ') }
 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
 => 1..10
```

#step — данный метод нам уже знаком, так как мы уже сталкивались с ним, когда разбирались с целыми числами. Его отличие от целочисленного #step состоит в том, что обе границы заданы и нам необходимо задать лишь шаг итерации, пример:

```
('a'..'z').step(2) { |ch| print ch, '', '' }
a, c, e, g, i, k, m, o, q, s, u, w, y,
=> "a".."z"
```

Диапазоны очень полезны, например, они могут здорово помочь тогда, когда нам необходимо создать массив последовательных элементов:

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

Представьте, что элементов не 10, а тысяча и вы поймете, что диапазоны действительно удобны. Разумеется, такой массив можно было бы создать при помощи цикла, однако это лишний, ухудшающий читабельность код!

Диапазоны очень удобны в использовании, с выражением case, пример:

```
a = 101
case a
when 1...50 then puts "0 < a < 50"
when 50...100 then puts "50 <= a < 100"
when 100...150 then puts "100 <= a < 150"
else
 puts "a > 150"
```

end
=> ‘100 <= a < 150’
Если бы не диапазон, вы бы не смогли написать такой простой и красивый код.

Nil:

- nil
- тоже объект :)

.nil?
.to_s..
.inspect

Regexp

- тоже объект :)
- // - инициализация
- возвращает nil, если не нашло
- стандартный синтаксис: ?,+,*,[], \d, \s и тд...
- http://rubular.com/

s = ‘strintg’
s =~ /t/
=> 1
s.scan(/t/)
=> [‘t’, ‘t’]

Циклы и управление потоком выполненения:

Вспомнили if..end / case - тоже управление потоком

Лучше использовать рубёвый .each чем for.

```
loop do
end
```

```
for element in somewhere
 puts element
end
```

- for есть для всех, у кого есть .each. Почему?)

```
while условие_правдивости do  # do не обязательно
        # не забыть изменять условие правдивости
end
```

упрощённая форма: do_something while правда

```
until ложное_условие_к_которому_стремимся do  # do не обязательно
        # не забыть изменять ложное_условие_к_которому_стремимся
end
```

упрощённая форма: do_something until ложное_условие_к_которому_стремимся

Прерывание циклов:

break – выходит из цикла к следующей инструкции
redo – начинает цикл сначала, но без пересчёта условия цикла или следующего элемента (если для итераторов – повтор прохода с текущим значением)
next – пропускает остаток и начинает следующий проход этого же цикла
retry – запускает тело сначала (обнуляет значение) (пока пропускаем, так как оно работает только при наличие обработчика иключений)

Вывод всех элементов массива:

```
array = [1, 2, 3, 4, 5]
for i in array
 puts i
```

```
end


puts 5.times.class
enum = 5.times
puts enum.inspect
# вывод чисел от 0 до 4
enum.each { |e| puts e }
```

Особая форма: вывод i от 5ти до 10:

```
i = 5
while (i == 5)...(i == 10) do
 puts i
 i += 1
end
```

Вывод чисел от 5 (4+1) до 10 (не включая), исключив 6.

```
x = 4
while x < 10
 x += 1
 redo if x == 6
 puts x
end
```

```
# выведет 3, 4, 5
(пока меньше 3х - перезапускаем, выводим 3..5, дальше срабатывает break)
i=0
loop do
 i += 1
 next if i < 3
 puts i
 break if i > 4
end
```

Дополнительный индекс, разобрать самостоятельно:

```
a = 0
```

```
=> 0
i = 0
=> 0
while a < 10
 puts a
 i += 1
 redo if a == 4 && i <  10
 a += 1
end
0
1
2
3
4
4
4
4
4
4
5
6
7
8
9
=> nil
```

Bonus (для любопытных):

Оптимизация над числами:

```
[200] pry(main)> a = 3
=> 3
[201] pry(main)> b = 3
```

```
=> 3
[202] pry(main)> a.object_id
=> 7
[203] pry(main)> b.object_id
=> 7
[204] pry(main)> a = 11111111111
=> 11111111111
[205] pry(main)> b = 11111111111
=> 11111111111
[206] pry(main)> a.object_id
=> 22222222223
[207] pry(main)> b.object_id
=> 22222222223
[208] pry(main)> a = 32222222222222222222222222222222
=> 32222222222222222222222222222222
[209] pry(main)> b = 32222222222222222222222222222222
=> 32222222222222222222222222222222
[210] pry(main)> b.object_id
=> 10130160
[211] pry(main)> a.object_id
=> 13256720


[218] pry(main)> t = 1
=> 1
[219] pry(main)> y = t
=> 1
[220] pry(main)> t +=1
=> 2
[221] pry(main)> t.object_id
=> 5
[222] pry(main)> y.object_id
=> 3
[223] pry(main)> s = :s
=> :s
[224] pry(main)> p = s
```

```
=> :s
[225] pry(main)> s.object_id
=> 309448
[226] pry(main)> p.object_id
=> 309448


t = Time.now
=> "2015-01-12 23:31:50 +0300"
t.hour
=> 23
t.monday?
=> true
t.utc
=> 2015-01-12 20:31:50 UTC

%w(1 2 3)
%W(1 #{1+1} 3)
=> [‘1’, ‘2’, ‘3’]

a = 0
a += 1 # a = a + 1
a -= 1 # a = a - 1
a *= 1 # a = a * 1
a /= 2 # a = a / 2
```

Домашнее задание #2:

Теория*:**

- прочесть заметки лекции ещё раз, два, три...
- изучить следующие ссылки:

  - информация о числах https://ru.wikibooks.org/wiki/Ruby/%D0%9F%D0%BE%D0%B4%D1%80%D0%

BE%D0%B1%D0%BD%D0%B5%D0%B5_%D0%BE_%D1%87%D0%B8%D1%81%D0%BB%D0%B0%D1%85

- информация о строках
  https://dl.dropboxusercontent.com/u/306877/bigsoft/more_about_strings.html
- https://ru.wikibooks.org/wiki/Ruby/%D0%9F%D0%BE%D0%B4%D1%80%D0%BE%D0%B1%D0%BD%D0%B5%D0%B5_%D0%BE_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2%D0%B0%D1%85
- https://ru.wikibooks.org/wiki/Ruby/%D0%9F%D0%BE%D0%B4%D1%80%D0%BE%D0%B1%D0%BD%D0%B5%D0%B5_%D0%BE%D0%B1_%D0%B0%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D1%85_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2%D0%B0%D1%85
- http://rubydev.ru/2010/05/ruby-iterators-loops/
- Осмотреться в документации:
  http://www.ruby-doc.org/core-2.2.0/Fixnum.html
  http://ruby-doc.org/core-2.2.0/Float.html
  http://www.ruby-doc.org/core-2.1.5/Integer.html
  http://www.ruby-doc.org/core-2.2.0/String.html
  http://www.ruby-doc.org/core-2.1.5/Symbol.html
  http://www.ruby-doc.org/core-2.2.0/Array.html
  http://ruby-doc.org/core-2.1.5/Hash.html
  http://ruby-doc.org/core-2.2.0/Range.html
  http://ruby-doc.org/core-2.1.1/Regexp.html

- составить список вопросов

- выделить одну интересную и запомнившуюся особенность/метод/факт связанный с новым материалом

Практика:

Написать скрипт, который выводит последовательность чисел Фибонначи

- скрипт должен быть исполняем
- он должен принимать только один аргумент, указывающий на число элементов в последовательности

Подсказка:

- используйте массив ARGV, чтобы получить входящие аргументы

- не забывайте про циклы

Пример работы: $ ./script.rb 7
=> 0 1 1 2 3 5 8
Дополнительные задачки для самопроверки. Решить любые 2.

- Создать переменную для строки "dniMyMdegnahCybuR". Написать **однострочную** инструкцию, которая поменяет порядок букв на противоположный и понизит все буквы в регистре кроме первой. Вывести результат. Значение переменной должно быть изменено.
- Дано семизначное число. Вывести на экран число, где первая цифра стала последней, вторая - предпоследней и тд.
- Дано целое число. Найти и вывести сумму его цифр.
- Дана строка. Необходимо подсчитать количество букв "а" в этой строке (независимо от регистра)
- Дана строка. Проверить, является ли она палиндромом (https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D0%BB%D0%B8%D0%BD%D0%B4%D1%80%D0%BE%D0%BC)
- циклом вывести на экран числа от 10 до 3 не влючая 5, причём числа, кратные 3м, вывести в квадрате :)
- дан хэш shop = {
  milk: 10,
  bread: 8,
  cornflakes: 12,
  ice_cream: 15,
  pie: 20
  }
  ответить на вопрос: если ли в магазине какой-либо продукт с ценой в 15?
- дан массив ar = [1, 6,1,8,2,-1,3,5]. Прибавить 100 к его максимальному элементу:
  puts ar
  => [1, 6, 1, 108, 2, -1, 3, 5]
- дан массив: ar = [7, 3, [4, 5, 1], 1, 9, [2, 8, 1]]
  вывести отсортированный по убыванию массив из уникальных элементов начального массива:
  puts ar
  => [9, 8, 7, 5, 4, 3, 2, 1]

Домашку присылать по адресу: aliaksandr_buhayeu@epam.com с темой письма: MTN:L_2:ИМЯ_ФАМИЛИЯ.

Называйте ваш скрипт: имя_фамилия_l2.rb
Спасибо!

На этом всё, жду вас на следующем занятии!