

Ruby Cheat Sheet

Scripting for Testers

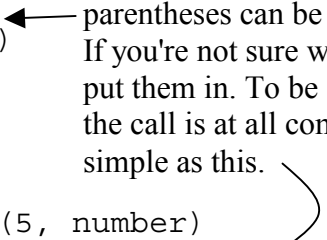
This cheat sheet describes Ruby features in roughly the order they'll be presented in class. It's not a reference to the language. You do have a reference to the language: the full text of Programming Ruby: The Pragmatic Programmer's Guide is installed with Ruby. Select Start-> Programs-> Ruby-> RubyBook Help to see what's known as the pickaxe book.

Function calls

```
puts "hello"
puts("hello")
```

← parentheses can be omitted if not required. If you're not sure whether they're required, put them in. To be safe, put them in whenever the call is at all complicated. Even one as simple as this.

```
assert_equal(5, number)
```



Variables

Ordinary ("local") variables are created through assignment:

```
number = 5
```

Now the variable `number` has the value 5. Ordinary variables begin with lowercase letters. After the first character, they can contain any alphabetical or numeric character. Underscores are helpful for making them readable:

```
this_is_my_variable = 5
```

A variable's value is gotten simply by using the name of the variable. The following has the value 10:

```
number + this_is_my_variable
```

Conditional tests (if)

```
if number == 5
  puts "Success"
else
  puts "FAILURE"
end
```

← A **string**. Strings can be surrounded with single or double quotes.

Put the `if`, `else`, and `end` on separate lines as shown. You don't have to indent, but you should.

Function definitions

```
def assert_equal(expected, actual)
  if expected != actual
    puts "FAILURE!"
  end
end
```

Functions can return values, and those values can be assigned to variables. The return value is the last statement in the definition. Here's a simple example:

```
def five
  5
end

variable = five
```

Note that no parentheses are required.

Variable's value is 5. Note that we didn't need to say `five()`, as is required in some languages. You can put in the parentheses if you prefer.

Here's a little more complicated example:

```
def make_positive(number)
  if number < 0
    -number
  else
    number
  end
end

variable = make_positive(-5)
variable = make_positive(five)
```

Variable's value is 5.

Variable's value is 5.

Regular expressions

Regular expressions are a useful feature common to many languages. They allow you to match strings.

Regular expressions are characters surrounded by `//` or `%r{ }`. A regular expression is compared to a string like this:

```
regexp =~ string
```

Most characters in a regular expression match the same character in a string. So, these all match:

```
/a/ =~ 'a string'
/a/ =~ 'string me along'
```

This also matches:

```
/as/ =~ 'a string with astounding length'
```

Notice that the regular expression can match anywhere in the string. If you want it to match only the beginning of the string, start it with a caret:

```
/^as/ =~ 'alas, no match'
```

If you want it to match at the end, end with a dollar sign:

```
/no$/ =~ 'no match, alas'
```

If you want the regular expression to match any character in a string, use a period:

```
/^./ =~ "As if I didn't know better!"
```

There are a number of other special characters that let you do amazing and wonderful things with strings. Ruby uses the standard syntax for regular expressions used in many scripting languages. See *Programming Ruby* for details.

Truth and falsehood (optional)

Read this only if you noticed that typing regular expression matching at the interpreter prints odd results.

You'll see that the ones that match print a number. That's the position of the first character in the match. The first expression (`/a/ =~ 'a string'`) returns 0. (Ruby, like most programming languages, starts counting with 0.) The second returns 10.

What happens if there's no match? Type this:

```
/^as/ =~ 'alas, no match'
```

and the result will be `nil`, signifying no match. You can use these results in an `if`, like this:

```
if /^as/ =~ some_string
  puts 'the string begins with "as".'
end
```

In Ruby, anything but the two special values `false` and `nil` are considered true for purposes of an `if` statement. So match results like 0 and 10 count as true.

Objects and methods and messages

A function call looks like this:

```
start('job')
```

A method call looks much the same:

```
"bookkeeper".include?('book') returns true
```

The difference is the thing before the period, which is the *object* to which the *message* is sent. That message invokes a *method* (which is like a `def`'d function). The method operates on the object.

Different types of objects respond to different messages. Everything in Ruby is really an object.

Arrays

This is an array with nothing in it:

```
[]
```

This is an array with two numbers in it:

```
[1, 2]
```

This is an array with two numbers and a string in it. You can put anything into an array.

```
[1, 'hello!', 220]
```

Here's how you get something out of an array:

```
array = [1, 'hello', 220]
array[0]           value is 1
```

Here's how you get the last element out:

```
array[2]           value is 220
```

Here's another way to get the last element:

```
array.last         value is 220
```

Here's how you change an element:

```
array[0] = 'boo!'   value printed is 'boo!'
                  array is now ['boo', 'hello', 220]
```

How long is an array?

```
array.length       value is 3
```

Here's how you tack something onto the end of an array:

```
array.push('fred')  array is now ['boo', 'hello', 220, 'fred']
```

There are many other wonderful things you can do with an array, like this:

```
[1, 5, 3, 0].sort   value is [0, 1, 3, 5]
a = ["hi", "bret", "p"]
a.sort              value is ["bret", "hi", "p"]
```

Iteration

How can you do something to each element of an array? The following prints each value of the array on a separate line.

```
[1, 2, 3].each do | value |
  puts value
end
```

If you prefer, you can use braces instead of do and end:

```
[1, 2, 3].each { | value | puts value }
```

What if you want to transform each element of an array? The following capitalizes each element of an array.

```
["hi", "there"].collect { | value | value.capitalize }
```

The result is ["Hi", "There"].

You may be more familiar with using for-loops for iteration. These also work in Ruby:

```
for value in [1, 2, 3]
  puts value
end
```

But the use of `each` is more common. The code that appears after the `each` is called a *block*. Blocks can be used for other things as well.

Blocks

A block is like a function without a name. It contains a set of parameters and one or more lines of code. Blocks are used a lot in Ruby.

Here's how to search an array for an element:

```
gems = ['emerald', 'pearl', 'ruby']
gems.detect { |gem| /^r/ =~ gem }
           returns "ruby"
```

When blocks are longer than one line, they are usually written using `do` and `end`.

Dictionaries

A *dictionary* lets you say "Give me the *value* corresponding to *key*." Dictionaries are also called *hashes* or *associative arrays*.

Here's how you create a dictionary:

```
dict = {}
```

Here's how you associate a value with a key:

```
dict['bret'] = 'texas'    looks a lot like an array, except that the key
                           doesn't have to be a number.
```

Here's how you retrieve a value, given a key:

```
dict['bret']              value is 'texas'.
```

Here's how you ask how many key/value pairs are in the dictionary:

```
dict.length               value is 1
```

What values does a dictionary have?

```
dict.values               value is the Array ['texas'].
```

What keys does it have?

```
dict.keys                 value is the Array ['bret'].
```