Ruby Code Count™

Counting Standard

University of Southern California

Center for Systems and Software Engineering

December , 2016

Revision Sheet

Date	Version	Revision Description	Author
7/6/2016	1.0	Original Release	Derek Lengenfelder
Date	Version No.	Click here to Description.	Name

Table of Contents

1. Definitions

1.1. **SLOC –** Source Lines of Code is a unit used to measure the size of software program. SLOC counts the program source code based on a certain set of rules. SLOC is a key input for estimating project effort and is also used to calculate productivity and other measurements.

1.2. **Physical SLOC –** One physical SLOC is corresponding to one line starting with the first character and ending by a carriage return or an end-of-file marker of the same line, and which excludes the blank and comment line.

1.3. **Logical SLOC –** Lines of code intended to measure "statements", which normally terminate by a semicolon (C/C++, Java, C#) or a carriage return (VB, Assembly), etc. Logical SLOC are not sensitive to format and style conventions, but they are language-dependent.

1.4. **Data declaration line or data line –** A line that contains declaration of data and used by a compiler or assembler to interpret other elements of the program. Ruby does not contain any data declarations.

1.5. **Compiler Directives –** A statement that tells the compiler how to compile a program, but not what to compile.

The following table lists the Ruby keywords that denote compiler directive lines:

$LOAD_PATH	require_relative

Table 1 Ruby Compiler Directive

1.6. **Blank Line –** A physical line of code, which contains any number of white space characters (spaces, tabs, form feed, carriage return, line feed, or their derivatives).

1.7. **Comment Line –** A comment is defined as a string of zero or more characters that follow language-specific comment delimiter. Ruby comment delimiters are "#" and "=begin", ending with "=end". A whole comment line may span one line and does not contain any compilable source code. An embedded comment can co-exist with compilable source code on the same physical line. Banners and empty comments are treated as types of comments.

NOTE: The '#' character is also used for other purposes within Ruby, apart from delimiting comments.

1.8. **Executable Line of code –** A line that contains software instruction executed during runtime and on which a breakpoint can be set in a debugging tool. An instruction can be stated in a simple or compound form.

An executable line of code may contain the following program control statements:

- Selection statements (if, ? operator, case)
- Iteration statements (for, while, loop-do-break, begin-end-while)

- Jump statements return, next, break, retry, redo, yield)
- Expression statements (function calls, assignment statements, operations, etc.)
- Block statements

An executable line of code may not contain the following statements:

- Compiler directives
- Data declaration (data) lines
- Whole line comments, including empty comments and banners
- Blank lines

2. Checklist for source statement counts

PHYSICAL SLOC COUNTING RULES			
MEASUREMENT UNIT	ORDER OF PRECEDENCE	PHYSICAL SLOC	COMMENTS
Executable lines	1	One per line	Defined in 1.8
Non-executable lines			
Declaration (Data) lines	2	One per line	Defined in 1.4
Compiler directives	3	One per line	Defined in 1.5
Comments			Defined in 1.7
On their own lines	4	Not included (NI)	
Embedded	5	NI	
Banners	6	NI	
Empty comments	7	NI	
Blank lines	8	NI	Defined in 1.6

LOGICAL SLOC COUNTING RULES				
NO.	STRUCTURE	ORDER OF PRECEDENCE	LOGICAL SLOC RULES	COMMENTS
R01	"for", "while" or "if" statement	1	Count once	"while" is an independent statement
R02	begin-end-while OR loop do-condition-break-end	2	Count once	The "end" keyword is not counted.
R03	Statements ending by a semicolon	3	Count once per statement, including empty statement	Semicolons within "for" statement are not counted. Semicolons used with R01 and R02 are not counted
R04	Block delimiters	4	Count once per block	Function definition is counted once
R05	Compiler Directive	5	Count once per directive	

3. Examples

EXECUTABLE LINES		
SELECTION Statement		
ESS1 - if, else if, else and nested if statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
if (<boolean expression>) <statements> end	if (x != 0) printf("non-zero") end	1 1 0

if (<boolean expression>)
  <statements>;
else if (<boolean expression>)
  <statements>;
…
else
  <statements>;
end
	if (x == 0)
	  printf("zero");
	elsif (x > 0)
	  printf("positive");
	else
	  printf("negative");
	end
		1
		1
		1
		1
		0
		1
		0

if (<boolean expression>)
  <statements>;
else
  <statements>;
end
	if (x > 0)
	  printf("positive")
	else
	  printf("not positive")
	end
		1
		1
		0
		1
		0

NOTE: complexity is not considered, i.e. multiple "&&" or "||" as part of the expression.	if ((x != 0) && (x > 0))
	  printf("%d", x);
	end
		1
		1
		0

ESS2 - ?: operator		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
Exp1 ? Exp2 : Exp3	x >= 0 ? printf("+") : printf("-")	1
ESS3 – case-when-else-end statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

case (<expression>)
when <constant 1>
  <statements>
when <constant 2>
  <statements>
when <constant 3>
  <statements>
else
  <statements>
end
	case (grade)
	when “A”
	  puts “Top marks!”;
	when “B”
	  puts “Good try.”;
	when “C”
	  puts “You are having difficulty.”;
	else
	  puts “Seek assistance.”;
	end
		1
		1
		1
		1
		1
		1
		1
		0
		1
		0

ESS4 – unless statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

unless <expression> [then]
  <statements>
else
  <statements>
end
 	
<statements> unless <bool expr>
 	
NOTE: complexity is not considered, i.e. multiple “&&” or “||” as part of the expression.
	unless (x != 0)
	  printf(“non-zero”);
	else
	  printf(“zero”);
	end
	
	printf(“positive”) unless x > 0
	
	if ((x != 0) && (x > 0))
	  printf(“%d”, x);
	end
		1
		1
		0
		1
		0
		
		1
		
		1
		1
		0

ITERATION Statement		
EIS1 – for loops		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
for variable [, variable ...] in expression [do] <statements> end	for i in 0..5 puts "Value of variable is #{i}"; end	1 1 0
EIS2 – while loops		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

while <boolean expression> [do]
  <statements>
end
 	
<statement[; statement;…]>while <boolean expression>
 	
begin
	while $i < $num
	  puts (“Inside the loop i = $i”);
	  $i += 1;
	end
	
	puts $i += 2 while $i < 10
	
	begin
		1
		1
		1
		0
		
		2
		
		1

<statements> end while <boolean expression>	puts (“Inside the loop $#$i”); $i += 1; end while $i < $num	1 1 1
EIS3 – until loops		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
until <Boolean expr> [do] <statements> end <statement[; statement;…]>until <boolean expression> begin <statements> end until <boolean expression>	until $i > $num puts("Inside the loop i = #$i"); $i +=1; end puts $1 += 2 until $i > 10 begin puts("Inside the loop i = #$i"); $i +=1; end until $i > $num	1 1 1 0 2 1 1 1 1
EIS4 – each iterator		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<collection>.each do |variable[, variable...]| <statements> end	(0..5).each do |i| puts "Value of variable is #{i}"; end	1 1 0
EIS5 – collect iterator		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<collection> = <collection>.collect <collection> = <collection>.collect{|variable| expr}	b = a.collect c = a.collect{|x| 10*x}	1 2
JUMP Statement		
EJS1 – throw statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
throw <: labelname> throw <:labelname> <condition>	throw :greeting throw :greeting if TIME == 0	1 2
EJS2 – catch statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

catch <:labelname> do <statements> end	catch :greeting do puts “Good morning!”; end	1 1 0
EJS3 – return statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
return <expression>	def test2 i = 100; j = 200; k = 300; return i,j,k; end	1 3 1 0
<condition> return	if x < 0 return	2
EJS4 – break statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
break	if (i > 2) then break; end	2
EJS5 – next statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
next	if (i > 2) then next end	1 1 0
EJS6 – redo statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
redo	if (i > 2) then redo end	1 1 0
EJS7 - retry statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
begin <statements> rescue <statements> retry end retry <condition>	begin nil; # exception raised rescue nil; # handles error retry # restart from begin block end for i in 1..5 retry if i > 2 puts "Value of variable is #{i}"	1 1 1 1 1 0 1 2 1

	end	0
EJS8 – abort statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
abort	abort	1
EJS9 – exit statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
exit(<result>) exit!(<result>)	exit(0) exit!(0)	1 1
EXPRESSION Statement		
EES1 - assignment statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<name> = <value> <name1> = <name2>	y = 3; x = y; $num = 10 @cust_name = "name" @@no_of_customers = 4 PI = 3.141592	2 1 1 1 1
EES2 – empty statement (is counted as it is considered to be a placeholder for something)		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

one or more ";" but not following another statement	while $i < 10 do
	  puts "Hello!";
	  $i += 1;
	  ;
	end
		1
		1
		1
		0

EES3 – function call		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<function_name> (<parameters>)	puts "Hello"	1
EES4 – function calls (special)		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

raise	begin
	  puts 'I am before the raise.'
	raise 'An error has occurred.'
	  puts 'I am after the raise.'
	rescue
	  puts 'I am rescued.'
	end
		1
		1
		1
		1
		1
		1
		0

require	require "Week"	1
include	class Decade	1
	include Week	1
	no_of_yrs=10	1
	def no_of_months	1
	puts Week::FIRST_DAY	1
	number = 10*12	1
	puts number	1
	end	0
	end	0
BLOCK Statement		
EBS1 – yield statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
yield [var1, var2, …]	def test1	1
	yield	1
	end	0
	def test2	1
	yield 5	1
	end	0
EBS2 – do-end statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<method invocation> do	test1 do	1
<statements>	puts "You are in the block"	1
end	end	0
EBS3 – {} delimiters		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<method_invocation> {	test2 {	1
<statements>	puts "You are in the block"	1
}	}	0
EBS4 – BEGIN-END statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
BEGIN {	BEGIN {	1
<statements>	puts "Initializing Ruby Program"	1
}	}	0
END {	END {	1
<statements>	puts "Ending Ruby Program"	1
}	}	0

EBS5 – begin-rescue-else-ensure-end statement

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT

begin
  <statements>
rescue
  <statements>
else
  <statements>
ensure
  <statements>
end
	begin
	  puts "I'm not raising exception"
	rescue Exception => e
	  puts e.message
	  puts e.backtrace.inspect
	else
	  puts "Congrats--noerrors!"
	ensure
	  puts "Ensuring execution"
	end
		1
		1
		1
		1
		1
		0
		1
		1
		1
		0

CLASS AND MODULE Statement

ECS1 – class statement

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
class <class_name> <statements> end	class Customer @@no_of_customers = 0 end	1 1 0

ECS2 – def statement

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
def <method_name>[var = value] <statements> end	def hello puts "Hello Ruby!" end	1 1 0

ECS3 – undef statement

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
undef <method_name>	undef hello	1

ECS$ – alias statement

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
alias	alias <new_method> <old_method> alias <new_glob_var> <old_glob_var>	1 1

ECS5 – super statement

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
super	class Employee < Sample	1

	def initialize(fname, lname, pos) super(fname,lname) @position = pos end def to_s super + ", #@position" end end	1 1 1 0 1 1 0 0
ECS6 - module statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
module <module identifier>	module Trig PI = 3.141592654 def Trig.sin(x) nil; # Code for sine of x end def Trig.cos(x) nil; # Code for cosine of x end end	1 1 1 1 0 1 1 0 0
OPERATORS AND PSEUDO VARIABLES		
EOP1 – defined? statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
defined? [parameter] (parameter = variable, method_call, super, yield)	defined? foo defined? $_ defined? puts defined? puts(bar) defined? super defined? yield	1 1 1 1 1 1
EOP2 – nil statement		
GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<variable> = nil; (functions as a variable with a logic value false) nil (functions as a placeholder)	@name = nil; def Trig.sin(x) nil # Code for sine of x end	1 1 1 0

COMPILER DIRECTIVES

CDL1 – directive type

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
$LOAD_PATH	$LOAD_PATH << '.'	1
require_relative	require_relative Trig	1

4. Complexity

Complexity measures the occurrences of different keywords in code baseline. Below table identifies the categories and their respective keywords that are counted as part of the complexity metrics.

Math Functions	Trig	Log	Calculations	Conditionals	Logic	Pre-processor	Assignment
exp	atan2	log	%	if	&&	$LOAD_PATH	=
frexp	cos	log10	+	elsif	||	require_relative	
ldexp	sin		-	when	==		
rand	tan		*	for	!=		
sqrt			**	while	<=>		
srand			/	unless	!		
			>>	until	and		
			<<		not		
			~		or		
			&		>		
			|		<		
					>=		
					<=		
					===		

Table 2 - Complexity Keywords List

5. Cyclomatic Complexity

Cyclomatic complexity measures the number of linearly independent paths through a program. It is measured for each function, procedure, or method according to each specific program language. This metric indicates the risk of program complexity and also determines the number of independent test required to verify program coverage.

The cyclomatic complexity is computed by counting the number of decisions plus one for the linear path. Decisions are determined by the number of conditional statements in a function. A function without any decisions would have a cyclomatic complexity of one. Each decision such as an if condition or a for loop adds one to the cyclomatic complexity.

The cyclomatic complexity metric v (G) was defined by Thomas McCabe. Several variations are commonly used but are not included in the UCC. The modified cyclomatic complexity counts select blocks as a single decision rather than counting each case. The strict or extended cyclomatic complexity includes boolean operators within conditional statements as additional decisions.

Table 3 – Cyclomatic Complexity Risk Evaluation

Cyclomatic Complexity	Risk Evaluation
1 - 10	A simple program, without much risk
11 - 20	More complex, moderate risk
21 - 50	Complex, high risk program
> 50	Untestable program, very high risk

For Ruby the following table lists the conditional keywords used to compute cyclomatic complexity.

Table 4 – Cyclomatic Complexity Counts

Statement	CC Count	Rationale
if	+1	if adds a decision
else if	+1	else if adds a decision
else	0	Decision is at the if statement
switch case	+1 per case	Each case adds a decision – not the

		switch
switch default	0	Decision is at the case statements
for	+1	for adds a decision at loop start
while	+1	while adds a decision at loop start or at end of do loop
do	0	Decision is at while statement – no decision at unconditional loop
try	0	Decision is at catch statement
catch	+1	catch adds a decision
ternary ? :	+1	Ternary ? adds a decision – : is similar to default or else