Input & output in Ruby

last modified October 18, 2023

In this part of the Ruby tutorial, we talk about input & output operations in Ruby. Input is any data that is read by the program, either from a keyboard, file or other programs. Output is data that is produced by the program. The output may go to the screen, to a file or to another program.

Input & output is a large topic. We bring forward some examples to give you a general idea of the subject. Several classes in Ruby have methods for doing input & output operations. For example Kernel, IO, Dir or File.

stones.txt

```
Garnet
Topaz
Opal
Amethyst
Ruby
Jasper
Pyrite
Malachite
Quartz
```

Some of the examples use this file.

Ruby writing to console

Ruby has several methods for printing output on the console. These methods are part of the Kernel module. Methods of the Kernel are available to all objects in Ruby.

printing.rb

```
#!/usr/bin/ruby

print "Apple "
```

```
print "Apple\n"

puts "Orange"
puts "Orange"
```

The print and puts methods produce textual output on the console. The difference between the two is that the latter adds a new line character.

```
print "Apple "
print "Apple\n"
```

The print method prints two consecutive "Apple" strings to the terminal. If we want to create a new line, we must explicitly include a newline character. The newline character is \n. Behind the scenes, the print method actually calls the to_s method of the object being printed.

```
puts "Orange"
puts "Orange"
```

The puts method prints two strings to the console. Each is on its own line. The method includes automatically the newline character.

```
$ ./printing.rb
Apple Apple
Orange
Orange
```

This the output of the printing.rb script file.

According to the Ruby documentation, the print method is an equivalent to the $stdout.print. The $stdout is a global variable which holds the standard output stream.

printing2.rb

```
#!/usr/bin/ruby

$stdout.print "Ruby language\n"
$stdout.puts "Python language"
```

We print two lines using the $stdout variable.

Ruby has another three methods for printing output.

printing3.rb

```
#!/usr/bin/ruby

p "Lemon"
p "Lemon"

printf "There are %d apples\n", 3

putc 'K'
putc 0xA
```

In the example, we present the p, printf and putc methods.

```
p "Lemon"
```

The p calls the inspect method upon the object being printed. The method is useful for debugging.

```
printf "There are %d apples\n", 3
```

The printf method is well known from the C programming language. It enables string formatting.

```
putc 'K'
putc 0xA
```

The putc method prints one character to the console. The second line prints a newline. The OxA is a hexadecimal code for the newline character.

```
$ ./printing3.rb
"Lemon"
"Lemon"
There are 3 apples
K
```

Printing data to the console using the kernel methods is a shortcut: a convenient way to print data. The following example shows a more formal way to print data to the terminal.

```
ios = IO.new STDOUT.fileno
ios.write "ZetCode\n"
ios.close
```

In the example, we open a standard output stream and write a string into it.

```
ios = IO.new STDOUT.fileno
```

The new method returns a stream to which we can write data. The method takes a numeric file descriptor. The STDOUT.fileno gives us the file descriptor for the standard output stream. We could also simply write 2.

```
ios.write "ZetCode\n"
```

We write a string to the opened stream.

```
ios.close
```

The input stream is closed.

On Unix systems the standard terminal output is connected to a special file called /dev/tty. By opening it and writing to it, we write to a console.

dev_tty.rb

```
#!/usr/bin/ruby

fd = IO.sysopen "/dev/tty", "w"
ios = IO.new(fd, "w")
ios.puts "ZetCode"
ios.close
```

A small example in which we write to a /dev/tty file. This only works on Unix.

```
fd = IO.sysopen "/dev/tty", "w"
```

The sysopen method opens the given path, returning the underlying file descriptor number.

```
ios = IO.new(fd, "w")
```

The file descriptor number is used to open a stream.

```
ios.puts "ZetCode"
ios.close
```

We write a string to the stream and close it.

Ruby reading input from console

In this section, we create some code examples that deal with reading from the console.

The $stdin is a global variable that holds a stream for the standard input. It can be used to read input from the console.

reading.rb

```
#!/usr/bin/ruby

inp = $stdin.read
puts inp
```

In the above code, we use the read method to read input from the console.

```
inp = $stdin.read
```

The read method reads data from the standard input until it reaches the end of the file. EOF is produced by pressing Ctrl+D on Unix and Ctrl+Z on Windows.

```
$ ./reading.rb
Ruby language
Ruby language
```

When we launch a program without a parameter, the script reads data from the user. It reads until we press Ctrl+D or Ctrl+Z.

```
$ echo "ZetCode" | ./reading.rb
ZetCode

$ ./input.rb < stones.txt
Garnet
Topaz
Opal
Amethyst
Ruby
Jasper
Pyrite
Malachite
Quartz
```

The script can read data from another program or a file if we do some redirections.

The common way to read data from the console is to use the gets method.

read_input.rb

```
#!/usr/bin/ruby

print "Enter your name: "
name = gets
puts "Hello #{name}"
```

We use the gets method to read a line from the user.

```
name = gets
```

The gets method reads a line from the standard input. The data is assigned to the name variable.

```
puts "Hello #{name}"
```

The data that we have read is printed to the console. We use interpolation to include the variable in the string.

```
$ ./read_input.rb
Enter your name: Jan
Hello Jan
```

This is a sample output.

In the following two scripts, we discuss the chomp method. It is a string method which removes white spaces from the end of the string. It is useful when doing input operations. The method name and usage comes from the Perl language.

no_chomp.rb

```
#!/usr/bin/ruby

print "Enter a string: "
inp = gets

puts "The string has #{inp.size} characters"
```

We read a string from a user and calculate the length of the input string.

```
$ ./no_chomp.rb
Enter a string: Ruby
The string has 5 characters
```

The message says that the string has 5 characters. It is because it counts the newline as well.

To get the correct answer, we need to remove the newline character. This is a job for the chomp method.

chomp.rb

```
#!/usr/bin/ruby

print "Enter a string: "
inp = gets.chomp

puts "The string has #{inp.size} characters"
```

This time we use we cut the newline character with the chomp method.

```
$ ./chomp.rb
Enter a string: Ruby
The string has 4 characters
```

The Ruby string has indeed 4 characters.

Ruby files

From the official Ruby documentation we learn that the IO class is the basis for all input and output in Ruby. The File class is the only subclass of the IO class. The two classes are closely related.

simple_write.rb

```
#!/usr/bin/ruby

f = File.open('output.txt', 'w')
f.puts "The Ruby tutorial"
f.close
```

In the first example, we open a file and write some data to it.

```
f = File.open('output.txt', 'w')
```

We open a file 'output.txt' in write mode. The open method returns an io stream.

```
f.puts "The Ruby tutorial"
```

We used the above opened stream to write some data. The puts method can be used to write data to a file as well.

```
f.close
```

At the end the stream is closed.

```
$ ./simple_write.rb
$ cat output.txt
The Ruby tutorial
```

We execute the script and show the contents of the output.txt file.

We have a similar example which shows additional methods in action.

simple_write2.rb

```
#!/usr/bin/ruby

File.open('langs.txt', 'w') do |f|

    f.puts "Ruby"
    f.write "Java\n"
    f << "Python\n"

end
```

If there is a block after the open method then Ruby passes the opened stream to this block. At the end of the block, the file is automatically closed.

```
f.puts "Ruby"
f.write "Java\n"
f << "Python\n"
```

We use three different methods to write to a file.

```
$ ./simple_write2.rb
$ cat langs.txt
Ruby
Java
Python
```

We execute the script and check the contents of the langs file.

In the second example, we show a few methods of the File class.

testfile.rb

```
#!/usr/bin/ruby

puts File.exist? 'tempfile'
```

```
f = File.new 'tempfile', 'w'
puts File.mtime 'tempfile'
puts f.size

File.rename 'tempfile', 'tempfile2'

f.close
```

The example creates a new file named tempfile and calls some methods.

```
puts File.exist? 'tempfile'
```

The exist? method checks if a file with a given name already exists. The line returns false, because we have not yet created the file.

```
f = File.new 'tempfile', 'w'
```

The file is created.

```
puts File.mtime 'tempfile'
```

The mtime method gives us the last modification time of the file.

```
puts f.size
```

We determine the file size. The method returns 0, since we have not written to the file.

```
File.rename 'tempfile', 'tempfile2'
```

Finally, we rename the file using the rename method.

```
$ ./testfile.rb
false
2020-09-14 15:54:13 +0200
0
```

This is a sample output.

Next, we read data from a file on the disk.

read_file.rb

```
#!/usr/bin/ruby

f = File.open("stones.txt")
```

```
while line = f.gets do
    puts line
end

f.close
```

The example opens a file called stones.txt and print its contents line by line to the terminal.

```
f = File.open("stones.txt")
```

We open a stones file. The default mode is a read mode. The stones file contains nine names of valued stones, each on a separate line.

```
while line = f.gets do
    puts line
end
```

The gets method reads a line from the I/O stream. The while block ends when we reach the end of file.

```
$ ./read_file.rb
Garnet
Topaz
Opal
Amethyst
Ruby
Jasper
Pyrite
Malachite
Quartz
```

The next example reads data from a file.

all_lines.rb

```
#!/usr/bin/ruby

fname = 'alllines.rb'

File.readlines(fname).each do |line|
    puts line
end
```

This script shows another way of reading a file's contents. The code example prints its own code to the terminal.

```
File.readlines(fname).each do |line|
    puts line
end
```

The readlines reads all lines from the specified file and returns them in the form of an array. We go through the array with the each method and print the lines to the terminal.

```
$ ./all_lines.rb
#!/usr/bin/ruby

fname = 'alllines.rb'

File.readlines(fname).each do |line|
    puts line
end
```

Ruby directories

In this section, we work with directories. We have a Dir class to work with directories in Ruby.

dirs.rb

```
#!/usr/bin/ruby

Dir.mkdir "tmp"
puts Dir.exist? "tmp"

puts Dir.pwd
Dir.chdir "tmp"
puts Dir.pwd

Dir.chdir '..'
puts Dir.pwd
Dir.rmdir "tmp"
puts Dir.exist? "tmp"
```

In the script we use four methods of the Dir class.

```
Dir.mkdir "tmp"
```

The mkdir method makes a new directory called tmp.

```
puts Dir.exist? "tmp"
```

With the exist? method, we check if a directory with a given name exists in the filesystem.

```
puts Dir.pwd
```

The pwd method prints a current working directory. This is the directory from which we launched the script.

```
Dir.chdir '..'
```

The chdir method changes to another directory. The .. directory is the parent directory of the current working directory.

```
Dir.rmdir "tmp"
puts Dir.exist? "tmp"
```

Finally, we remove a directory with the rmdir method. This time the exist? method returns false.

In the second example, we retrieve all of a directory's entries, including its files and subdirectories.

all_files.rb

```
#!/usr/bin/ruby

fls = Dir.entries '.'
puts fls.inspect
```

The entries method returns all entries of a given diretory.

```
fls = Dir.entries '.'
puts fls.inspect
```

We get the array of files and directories of a current directory. The . character stands for the current working directory in this context. The inspect method gives us a more readable representation of the array.

The third example works with a home directory. Every user in a computer has a unique directory assigned to him. It is called a home directory. It is a place where he can place his files and create his own hierarchy of directories.

home_dir.rb

```
#!/usr/bin/ruby
```

```
puts Dir.home
puts Dir.home 'root'
```

The script prints two home directories.

```
puts Dir.home
```

If we do not specify the user name, then a home directory of a current user is returned. The current user is the owner of the script file. Someone, who has launched the script.

```
puts Dir.home 'root'
```

Here we print the home directory of a specific user: in our case, the superuser.

```
$ ./homedir.rb
/home/janbodnar
/root
```

This is a sample output.

Ruby executing external programs

Ruby has several ways to execute external programs. We deal with some of them. In our examples we use well known Linux commands. Readers with Windows or Mac can use commands specific for their systems.

system.rb

```
#!/usr/bin/ruby

data = system 'ls'
puts data
```

We call a ls command which lists directory contents.

```
data = system 'ls'
```

The system command executes an external program in a subshell. The method belongs to the Kernel Ruby module.

We show two other ways of running external programs in Ruby.

system2.rb

```
#!/usr/bin/ruby

out = `pwd`
puts out

out = %x[uptime]
puts out

out = %x[ls | grep 'readline']
puts out
```

To run external programs we can use backticks `` or %x[] characters.

```
out = `pwd`
```

Here we execute the pwd command using backticks. The command returns the current working directory.

```
out = %x[uptime]
```

Here we get the output of the uptime command, which tells how long a system is running.

```
out = %x[ls | grep 'readline']
```

We can use also a combination of commands.

We can execute a command with the open method. The method belongs to the Kernel module. It creates an IO object connected to the given stream, file, or subprocess. If we want to connect to a subprocess, we start the path of the open with a pipe character |.

system3.rb

```
#!/usr/bin/ruby

f = open("|ls -l |head -3")
out = f.read
puts out
f.close

puts $?.success?
```

In the example, we print the outcome of the ls -l | head -3 commands. The combination of these two commands returns the first three lines of the ls -l command. We also check the status of the child subprocess.

```
f = open("|ls -l |head -3")
```

We connect to a subprocess, created by these two commands.

```
out = f.read
puts out
```

We read and print data from the subprocess.

```
f.close
```

We close the file handler.

```
puts $?.success?
```

The $? is a special Ruby variable that is set to the status of the last executed child process. The success? method returns true if the child process ended OK.

Ruby redirecting standard output

Ruby has predefined global variables for standard input, standard output and standard error output. The $stdout is the name of the variable for the standard output.

redirecting.rb

```
#!/usr/bin/ruby

$stdout = File.open "output.log", "a"

puts "Ruby"
puts "Java"

$stdout.close
$stdout = STDOUT

puts "Python"
```

In the above example, we redirect a standard output to the output.log file.

```
$stdout = File.open "output.log", "a"
```

This line creates a new standard ouput. The standard output will now flow to the ouput.log file. The file is opened in the append mode. If the file does not exist yet, it is created. Otherwise it is opened and the data is written at the end of the file.

```
puts "Ruby"
puts "Java"
```

We print two strings. The strings will not be shown in the terminal as usual. Rather, they will be appended to the output.log file.

```
$stdout.close
```

The handler is closed.

```
$stdout = STDOUT

puts "Python"
```

We use a predefined standard constant STDOUT to recreate the normal standard ouput. The "Python" string is printed to the console.

In this part of the Ruby tutorial, we worked with input and output operations in Ruby.

Contents Previous