Chapter 6: Reading and writing strings

Section 6.1: Printing and displaying strings

R has several built-in functions that can be used to print or display information, but `print` and `cat` are the most basic. As R is an interpreted language, you can try these out directly in the R console:

```
print("Hello World")
#[1] "Hello World"
cat("Hello World\n")
#Hello World
```

Note the difference in both input and output for the two functions. (Note: there are no quote-characters in the value of x created with `x <- "Hello World"`. They are added by `print` at the output stage.)

`cat` takes one or more character vectors as arguments and prints them to the console. If the character vector has a length greater than 1, arguments are separated by a space (by default):

```
cat(c("hello", "world", "\n"))
#hello world
```

Without the new-line character (`\n`) the output would be:

```
cat("Hello World")
#Hello World>
```

The prompt for the next command appears immediately after the output. (Some consoles such as RStudio's may automatically append a newline to strings that do not end with a newline.)

`print` is an example of a "generic" function, which means the class of the first argument passed is detected and a class-specific *method* is used to output. For a character vector like `"Hello World"`, the result is similar to the output of `cat`. However, the character string is quoted and a number `[1]` is output to indicate the first element of a character vector (In this case, the first and only element):

```
print("Hello World")
#[1] "Hello World"
```

This default print method is also what we see when we simply ask R to print a variable. Note how the output of typing `s` is the same as calling `print(s)` or `print("Hello World")`:

```
s <- "Hello World"
s
#[1] "Hello World"
```

Or even without assigning it to anything:

```
"Hello World"
#[1] "Hello World"
```

If we add another character string as a second element of the vector (using the `c()` function to **c**oncatenate the elements together), then the behavior of `print()` looks quite a bit different from that of `cat`:

```
print(c("Hello World", "Here I am."))
```

```
#[1] "Hello World" "Here I am."
```

Observe that the `c()` function does *not* do string-concatenation. (One needs to use `paste` for that purpose.) R shows that the character vector has two elements by quoting them separately. If we have a vector long enough to span multiple lines, R will print the index of the element starting each line, just as it prints `[1]` at the start of the first line.

```
c("Hello World", "Here I am!", "This next string is really long.")
#[1] "Hello World"                      "Here I am!"
#[3] "This next string is really long."
```

The particular behavior of `print` depends on the *class* of the object passed to the function.

If we call `print` an object with a different class, such as "numeric" or "logical", the quotes are omitted from the output to indicate we are dealing with an object that is not character class:

```
print(1)
#[1] 1
print(TRUE)
#[1] TRUE
```

Factor objects get printed in the same fashion as character variables which often creates ambiguity when console output is used to display objects in SO question bodies. It is rare to use `cat` or `print` except in an interactive context. Explicitly calling `print()` is particularly rare (unless you wanted to suppress the appearance of the quotes or view an object that is returned as `invisible` by a function), as entering `foo` at the console is a shortcut for `print(foo)`. The interactive console of R is known as a REPL, a "read-eval-print-loop". The `cat` function is best saved for special purposes (like writing output to an open file connection). Sometimes it is used inside functions (where calls to `print()` are suppressed), however **using `cat()` inside a function to generate output to the console is bad practice**. The preferred method is to `message()` or `warning()` for intermediate messages; they behave similarly to `cat` but can be optionally suppressed by the end user. The final result should simply returned so that the user can assign it to store it if necessary.

```
message("hello world")
#hello world
suppressMessages(message("hello world"))
```

Section 6.2: Capture output of operating system command

Functions which return a character vector

Base R has two functions for invoking a system command. Both require an additional parameter to capture the output of the system command.

```
system("top -a -b -n 1", intern = TRUE)
system2("top", "-a -b -n 1", stdout = TRUE)
```

Both return a character vector.

```
  [1] "top - 08:52:03 up 70 days, 15:09,  0 users,  load average: 0.00, 0.00, 0.00"
  [2] "Tasks: 125 total,   1 running, 124 sleeping,   0 stopped,   0 zombie"
  [3] "Cpu(s):  0.9%us,  0.3%sy,  0.0%ni, 98.7%id,  0.1%wa,  0.0%hi,  0.0%si,  0.0%st"
  [4] "Mem:  12194312k total,  3613292k used,  8581020k free,   216940k buffers"
  [5] "Swap: 12582908k total,  2334156k used, 10248752k free,  1682340k cached"
  [6] ""
  [7] "  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND           "
```

```
 [8] "11300 root      20   0 1278m 375m 3696 S  0.0  3.2 124:40.92 trala              "
 [9] " 6093 user1     20   0 1817m 269m 1888 S  0.0  2.3  12:17.96 R                  "
[10] " 4949 user2     20   0 1917m 214m 1888 S  0.0  1.8  11:16.73 R                  "
```

> For illustration, the UNIX command `top -a -b -n 1` is used. This is OS specific and may need to be amended to run the examples on your computer.

Package `devtools` has a function to run a system command and capture the output without an additional parameter. It also returns a character vector.

```
devtools::system_output("top", "-a -b -n 1")
```

Functions which return a data frame

The `fread` function in package `data.table` allows to execute a shell command and to read the output like **`read.table`**. It returns a `data.table` or a **`data.frame`**.

```
fread("top -a -b -n 1", check.names = TRUE)
       PID     USER PR NI  VIRT  RES  SHR S X.CPU X.MEM     TIME.         COMMAND
  1: 11300     root 20  0 1278m 375m 3696 S     0   3.2 124:40.92           trala
  2:  6093    user1 20  0 1817m 269m 1888 S     0   2.3  12:18.56               R
  3:  4949    user2 20  0 1917m 214m 1888 S     0   1.8  11:17.33               R
  4:  7922    user3 20  0 3094m 131m 1892 S     0   1.1  21:04.95               R
```

Note, that `fread` automatically has skipped the top 6 header lines.

> Here the parameter `check.names = TRUE` was added to convert `%CPU`, `%MEN`, and `TIME+` to syntactically valid column names.

Section 6.3: Reading from or writing to a file connection

Not always we have liberty to read from or write to a local system path. For example if R code streaming map-reduce must need to read and write to file connection. There can be other scenarios as well where one is going beyond local system and with advent of cloud and big data, this is becoming increasingly common. One of the way to do this is in logical sequence.

Establish a file connection to read with **`file()`** command ("r" is for read mode):

```
conn <- file("/path/example.data", "r") #when file is in local system
conn1 <- file("stdin", "r") #when just standard input/output for files are available
```

As this will establish just file connection, one can read the data from these file connections as follows:

```
line <- readLines(conn, n=1, warn=FALSE)
```

Here we are reading the data from file connection `conn` line by line as `n=1`. one can change value of `n` (say 10, 20 etc.) for reading data blocks for faster reading (10 or 20 lines block read in one go). To read complete file in one go set `n=-1`.

After data processing or say model execution; one can write the results back to file connection using many different commands like **`writeLines()`**,**`cat()`** etc. which are capable of writing to a file connection. However all of these commands will leverage file connection established for writing. This could be done using **`file()`** command as:

```
conn2 <- file("/path/result.data", "w") #when file is in local system
conn3 <- file("stdout", "w") #when just standard input/output for files are available
```

Then write the data as follows:

```
writeLines("text",conn2, sep = "\n")
```