input/output (i/o) in R

Input/Output (I/O) serves as a key component in most real-world applications of programming. R programming is not designed to be the outlet most suitable for implementing I/O applications in enterprises or large network structures. However, R employs a versatile set of tools for reading and writing files, navigating directories, and accessing the keyboard and monitor, the Internet, etc.

accessing the keyboad and monitor in R

R offers several functions for accessing the keyboard and monitor:

```
Access the keyboard and monitor in R:
scan()     readline()     print()     cat()
```

The **scan( )** function can be used to read into a vector (numeric or character), from a file or keyboard:

```
scan(file = "", what = double(), nmax = -1, n = -1, sep = "",
     quote = if(identical(sep, "\n")) "" else "'\"", dec = ".",)
```

The **scan( )** function reads files all at once, assuming vector elements are separated by ***whitespace***; including blanks, carriage return/line feeds, and horizontal tabs. The **sep =** argument can be used to specify the latter.

Additionally, the **scan( )** function defaults to reading data as mode (class) **"double"**, controlled by **what =**.

The **scan( )** function is also available for reading from the keyboard for individual entry (enter twice to esc.):

```
> scan()                    #use argument quiet = TRUE to prevent printing read items
1: 1
2: 2 3
4: 4 5 6 7 8 9 10
11:
Read 10 items
 [1]  1  2  3  4  5  6  7  8  9 10
```

The **readline( )** function can be used to read in a single line from the keyboard; often used for prompts:

```
> inits <- readline("type your initials: ")
type your initials: MR
> inits
[1] "MR"
```

At the top-level environment in R, objects can be printed by inputting the variable name and executing the code. Conversely, the latter does not work within the body of a function; the **print( )** function works instead.

```
> x <- 1:3
> print(x^2)
[1] 1 4 9
```

print( ) is a **generic function**; the call is dependent upon what class object the argument is. For example, if the object class passed through the **print( )** function is of the class "table", then **print.table( )** is called.

Because the **print( )** function can only output a single expression that is numbered, **cat( )** can be better:

```
> print("abc")
[1] "abc"

> cat("abc\n")          #need to specify an end-of-line character to signal break
abc
```

By default, the **cat( )** function prints with intervening spaces:

```
> x
[1] 1 2 3

> cat(x, "abc", "de\n")
1 2 3 abc de
```

The intermittent spaces can be prevented by the use of **sep = “”** argument:

```
> cat(x, "abc", "de\n", sep = "")
123abcde
```

Any string can be used as a value to the **sep =** argument as illustrated below:

```
> cat(x, "abc", "de\n", sep = "\n")          #separated by new line character
1
2
3
abc
de

> x <- c(5,12,13,8,88)
> cat(x, sep = c(".",".",".","\n","\n"))    #separated by a vector of characters
5.12.13.8
88
```

reading and writing files in R

The following material discusses:

- … reading dataframes and matrices from files
- … working with text files
- … accessing files on remote machines
- … obtaining file directory information

reading a dataframe or matrix from a file in R

Given file **z** with contents as follows, the file can be read including header information.

```
> z <- read.table("z.txt", header = TRUE)
> z
  Name Age
1 John  25
2 Mary  28
3  Jim  19
```

Note that the **scan( )** function will not work here due to the file having a mixture of ***numeric***, ***character***, and ***header*** information. A simple way to use the **scan( )** function would to read a file containing a matrix with the **byrow** argument.

```
> x <- matrix(scan("x.txt"),
nrow = 5, byrow = TRUE)
Read 15 items
> x
     [,1] [,2] [,3]
[1,]    1    0    1
[2,]    1    1    1
[3,]    1    1    0
[4,]    1    1    0
[5,]    0    0    1
```

```
> read.matrix <- function(filename) {
+     as.matrix(read.table(filename))
+ }
> read.matrix("x.txt")
     V1 V2 V3
[1,]  1  0  1
[2,]  1  1  1
[3,]  1  1  0
[4,]  1  1  0
[5,]  0  0  1
```

Note in the above illustration that reading a matrix is also available with **read.table( )**, converted to a matrix.

reading a text file 🅃 in R

When referring to ***text files***, some computer scientists distinguish between the latter and ***binary files***—every file is technically ***binary*** consisting of 0s and 1s. Therefore, this context refers to ***text files*** as mainly consisting of ASCII characters with the use of newline characters for human perception. Nontext files, like JPEG or .exe files are generally referred to as ***binary files***.

The **readlines( )** functions can be used to read in a ***text file***; one line at a time or in a single operation.

Given the following file **z** from before, without the header information, the files can be read at once:

```
> readLines("z.txt")              #each line is treated as a string, returning a vector
[1] "John 25" "Mary 28" "Jim 19"
```

In order to read the file one line at a time, a **connection** needs to be created first. A **connection** in R is a fundamental mechanism used in various I/O operations; used for file access in the following context. The **file( )**, **url( )**, or other R functions will create a connection.

```
> c <- file("z.txt", "r")                              #open the connection to z.txt
> c
description       class        mode        text      opened    can read    can write
    "z.txt"      "file"         "r"      "text"    "opened"       "yes"       "no"

> readLines(c, n=1)                                    #read the first line of z.txt
[1] "John 25"

> readLines(c, n=1)                                    #read the second line of z.txt
[1] "Mary 28"

> readLines(c, n=1)                                    #read the third line of z.txt
[1] "Jim 19"

> readLines(c, n=1)                          #returned the end of file (EOF) of z.txt
character(0)
```

Note that the connection had to be established for R to keep track of the index as it was read through. Furthermore, the **End of File (EOF)** can be detected in code:

```
> c <- file("z.txt", "r")
> while(TRUE) {
+     rl <- readLines(c, n=1)
+     if (length(rl) == 0) {
+         print("end of file")
+         break
+     } else print(rl)
+ }
[1] "John 25"
[1] "Mary 28"
[1] "Jim 19"
[1] "end of file"
```

```
> c <- file("z.txt", "r")

> readLines(c, n=2)
[1] "John 25" "Mary 28"


> seek(con = c, where = 0)
[1] 18


> readLines(c, n=1)
[1] "John 25"
```

Note that the read can be backtracked by using the **seek( )** function. The **where=0** argument in the **seek( )** function resets the position of the file pointer ***zero*** characters from the start of the file (at the beginning). The return of **16** represents the cursor position prior to the **seek( )** function was called. The **close( )** function will be used to inform the system that the file writing is complete and to write the results to disk.

```
> close(c)
Warning messages:
1: closing unused connection (z.txt)
```

accessing files on remote machines with urls in R

With R being a statistically based programming language, ***file reads*** are often more common than ***writes***. When ***writes*** are needed, the **write.table( )** works almost synonymously to the **read.table( )** function; the difference is *writing* to a dataframe, opposed to *reading* a dataframe.

```
> kids <- c("Jack", "Jill")
> ages <- c(12, 10)
> d <- data.frame(kids, ages, stringsAsFactors = FALSE)
> d
  kids ages
1 Jack   12
2 Jill   10

> write.table(d, "kids")          #writes dataframe d to file kids in working directory
```

The preceding illustration writes the dataframe **d** to a file named **"kids"** located in the working directory. The following illustration writes a **matrix** to a file by using arguments **row.names** and **col.names**.

```
> write.table(xc, "xnew", row.names = FALSE, col.names = FALSE)
```

Additionally, the **cat( )** function can be used in writing to files, appending to files, and writing multiple fields.

```
> cat("abc\n", file = "u")                  #using the cat() function to write to files
> cat("de\n", file = "u", append = TRUE)    #using the cat() function to append files

> cat(file = "v", 1, 2, "xyz\n")      #using the cat() function to write multiple fields
```

Lastly, the **writeLines( )** function can be used in writing to files; if a ***connection*** is to be used, the **"w"** is required to be specified, indicating the function is ***writing*** to a file, opposed to ***reading*** a file.

```
> c <- file("www", "w")                          #creates files www for writing ("w")
> writeLines(c("abc", "de", "f"), c)             #writes contents to the connection www
> close(c)                                       #closes open connection
```

obtaining file and directory information in R

Among the various function in R to obtain information about files and directories, setting permissions, etc:

Access directory and file information in R:	
file.info()	provides file size, creation time, directory-vs-ordinary file status, and various other set arguments in a character vector
dir()	returns a character vector listing the names of all files in the directory (first argument). if the option **recursive = true** argument is specified, the entire directory tree will be displayed from the root argument.
file.exists()	returns a boolean vector indicating whether the given files exists for each name in the first argument, a character vector
getwd() and setwd()	function to determine the current working directory; function to set/change the current working directory.
?files	returns all the file- and directory-related functions in R

accessing the internet 🌐 in R

R's **socket facilities** give programmers access to the Internet's TCP/IP protocol. For the context of the material discussed, the term ***network*** refers to a set of locally connected machines, without accessing the internet. The medium between machines in these circumstances is often an Ethernet connection. The ***Internet*** essentially connects ***networks*** together through ***routers***; special purpose computers for connecting two or more machines. All machines connected to the Internet obtain an **Internet Protocol (IP)** address. These numerically rooted numbers can be stored as characters that are translated by a **Domain Name Service (DNS)** provider. Beyond the **IP** address specifying the base location, a **port number** is necessary to allocate the information transmittal; for example, how program A or B on one machine connected to the internet specifies the destination on another machine's program A or B, also connected to the internet.

When A intends to deliver information to B, software is written to a ***socket***; a system call syntactically similar to a call that writes to a file. Within the latter call, A specifies the **IP** address and **port number** of B; In turn, B also writes responses to A in the latter socket. The ***connection*** between A and B at that point is somewhat of an "agreement" to exchange information, opposed to an actual physical connection.

Applications correspond to a ***client/server*** model. Programmers in R that write server programs must assign a port number above 1024 (Matloff, 247). When a server program crashes, a delay occurs before the port is reusable.

sockets 🔌 in R

Note that all bytes sent by A to B amidst an existing connection are synonymous to a ***single large message***. Therefore, if A relays one line of 10 characters and one line of 20 characters to B, A will consider the two separate lines, but the **TCP/IP** read 30 characters of an incomplete message. Parsing the single large message into lines can be achieved in R through some of the following functions:

Parsing character strings received through sockets in R:	
`readLines()` and `writeLines`	Allows programming as if **TCP/IP** is sending message line by line. An application that is naturally viewed in line find the functions useful.
`serialize()` and `unserialize()`	R objects, like matrices and function calls can be sent by converting to a character string by the sender; returned to object by the receiver.
`readBin()` and `writeBin()`	Used for sending data in ***binary form***. Noting the distinction between ***binary*** files and ***text*** files discussed earlier.

Each of the above functions operate within R ***connections***. The choice of function remains critical. A long vector will be more conveniently sent via the **serialize( )** and/or **unserialize( )** functions, but can sacrifice time consumption. The latter is caused by the initial conversion necessary to and from the character representations of the vector, but also because the character representation is typically much longer. The following lists two additional R ***socket*** functions:

Parsing character strings received through sockets in R:	
`socketConnection()`	Establishes an R connection through ***sockets***. The ***port number*** is specified through the **port =** argument; additionally stating whether a server or client is to be created. The latter is achieved through setting the **server =** argument to TRUE or FALSE, respectively. Client cases must also provide the servers **IP** address through the **host =** argument.
`socketSelect()`	Useful what a server is connected to multiple clients. The main **socklist =** argument consists of a list of connections; the return value is the sublist of connections with data primed for the server to read.