R – String Manipulation

R-String Manipulation

- Any value written within a pair of single quote or double quotes in R is treated as a string.
- Internally R stores every string within double quotes, even when you create them with single quote.
- Rules Applied in String Construction
  - The quotes at the beginning and end of a string should be both double quotes or both single quote. They can not be mixed.
  - Double quotes can be inserted into a string starting and ending with single quote.
  - Single quote can be inserted into a string starting and ending with double quotes.
  - Double quotes can not be inserted into a string starting and ending with double quotes.
  - Single quote can not be inserted into a string starting and ending with single quote.

R-Examples of Valid Strings

```
> a <- 'Start and end with single quote'
> print(a)

> b <- "Start and end with double quotes"
> print(b)

> c <- "single quote ' in between double quotes"
> print(c)

> d <- 'Double quotes " in between single quote'
> print(d)
```

R- Examples of Invalid Strings

```
> e <- 'Mixed quotes"
> print(e)

> f <- 'Single quote ' inside single quote'
> print(f)

> g <- "Double quotes " inside double quotes"
> print(g)
```

R- Concatenating Strings - paste() function

- Many strings in R are combined using the paste() function. It can take any number of arguments to be combined together.
- Syntax: `paste(..., sep = " ", collapse = NULL)`
- Parameters used are:
  - ... represents any number of arguments to be combined.
  - sep represents any separator between the arguments. It is optional.
  - collapse is used to eliminate the space in between two strings. But not the space within two words of one string.
- Example:

```
> a <- "Hello"; b <- 'How'; c <- "are you? "
> print(paste(a,b,c))       #"Hello How are you? "
> print(paste(a,b,c, sep = "-"))  # "Hello-How-are you? "
> print(paste(a,b,c, sep = "", collapse = "")) "HelloHoware you? "
```

R- format() function

- Numbers and strings can be formatted to a specific style using format() function.
- Syntax: `format(x, digits, nsmall, scientific, width, justify = c("left", "right", "centre", "none"))`
- Parameters used are:
  - x is the vector input.
  - digits is the total number of digits displayed.
  - nsmall is the minimum number of digits to the right of the decimal point.
  - scientific is set to TRUE to display scientific notation.
  - width indicates the minimum width to be displayed by padding blanks in the beginning.
  - justify is the display of the string to left, right or center.

R- format() function

- Examples:

```
# Total number of digits displayed. Last digit rounded off.
> result <- format(23.123456789, digts = 9)
> print(result)            # "23.1234568"

# Display numbers in scientific notation.
> result <- format(c(6, 13.14521), scientific = TRUE)
> print(result)            # "6.000000e+00" "1.314521e+01"

# The minimum number of digits to the right of the decimal point.
> result <- format(23.47, nsmall = 5)
> print(result)            # "23.47000"

# Format treats everything as a string.
> result <- format(6)
> print(result)            # "6"
```

R- format() function

- Example:

```
# Numbers are padded with blank in the beginning for width.
> result <- format(13.7, width = 6)
> print(result)            # " 13.7"

# Left justify strings.
> result <- format("Hello", width = 8, justify = "l")
> print(result)            # "Hello "

# Justify string with center.
> result <- format("Hello", width = 8, justify = "c")
> print(result)                  # " Hello "
```

R- nchar() function

- This function counts the number of characters including spaces in a string.
- Syntax: `nchar(x)`
- Parameters used are:
  - x is the vector input.
- Example:

```
> result <- nchar("Count the number of characters")

> print(result)         # 30
```

R-Changing the case of strings

- These functions change the case of characters of a string.
- Syntax: `toupper(x); tolower(x)`
- Parameters used are:
  - x is the vector input.
- Example

```
> result <- toupper("Changing To Upper")
> print(result)             # "CHANGING TO UPPER"

> result <- tolower("Changing To Lower")
> print(result)             # "changing to lower"
```

R- substring() function

- This function extracts parts of a String.
- Syntax: `substring(x, first, last)`
- Parameters used are:
  - x is the character vector input.
  - first is the position of the first character to be extracted.
  - last is the position of the last character to be extracted.
- Example

```
# Extract characters from 5th to 7th position.
> result <- substring("Extract", 5, 7)
> print(result)          # "act"
```

R- sprintf() function

- R comes with the sprintf() function that provides string formatting like in the C language.
- To be more precise, this function is a wrapper for the C library function of the same name.
- In many other programming languages, this type of printing is known as printf which stands for print formatting.
- The function sprintf() allows you to create strings as output using formatted data.
- Syntax:

```
> sprintf("%s is %f feet tall\n", "Sven", 7.1)
```

R- sprintf() function

- The 1st argument is a character vector of one element that contains the text to be formatted.
- Inside the text there are various percent symbols % followed by the letter s and letter f.
- Each % is referred to as a slot, which is basically a placeholder for a variable that will be formatted.
- The rest of the inputs passed to sprintf() are the values that will be used in each of the slots.
- The string in the previous example contains two slots, string and float type, %s and %f respectively, and the subsequent arguments are "Sven", and 7.1.
- Each number is used as a value for each slot. The letter s indicates that the formatted variable is specified as a string.

R- sprintf() function

- Most of the times we pass variables containing different values as follows:

```
> hours <- 8; mins1 <- 2; mins2 <- 5
> sprintf("I woke up at %s:%s%s a.m.", hours, mins1, mins2)
```

R- sprintf() function

- The string format %s is just one of a larger list of available formatting options. The most common formatting:

Notation	Description
%s	a string
%d	an integer
%0xd	an integer padded with x leading zeros
%f	decimal notation with six decimals
%.xf	floating point number with x digits after decimal point
%e	compact scientific notation, e in the exponent
%E	compact scientific notation, E in the exponent
%g	compact decimal or scientific notation (with e)

R- sprintf() Format Slot Syntax

- The full syntax for a format slot is defined by:

```
%[parameter][flags][width][.precision][length]type
```

- The percent symbol, %, indicates a placeholder or slot.
- The parameter field is an optional field that can take the value n$ in which n is the number of the variable to display, allowing the variables provided to be used multiple times, using varying format specifiers or in different orders.

```
> sprintf("The second number is %2$d, the first number is %1$d", 2, 1)
> [1] "The second number is 1, the first number is 2"
```

R- sprintf() Format Slot Syntax

- The flags field can be zero or more (in any order) of:
  - - (minus) Left-align the output of this placeholder.
  - + (plus) Prepends a plus for positive signed-numeric types.
  - ' ' (space) Prepends a space for positive signed-numeric types.
  - 0 (zero) When the 'width' option is specified, prepends zeros for numeric types.
  - # (hash) Alternate form:
    - for g and G types, trailing zeros are not removed.
    - for f, F, e, E, g, G types, the output always contain a decimal point.
    - for o, x, X types, the text 0, 0x, 0X, respectively, is prepended to non-zero numbers.

R- sprintf() Format Slot Syntax

- The width field is an optional field that you use to specify a minimum number of characters to output, and is typically used to pad fixed-width fields in tabulated output, where the fields would otherwise be smaller, although it does not cause truncation of oversized fields.

```
> sprintf("%*d", 5, 10)     # "   10"
```

- The precision field usually specifies a maximum limit on the output, depending on the particular formatting type.

```
> sprintf("%.*s", 3, "abcdef") #"abc"
```

R- sprintf() Format Slot Syntax

- The length field is also optional, and can be any of:
- The most important field is the type field.
  - %: Prints a literal % character (this type doesn't accept any flags, width, precision, length fields).
  - d, i: integer value as signed decimal number.
  - f: double value in normal fixed-point notation.
  - e, E: double value in standard form.
  - g, G: double value in either normal or exponential notation.
  - x, X: unsigned integer as a hexadecimal number. x uses lower case, while X uses upper case.
  - o: unsigned integer in octal notation.
  - s: null terminated string.
  - a, A: double value in hexadecimal notation

R- sprintf() - Basic Examples

- Consider a real fraction like 1/6; in R the default output of this fraction will be:

```
> 1 / 6                    # [1] 0.167
```

- Notice that 1/6 is printed with seven decimal digits.
- The number 1/6 is actually an irrational number and so the computer needs to round it to some number of decimal digits.
- We can modify the default printing format in several ways.
- One option is to display only six decimal digits with the %f option:

```
# print 6 decimals
> sprintf('%f', 1/6)        # [1] "0.166667"
```

R- sprintf() - Basic Examples

- But you can also specify a different number of decimal digits, say 3. This can be achieved specifying an option of %.3f:

```
# print 3 decimals
sprintf('%.3f', 1/6)        # [1] "0.167"
```

- The table below shows six different outputs for 1/6

Notation	Output
%s	0.166666666666667
%f	0.166667
%.3f	0.167
%e	1.666667e-01
%E	1.666667E-01
%g	0.166667

R- sprintf() - Basic Examples

- When working on data analysis projects, it is common to generate different files with similar names (e.g. either for creating images, or data files, or documents).
- Imagine that we need to generate the names of 3 data files (with .csv extension).
- All the files have the same prefix name but each of them has a different number: data01.csv, data02.csv, and data03.csv.
- Taking advantage of the vectorized nature of paste0():

```
> file_names <- paste0('data0', 1:3, '.csv')
> file_names  # "data01.csv" "data02.csv" "data03.csv"
```

- Now imagine that you need to generate 100 file names numbered from 01, 02, 03, to 100.
- A preferable solution is to use paste0() like in the approach of the previous example.

R- sprintf() - Basic Examples

- In this case however, you would need to create two separate vectors—one with numbers 01 to 09, and another one with numbers 10 to 100—and then concatenate them in one single vector:

```
> files1 <- paste0('data0', 1:9, '.csv')
> files2 <- paste0('data', 10:100, '.csv')
> file_names <- c(files1, files2)
```

- Instead of using paste0() to create two vectors, we can use sprintf() with the %0xd option to indicate that an integer should be padded with x leading zeros.
- For instance, the first nine file names can be generated as:

```
> sprintf('data%02d.csv', 1:9)
#> [1] "data01.csv" "data02.csv" "data03.csv" "data04.csv"
"data05.csv" "data06.csv" "data07.csv" "data08.csv"
"data09.csv"
```

R- sprintf() - Basic Examples

- To generate the 100 file names do:

```
> file_names <- sprintf('data%02d.csv', 1:100)
```

- The first nine elements in file_names will include a leading zero before the integer; the following elements will not include the leading zero.

R- sprintf(): Fahrenheit to Celsius

- This example involves working on a function to convert Fahrenheit degrees into Celsius degrees.
- The conversion formula is:

```
\[ Celsius = (Fahrenheit - 32) \times \frac{5}{9} \]
```

- We can define a simple function to_celsius() that takes one argument, temp, which is a number representing temperature in Fahrenheit degrees.
- This function will return the temperature in Celsius degrees:

```
> to_celsius <- function(temp = 1) {
  (temp - 32) * 5/9 }
```

- We can use to_celsius() as any other function in R. If we want to know how many Celsius degrees are 95 Fahrenheit degrees:

```
> to_celsius(95)            # 35
```

R- sprintf(): Fahrenheit to Celsius

- Let's create another function that not only computes the temperature conversion but also prints a more informative message, something like: 95 Fahrenheit degrees = 35 Celsius degrees.
- We'll name this function fahrenheit2celsius():

```
> fahrenheit2celsius <- function(temp = 1) {
  celsius <- to_celsius(temp)
  sprintf('%.2f Fahrenheit degrees = %.2f Celsius
       degrees', temp, celsius)}
```

- Notice that fahrenheit2celsius() makes use of to_celsius() to compute the Celsius degrees. And then sprintf() is used with the options %.2f to display the temperatures with two decimal digits.

```
> fahrenheit2celsius(95)
#> [1] "95.00 Fahrenheit degrees = 35.00 Celsius degrees"
> fahrenheit2celsius(50)
#> [1] "50.00 Fahrenheit degrees = 10.00 Celsius degrees"
```