Module 6

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 6 Study Guide and Deliverables

Topics: Strings and Data Wrangling

Readings:
- Lecture material

Assignments:
- Assignment 6 due **Tuesday, June 20 at 6:00 AM ET**
- Final Project due **Thursday, June 22 at 6:00 AM ET**
- Final Project Presentation due date TBD.
  - Presentation times will be set up with your facilitator in advance.

Course Evaluation:
- Course Evaluation opens on **Tuesday, June 13, at 10:00 AM ET and closes on Tuesday, June 20 at 11:59 PM ET.**
- Please complete the course evaluation. Your feedback is important to MET, as it helps us make improvements to the program and the course for future students.

Live Classrooms:
- **Wednesday, June 14 from 7:00-9:00 PM ET**
- One-hour open office with facilitator. Day and time will be provided by your facilitator.
- Live sessions will be recorded.

Strings and Regular Expressions

Introduction

R provides very basic support for manipulating strings, both for data preparation and data cleaning. String manipulation is quite often needed for text processing and analyzing data which has string attributes. The `stringr` package provides more robust support for most commonly used string operations. The `stringr` package can be installed through the *RStudio* menu options, or programmatically with the following statement, if it was not already installed before.

```
if (!is.element("stringr", installed.packages()[,"Package"]))
  install.packages("stringr", repos="http://cran.us.r-project.org",
                   dependencies = TRUE)
```

The `stringr` package should be loaded into the *R* session before accessing the string operations as shown below.

```
library(stringr)
```

Common String Operations

The following sections describe some of the commonly used string operations provided by the `stringr` package.

Joining Strings

The `str_c` function is used for joining multiple strings into a single string. If the function is provided a single vector, the function returns a vector of the same length with every value converted to a string (character base type).

```
> str_c(c(1,2))
[1] "1" "2"
```

If the `str_c` function has multiple arguments, the corresponding elements from each vector are joined together and returned as a vector as shown below.

```
> str_c(c("a", "b"), c(1,2), c("c", "d"))
[1] "a1c" "b2d"
```

The default separator is the empty string when the corresponding values are joined. An explicit value for the separator can also be specified as shown below. In that case, the corresponding values are joined together, separated by the specified value.

```
> str_c(c("a", "b"), c(1,2), c("c", "d"), sep = "-")
[1] "a-1-c" "b-2-d"
```

Similar to other vector operations, when the corresponding lengths are not the same, the values in the shorter vectors are repeated to accommodate the corresponding values in the longer vector(s), as shown in the following two examples.

```
> str_c(LETTERS, " is for", "...")
 [1] "A is for..." "B is for..." "C is for..." "D is for..." "E is for..."
 [6] "F is for..." "G is for..." "H is for..." "I is for..." "J is for..."
[11] "K is for..." "L is for..." "M is for..." "N is for..." "O is for..."
[16] "P is for..." "Q is for..." "R is for..." "S is for..." "T is for..."
[21] "U is for..." "V is for..." "W is for..." "X is for..." "Y is for..."
[26] "Z is for..."
> str_c(LETTERS, c(" is for", " for"), "...")
 [1] "A is for..." "B for..."    "C is for..." "D for..."    "E is for..."
 [6] "F for..."    "G is for..." "H for..."    "I is for..." "J for..."
[11] "K is for..." "L for..."    "M is for..." "N for..."    "O is for..."
[16] "P for..."    "Q is for..." "R for..."    "S is for..." "T for..."
[21] "U is for..." "V for..."    "W is for..." "X for..."    "Y is for..."
[26] "Z for..."
```

The vector `letters[-26]` provides the vector of letters from *a* through *y*, whereas the vector `letters[-1]` provides the vector of letters from *b* through *z*. Joining the corresponding values with the string *is before* in between results in the following pattern:

```
> str_c(letters[-26], " is before ", letters[-1])
 [1] "a is before b" "b is before c" "c is before d" "d is before e"
 [5] "e is before f" "f is before g" "g is before h" "h is before i"
 [9] "i is before j" "j is before k" "k is before l" "l is before m"
[13] "m is before n" "n is before o" "o is before p" "p is before q"
[17] "q is before r" "r is before s" "s is before t" "t is before u"
[21] "u is before v" "v is before w" "w is before x" "x is before y"
[25] "y is before z"
```

The input vectors can be collapsed into a single string using the `collapse` argument. The following examples show the results with a single vector, multiple vectors, and with both the `sep` and `collapse` values.

```
> str_c(c(1,2), collapse = "")
[1] "12"
> str_c(c("a", "b"), c(1,2), c("c", "d"), collapse=":")
[1] "a1c:b2d"
> str_c(c("a", "b"), c(1,2), c("c", "d"), sep = "-", collapse=":")
[1] "a-1-c:b-2-d"
```

The vector of letters can be collapsed into a single string as shown in the following two examples.

```
> str_c(letters, collapse = "")
[1] "abcdefghijklmnopqrstuvwxyz"
> str_c(letters, collapse = ":")
[1] "a:b:c:d:e:f:g:h:i:j:k:l:m:n:o:p:q:r:s:t:u:v:w:x:y:z"
```

The `str_flatten` takes a single argument and flattens it to a single string. The default `collapse` value is the empty string.

```
> str_flatten(letters)
[1] "abcdefghijklmnopqrstuvwxyz"
> str_flatten(letters, collapse = ":")
[1] "a:b:c:d:e:f:g:h:i:j:k:l:m:n:o:p:q:r:s:t:u:v:w:x:y:z"
```

Lengths of Strings

The `str_length` function returns the length of the input values, coerced as strings if needed, in the given vector.

```
> str_length(c("a", "b", "c"))
[1] 1 1 1
> str_length(c("a1", "b23", "c456"))
[1] 2 3 4
```

The length of a missing value is reported as missing length.

```
> str_length(c("a1", NA, "c456"))
[1]  2 NA  4
```

Substrings of Strings

The `str_sub` function extracts substrings from the given input string vector. The *start* and *end* values may optionally specify the portion of the string to be extracted. The default value for *start* is 1, the beginning index of the string. The default value for *end* is -1, the last index of the string. The following example shows extracting the first 6 characters of the specified string.

```
> s <- "United States"
> str_length(s)
[1] 13
> str_sub(s, 1, 6)
[1] "United"
> str_sub(s, end = 6)
[1] "United"
```

The following example shows how to extract the substring *nite* from the given string.

```
> str_sub(s, start = 2, end = 5)
[1] "nite"
```

The following example shows two different ways of extracting the second word, *States*, from the given string. In the second case, the default *end* value -1 is used for the end of the string.

```
> str_sub(s, 8, 13)
[1] "States"
> str_sub(s, 8)
[1] "States"
```

If a vector of values is specified for the beginning and end positions, a resulting vector of the substrings is produced. The *start* argument provides the two beginning indices and the *end* argument provides the two ending indices. As a result, two substrings are extracted with the corresponding *start* and *end* values from each vector.

```
> str_sub(s, c(1, 8), c(6, 13))
[1] "United" "States"
> str_sub(s, start = c(1, 8), end = c(6, 13))
[1] "United" "States"
```

In the following example, the default *end* index of -1 is used for both the substrings.

```
> str_sub(s, c(1, 8))
[1] "United States" "States"
```

In the following example, the default *start* index of 1 is used for both the substrings.

```
> str_sub(s, end = c(6, 13))
[1] "United"        "United States"
```

A negative value for any index is interpreted as the position counting backwards from the end of the string, -1 being the last index. The following examples provide the *start* index value as a negative value. The default *end* index, -1, is used.

```
> str_sub(s, -1)
[1] "s"
> str_sub(s, -6)
[1] "States"
```

The following example shows the resulting string, *ted*, when both the beginning and ending indices are provided with negative values.

```
> str_sub(s, -10, -8)
[1] "ted"
```

The following example assumes the default start index of 1.

```
> str_sub(s, end = -6)
[1] "United S"
```

The `str_sub` function may also be used for replacing substrings from the given character vector. The following example shows the replacement of the first character in the given string, resulting in changing the given string

from *CLAP* to *FLAP.*

```
> x <- "CLAP"
> str_sub(x, 1, 1) <- "F"; x
[1] "FLAP"
```

The following examples show the replacement of the last character, changing the current string from *FLAP* to *FLAT*, and then a second replacement that changes the string from *FLAT* to *FOOT*. In the latter case, the target substring being replaced starts at index 2 and ends at index -2.

```
> str_sub(x, -1, -1) <- "T"; x
[1] "FLAT"
> str_sub(x, 2, -2) <- "OO"; x
[1] "FOOT"
```

The following example only retains the first and last character of the given string, replacing everything else in between with the empty string.

```
> str_sub(x, 2, -2) <- ""; x
[1] "FT"
```

Duplication of Strings

The `str_dup` function allows for duplication of strings to be duplicated the specified number of times. In the simplest case, with a single value of *2* for the times to duplicate, each string in the input vector is duplicated twice, as shown below.

```
> x <- c("a1", "b2", "c3")
> str_dup(x, 2)
[1] "a1a1" "b2b2" "c3c3"
```

If the times to duplicate is a vector of values, the corresponding string in the input vector is duplicated the respective number of times, as shown below.

```
> str_dup(x, 1:3)
[1] "a1"      "b2b2"    "c3c3c3"
```

The following example concatenates the string *a* in front of each of the duplicated *ha* strings.

```
> str_c("a", str_dup("ha", 0:4))
[1] "a"         "aha"       "ahaha"     "ahahaha"   "ahahahaha"
```

Trimming of Strings

The `str_trim` function removes white spaces from the ends of the input strings. The default behavior of the function is to remove both from the left and the right, if there are any.

```
> x <- "    How     are \n you?\t"
> x
[1] "    How     are \n you?\t"
> str_trim(x)
[1] "How     are \n you?"
```

The optional argument `side` can be set to either *left* or *right* only to remove the white spaces from those respective sides, as shown below.

```
> str_trim(x, side="left")
[1] "How     are \n you?\t"
> str_trim(x, side="right")
[1] "    How     are \n you?"
```

The `str_squish` function removes the white spaces from the ends of the string and also collapses consecutive white space character into a single space within the string.

```
> x <- "    How     are \n you?\t"
> str_squish(x)
[1] "How are you?"
> x <- "\t   How\t\t  \t\tare\tyou?"
> str_squish(x)
[1] "How are you?"
```

Padding and Truncation of Strings

The `str_pad` function may be used to pad the given strings to the given width. The default padding character is the space and padded on the left when the *side* is not specified explicitly. In the following example, the length of the resulting string will be 10 with 5 spaces padded on the left.

```
> str_pad("cs544", 10)
[1] "     cs544"
```

A single padding character can also be explicitly specified as shown below. The specified padding character is padded on the left.

```
> str_pad("cs544", 10, pad = "_")
[1] "_____cs544"
```

The operations are vectorized when multiple strings and/or padding characters are specified as shown below.

```
> str_pad(c("a", "abc", "abcdef"), 5)
[1] "    a"  "  abc"  "abcdef"
> str_pad("a", c(2, 4, 6))
[1] " a"     "   a"    "     a"
```

```
> str_pad("cs544", 10, pad = c("_", "#"))
[1] "_____cs544" "#####cs544"
> str_pad(c("cs544", "cs555"), 10, pad = c("_", "#"))
[1] "_____cs544" "#####cs555"
```

If the size is less than the length of the input string, the string is left unchanged.

```
> str_pad("cs544", 3)
[1] "cs544"
```

The `str_trunc` function is used for truncating the given string. The default string that replaces the truncated content on the specified side is the ellipsis …

```
> x <- "Foundations of Data Analytics with R";x
[1] "Foundations of Data Analytics with R"
> str_length(x)
[1] 36
```

```
> rbind(
    str_trunc(x, 25, "left"),
    str_trunc(x, 25, "right"),
    str_trunc(x, 25, "center")
  )
     [,1]
[1,] "... Data Analytics with R"
[2,] "Foundations of Data An..."
[3,] "Foundations...tics with R"
```

Detecting Patterns in Strings

The `str_detect` function is useful for detecting if the specified pattern is present or not in the given input string. The `fruit` dataset is a vector of 80 fruit names available with the `stringr` library.

```
> head(fruit)
[1] "apple"       "apricot"     "avocado"     "banana"      "bell pepper"
[6] "bilberry"
> tail(fruit)
[1] "star fruit" "strawberry" "tamarillo"  "tangerine"  "ugli fruit"
[6] "watermelon"
> length(fruit)
[1] 80
```

The `str_detect` function returns `TRUE` or `FALSE` for each presence or absence of the pattern. The logical results can then be indexed to select the detected values. The following example shows all the fruits which have the pattern *ap* in them.

```
> fruit[str_detect(fruit, "ap")]
[1] "apple"      "apricot"    "grape"      "grapefruit" "papaya"
[6] "pineapple"
```

The regular expression pattern *^ap* specifies that the matched value should *begin with* the value *ap*. The result is the two fruits that match as shown below.

```
> fruit[str_detect(fruit, "^ap")]
[1] "apple"   "apricot"
```

The pattern `it$` can be used to match all the fruits that *end with* the specified value *it*. This results in the 8 fruits that end with *it* as shown below.

```
> fruit[str_detect(fruit, "it$")]
[1] "breadfruit"   "dragonfruit"  "grapefruit"   "jackfruit"    "kiwi fruit"
[6] "passionfruit" "star fruit"   "ugli fruit"
```

The following example shows all fruits that either have a *d* or a *v* or a *w* in them.

```
> fruit[str_detect(fruit, "[dvw]")]
 [1] "avocado"      "blood orange" "breadfruit"   "cloudberry"
 [5] "damson"       "date"         "dragonfruit"  "durian"
 [9] "elderberry"   "guava"        "honeydew"     "kiwi fruit"
[13] "mandarine"    "olive"        "redcurrant"   "strawberry"
[17] "watermelon"
```

The following example shows all fruits that have a white space in them.

```
> fruit[str_detect(fruit, "[:space:]")]
 [1] "bell pepper"       "blood orange"      "canary melon"
 [4] "chili pepper"      "goji berry"        "kiwi fruit"
 [7] "purple mangosteen" "rock melon"        "salal berry"
[10] "star fruit"        "ugli fruit"
```

Matching Patterns in Strings

Pattern matching is an important step in text data processing. Typical searches involve looking for patterns in the data and finding those patterns. For the following samples, the data shown with 4 strings are used for illustrating the pattern matching functions.

```
> data <- c(
    "123 Main St",
    "6175551234",
    "978-356-1234",
    "Work: 617-423-4567; Home: 508.555.3589; Cell: 555 777 3456"
  )
```

The goal in this example is to look for phone numbers in the input strings. The phone numbers can be found with varying formats as in the example above. A regular expression can be used to capture these varying patterns. The following regular expression states that the phone number starts with a digit (2 to 9), followed by two digits (0 to 9), followed by an optional hyphen, space, or a period, followed by three digits (0 to 9), followed by an optional hyphen, space, or a period, and then four digits (0 to 9).

```
> phone <- "([2-9][0-9]{2})([- .]?)([0-9]{3})([- .])?([0-9]{4})"
```

The `str_detect` function can be used to check if the string has the pattern or not as shown below.

```
> str_detect(data, phone)
[1] FALSE  TRUE  TRUE  TRUE
```

The above result can be used as a logical index into the data to display the matched inputs.

```
> data[str_detect(data, phone)]
[1] "6175551234"
[2] "978-356-1234"
[3] "Work: 617-423-4567; Home: 508.555.3589; Cell: 555 777 3456"
```

The `str_subset` function can also be used to directly return the matched strings as shown below.

```
> str_subset(data, phone)
[1] "6175551234"
[2] "978-356-1234"
[3] "Work: 617-423-4567; Home: 508.555.3589; Cell: 555 777 3456"
```

The `str_locate` function is useful for locating the positions of the pattern in the given string. The start and end indices of only the first match in the input strings are returned as a matrix.

```
> str_locate(data, phone)
     start end
[1,]    NA  NA
[2,]     1  10
[3,]     1  12
[4,]     7  18
```

The `str_extract` function can be used for extracting the first matched pattern in the input strings as shown below.

```
> str_extract(data, phone)
[1] NA             "6175551234"   "978-356-1234" "617-423-4567"
```

For locating all occurrences of the pattern in the input strings, the `str_locate_all` function returns a list of matrices as shown below. The last input string in the data has three matches.

```
> str_locate_all(data, phone)
[[1]]
     start end

[[2]]
     start end
[1,]     1  10

[[3]]
     start end
[1,]     1  12

[[4]]
     start end
[1,]     7  18
[2,]    27  38
[3,]    47  58
```

The `str_extract_all` function can be used for extracting all the matched patterns as shown below. The result is a list of character vectors.

```
> str_extract_all(data, phone)
[[1]]
character(0)

[[2]]
[1] "6175551234"

[[3]]
[1] "978-356-1234"

[[4]]
[1] "617-423-4567" "508.555.3589" "555 777 3456"
```

With the `simplify` option, the same function returns all matches as a matrix as shown below. The number of columns will correspond to the maximum number of matches in any of the input data.

```
> str_extract_all(data, phone, simplify = TRUE)
     [,1]           [,2]           [,3]
[1,] ""             ""             ""
[2,] "6175551234"   ""             ""
[3,] "978-356-1234" ""             ""
[4,] "617-423-4567" "508.555.3589" "555 777 3456"
```

The above functions were useful for extracting the matched pattern. The regular expression used for the phone numbers has the round brackets or parenthesis which can be used to identify the corresponding part of the matched pattern. The pattern in the example has five such groups underlined as shown below.

```
> phone <- "([2-9][0-9]{2})([- .]?)([0-9]{3})([- .])?([0-9]{4})"
```

The `str_match` function returns a matrix with the matched groups for the first match in the input strings. The first column shows the match and the next five columns show the corresponding group of the pattern.

```
> str_match(data, phone)
     [,1]           [,2]  [,3] [,4]  [,5] [,6]
[1,] NA             NA    NA   NA    NA   NA
[2,] "6175551234"   "617" ""   "555" NA   "1234"
[3,] "978-356-1234" "978" "-"  "356" "-"  "1234"
[4,] "617-423-4567" "617" "-"  "423" "-"  "4567"
```

The `str_match_all` returns a list of matrices showing all the matches in the input strings as shown below.

```
> str_match_all(data, phone)
[[1]]
     [,1] [,2] [,3] [,4] [,5] [,6]

[[2]]
     [,1]           [,2]  [,3] [,4]  [,5] [,6]
[1,] "6175551234" "617" ""   "555" NA   "1234"

[[3]]
     [,1]           [,2]  [,3] [,4]  [,5] [,6]
[1,] "978-356-1234" "978" "-"  "356" "-"  "1234"

[[4]]
     [,1]           [,2]  [,3] [,4]  [,5] [,6]
[1,] "617-423-4567" "617" "-"  "423" "-"  "4567"
[2,] "508.555.3589" "508" "."  "555" "."  "3589"
[3,] "555 777 3456" "555" " "  "777" " "  "3456"
```

When publishing data, some of the information is redacted for public use. In such cases, the `str_replace` can be used as in the following example to replace the first matched phone number in each input string with the specified string.

```
> str_replace(data, phone, "XXX-XXX-XXXX")
[1] "123 Main St"
[2] "XXX-XXX-XXXX"
[3] "XXX-XXX-XXXX"
[4] "Work: XXX-XXX-XXXX; Home: 508.555.3589; Cell: 555 777 3456"
```

All the matched patterns in each string can be replaced with the help of the `str_replace_all` function as shown below.

```
> str_replace_all(data, phone, "XXX-XXX-XXXX")
[1] "123 Main St"
[2] "XXX-XXX-XXXX"
[3] "XXX-XXX-XXXX"
[4] "Work: XXX-XXX-XXXX; Home: XXX-XXX-XXXX; Cell: XXX-XXX-XXXX"
```

The `str_remove` and `str_remove_all` are wrappers for the `str_replace` and `str_replace_all` functions and can also be used in their place instead.

Counting Matches in Strings

The `str_count` function returns the number of matches of the pattern in the input strings. For the data used in the previous section, the `str_count` function returns a vector of the number of the matches as shown below.

```
> data <- c(
    "123 Main St",
    "6175551234",
    "978-356-1234",
    "Work: 617-423-4567; Home: 508.555.3589; Cell: 555 777 3456"
  )
> phone <- "([2-9][0-9]{2})([- .]?)([0-9]{3})([- .])?([0-9]{4})"

> str_count(data, phone)
[1] 0 1 1 3
```

The `str_count` function can also be used for counting the number of sentences, the number of words, or the number of characters in the input string as shown below.

```
> x <- "Hello, how are you? I am fine, thank you."
> str_length(x)
[1] 41

> str_count(x, boundary("sentence"))
[1] 2
> str_count(x, boundary("word"))
[1] 9
> str_count(x, boundary("character"))
[1] 41
```

If the input is a vector of strings, the `str_count` function returns the number of matches in each input string as shown below.

```
> y <- c("Hello, how are you? I am fine, thank you.", "Good bye!")
> str_length(y)
[1] 41  9

> str_count(y, boundary("sentence"))
[1] 2 1
> str_count(y, boundary("word"))
[1] 9 2
> str_count(y, boundary("character"))
[1] 41  9
```

Splitting of Strings

The `str_split` function splits the input string into parts over the specified pattern. Typically sentence, word, and character boundaries are useful for splitting the given string. The following examples show how two input strings are split at the *sentence* boundary. The first input string has two sentences, whereas the second input string has only one sentence. By default, the results are returned as a list of character vectors.

```
> x <- c("Hello, how are you? I am fine, thank you.", "Good bye!")
> str_split(x, boundary("sentence"))
[[1]]
[1] "Hello, how are you? "  "I am fine, thank you."

[[2]]
[1] "Good bye!"
```

The `simplify` option with the `TRUE` value returns the same results as a matrix as shown below.

```
> str_split(x, boundary("sentence"), simplify = TRUE)
     [,1]                      [,2]
[1,] "Hello, how are you? " "I am fine, thank you."
[2,] "Good bye!"             ""
```

The input strings can be split at the *word* boundary as shown below.

```
> str_split(x, boundary("word"))
[[1]]
[1] "Hello" "how"   "are"   "you"   "I"     "am"    "fine"  "thank"
[9] "you"

[[2]]
[1] "Good" "bye"
```

The number of pieces that an input string is split into can be limited by optionally providing a value for the *n* argument. In the following case, a maximum of 4 splits are allowed in each sentence at the *word* boundary.

```
> str_split(x, boundary("word"), n = 4)
[[1]]
[1] "Hello"                      "how"
[3] "are"                        "you? I am fine, thank you."

[[2]]
[1] "Good" "bye"
```

The input strings can also be split at the *character* boundary as shown below. Each individual character will be a part of the vector of splits.

```
> str_split(x, boundary("character"))
[[1]]
 [1] "H" "e" "l" "l" "o" "," " " "h" "o" "w" " " "a" "r" "e" " " "y"
[17] "o" "u" "?" " " "I" " " "a" "m" " " "f" "i" "n" "e" "," " " "t"
[33] "h" "a" "n" "k" " " "y" "o" "u" "."

[[2]]
[1] "G" "o" "o" "d" " " "b" "y" "e" "!"
```

References

Introduction to *stringr*.

Regular expressions.

Data Wrangling

Introduction

The `tidyverse` is a collection of R packages useful for data preparation, data cleaning, and data transformations falling under the broader scope of data wrangling. The lecture explored the `tibble`, `dplyr`, and `tidyr` packages from this collection. These packages can be installed individually, or as a collection. The easiest way is install the entire collection of these packages using the following statement.

```
> install.packages("tidyverse")
```

After the installation, the packages can be loaded as shown below.

```
> library(tidyverse)
```

The data set used in this lecture comes from the `nycflights13` package which contains information about the flights departing from the three New York City area airports. The package is installed as shown below.

```
> install.packages("nycflights13")
```

After the installation, the package can be loaded as shown below.

```
> library(nycflights13)
```

Data Frame Alternative - *tibble*

A *tibble* is an inherited version of the data frame that is more suitable and easier to handle than the data frame. A tibble with two columns of data can be created as shown below. By default, the tibble displays the first 10 rows of data and leaves a comment on how many more rows are there.

```
> a <- 1:100
>
> tibble(a, b = 2 * a)
# A tibble: 100 x 2
       a     b
   <int> <dbl>
 1     1     2
 2     2     4
 3     3     6
 4     4     8
 5     5    10
 6     6    12
 7     7    14
 8     8    16
 9     9    18
10    10    20
# ... with 90 more rows
```

A tibble can also refer to the new columns on the left while creating more columns on the right. In the following example, the column *c* refers to the new column *b*.

```
> tibble(a, b = 2 * a, c = b^2)
# A tibble: 100 x 3
       a     b     c
   <int> <dbl> <dbl>
 1     1     2     4
 2     2     4    16
 3     3     6    36
 4     4     8    64
 5     5    10   100
 6     6    12   144
 7     7    14   196
 8     8    16   256
 9     9    18   324
10    10    20   400
# ... with 90 more rows
```

The above functionality is not possible with a `data.frame` for creating the same data in one step.

```
> data.frame(a, b = 2 * a, c = b^2)
Error in data.frame(a, b = 2 * a, c = b^2) : object 'b' not found
```

Existing data frames can be converted to tibbles as shown below.

```
> as_tibble(iris)
# A tibble: 150 x 5
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          <dbl>       <dbl>        <dbl>       <dbl>  <fctr>
 1          5.1         3.5          1.4         0.2  setosa
 2          4.9         3.0          1.4         0.2  setosa
 3          4.7         3.2          1.3         0.2  setosa
 4          4.6         3.1          1.5         0.2  setosa
 5          5.0         3.6          1.4         0.2  setosa
 6          5.4         3.9          1.7         0.4  setosa
 7          4.6         3.4          1.4         0.3  setosa
 8          5.0         3.4          1.5         0.2  setosa
 9          4.4         2.9          1.4         0.2  setosa
10          4.9         3.1          1.5         0.1  setosa
# ... with 140 more rows
```

The `flights` dataset from the `nycflights13` package is a tibble with 19 columns of data. When a tibble is displayed, only the columns that fit in the window are displayed and in the comments shows the names of the rest of the columns.

```
> nycflights13::flights
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
 7  2013     1     1      555            600        -5      913
 8  2013     1     1      557            600        -3      709
 9  2013     1     1      557            600        -3      838
10  2013     1     1      558            600        -2      753
# ... with 336,766 more rows, and 12 more variables:
#   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

The `tribble` function allows the tibbles to be created by specifying the data row-wise as shown below.

```
> athlete.info <- tribble(
    ~Name,        ~Salary, ~Endorsements, ~Sport,
    "Mayweather",  105,      0,            "Boxing",
    "Ronaldo",      52,     28,            "Soccer",
    "James",      19.3,     53,            "Basketball",
    "Messi",      41.7,     23,            "Soccer",
    "Bryant",     30.5,     31,            "Basketball"
  )
```

```
> athlete.info
# A tibble: 5 x 4
        Name Salary Endorsements       Sport
       <chr>  <dbl>         <dbl>       <chr>
1 Mayweather  105.0             0      Boxing
2    Ronaldo   52.0            28      Soccer
3      James   19.3            53  Basketball
4      Messi   41.7            23      Soccer
5     Bryant   30.5            31  Basketball
```

The `glimpse` function provides a sneak peek of the data by printing the number of rows, the number of columns, a row of data for each column as it fits the printing area of the window.

```
> glimpse(athlete.info)
Observations: 5
Variables: 4
$ Name         <chr> "Mayweather", "Ronaldo", "James", "Messi", "...
$ Salary       <dbl> 105.0, 52.0, 19.3, 41.7, 30.5
$ Endorsements <dbl> 0, 28, 53, 23, 31
$ Sport        <chr> "Boxing", "Soccer", "Basketball", "Soccer", ...
```

```
> glimpse(nycflights13::flights)
Observations: 336,776
Variables: 19
$ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, ...
$ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
$ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
$ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 55...
$ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 60...
$ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -...
$ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 8...
$ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 8...
$ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, ...
$ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", ...
$ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 570...
$ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N...
$ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", ...
$ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", ...
$ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140...
$ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 22...
$ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, ...
$ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0...
$ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00,...
```

The pipe operator (`%>%`) is useful for chaining the functions. The data on the left-hand side of the pipe is passed as the first argument for the function on the right-hand side. The following example shows the summary of the data set without the pipe and with the pipe operator.

```
> summary(athlete.info)
     Name               Salary       Endorsements    Sport
 Length:5           Min.   : 19.3   Min.   : 0    Length:5
 Class :character   1st Qu.: 30.5   1st Qu.:23    Class :character
 Mode  :character   Median : 41.7   Median :28    Mode  :character
                    Mean   : 49.7   Mean   :27
                    3rd Qu.: 52.0   3rd Qu.:31
                    Max.   :105.0   Max.   :53
```

```
> athlete.info %>% summary
     Name                Salary        Endorsements    Sport
 Length:5           Min.   : 19.3   Min.   : 0    Length:5
 Class :character   1st Qu.: 30.5   1st Qu.:23    Class :character
 Mode  :character   Median : 41.7   Median :28    Mode  :character
                    Mean   : 49.7   Mean   :27
                    3rd Qu.: 52.0   3rd Qu.:31
                    Max.   :105.0   Max.   :53
```

The `is.tibble` or `is_tibble` functions can be used to check if the dataset is a tibble or not.

```
> is.tibble(iris)
[1] FALSE
> is_tibble(athlete.info)
[1] TRUE
```

Data Transformations with dplyr

The `dplyr` package provides the functions that are most commonly needed for data preprocessing, transformations, and manipulations. The following are some of the frequently encountered functions from this package:

- filter – extract existing observations by their values
- arrange – reorder the rows of the data
- select – pick existing variables by their names
- mutate – create new variables from existing variables
- summarize – used for summarizing grouped data

Filtering Rows

The `filter` function selects the rows from the dataset that match the specified conditions involving the variables in the data. The following example filters all rows for the month of April.

```
> filter(flights, month == 4)
# A tibble: 28,330 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     4     1      454            500     -6.00      636
 2  2013     4     1      509            515     -6.00      743
 3  2013     4     1      526            530     -4.00      812
 4  2013     4     1      534            540     -6.00      833
 5  2013     4     1      542            545     -3.00      914
 6  2013     4     1      543            545     -2.00      921
 7  2013     4     1      551            600     -9.00      748
 8  2013     4     1      552            600     -8.00      641
 9  2013     4     1      553            600     -7.00      725
10  2013     4     1      554            600     -6.00      752
# ... with 28,320 more rows, and 12 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

The following example specifies conditions involving multiple variables. The filter results in all rows of the data set for the 10th day of April.

```
> filter(flights, month == 4, day == 10)
# A tibble: 989 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     4    10        1           1930       271      106
 2  2013     4    10       18           2059       199      130
 3  2013     4    10       25           1900       325      136
 4  2013     4    10       25           1905       320      136
 5  2013     4    10       28           2030       238      228
 6  2013     4    10       28           2004       264      302
 7  2013     4    10       32           2000       272      241
 8  2013     4    10       38           2104       214      311
 9  2013     4    10       41           2040       241      355
10  2013     4    10       51           1950       301      330
# ... with 979 more rows, and 12 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

Multiple arguments for the `filter` function are interpreted as the *and* operator. Other Boolean operators can also be used as in the following example to filter all rows for the month of June or July.

```
> filter(flights, month == 6 | month == 7)
# A tibble: 57,668 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     6     1        2           2359      3.00      341
 2  2013     6     1      451            500     - 9.00      624
 3  2013     6     1      506            515     - 9.00      715
 4  2013     6     1      534            545    -11.0        800
 5  2013     6     1      538            545     - 7.00      925
 6  2013     6     1      539            540     - 1.00      832
 7  2013     6     1      546            600    -14.0        850
 8  2013     6     1      551            600     - 9.00      828
 9  2013     6     1      552            600     - 8.00      647
10  2013     6     1      553            600     - 7.00      700
# ... with 57,658 more rows, and 12 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>

> filter(flights, month == 6 | month == 7) %>% tail()
# A tibble: 6 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
1  2013     7    31     2346           2305      41.0       38             13
2  2013     7    31     2352           2245      67.0       49           2359
3  2013     7    31       NA            655      NA         NA            930
4  2013     7    31       NA           1400      NA         NA           1508
5  2013     7    31       NA            959      NA         NA           1125
6  2013     7    31       NA           1025      NA         NA           1225
# ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

The `%in%` operator checks for group membership for any match in the specified values. The following example selects all rows matching the months of March, May, or July.

```
> filter(flights, month %in% c(3, 5, 7))
# A tibble: 87,055 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     3     1        4           2159    125         318
 2  2013     3     1       50           2358     52.0       526
 3  2013     3     1      117           2245    152         223
 4  2013     3     1      454            500   -  6.00      633
 5  2013     3     1      505            515   - 10.0       746
 6  2013     3     1      521            530   -  9.00      813
 7  2013     3     1      537            540   -  3.00      856
 8  2013     3     1      541            545   -  4.00     1014
 9  2013     3     1      549            600   - 11.0       639
10  2013     3     1      550            600   - 10.0       747
# ... with 87,045 more rows, and 12 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

Arranging Rows

The arrange function changes the order of the data set based on the specified columns. The following example data set is used for illustrating the function.

```
> athlete.info
# A tibble: 5 x 4
  Name        Salary Endorsements Sport
  <chr>        <dbl>        <dbl> <chr>
1 Mayweather  105             0   Boxing
2 Ronaldo      52.0          28.0 Soccer
3 James        19.3          53.0 Basketball
4 Messi        41.7          23.0 Soccer
5 Bryant       30.5          31.0 Basketball
```

By default, the data is arranged in ascending order of the specified column (*Salary*) as shown below.

```
> arrange(athlete.info, Salary)
# A tibble: 5 x 4
  Name        Salary Endorsements Sport
  <chr>        <dbl>        <dbl> <chr>
1 James        19.3          53.0 Basketball
2 Bryant       30.5          31.0 Basketball
3 Messi        41.7          23.0 Soccer
4 Ronaldo      52.0          28.0 Soccer
5 Mayweather  105             0   Boxing
```

Multiple variables can also be used for arranging the data. In the following example, the data is first arranged in ascending order of the *Sport*, and then by *Name* within the *Sport*.

```
> arrange(athlete.info, Sport, Name)
# A tibble: 5 x 4
  Name        Salary Endorsements Sport
  <chr>        <dbl>         <dbl> <chr>
1 Bryant        30.5          31.0 Basketball
2 James         19.3          53.0 Basketball
3 Mayweather   105             0   Boxing
4 Messi         41.7          23.0 Soccer
5 Ronaldo       52.0          28.0 Soccer
```

If the results are desired in a descending order, the `desc` function on the variable is used as shown below.

```
> arrange(athlete.info, desc(Salary))
# A tibble: 5 x 4
  Name        Salary Endorsements Sport
  <chr>        <dbl>         <dbl> <chr>
1 Mayweather   105             0   Boxing
2 Ronaldo       52.0          28.0 Soccer
3 Messi         41.7          23.0 Soccer
4 Bryant        30.5          31.0 Basketball
5 James         19.3          53.0 Basketball
```

Selecting Columns

Typical datasets have more columns than what are required for the analysis at hand. In such cases, the datasets can be narrowed down to only the required columns using the `select` function as shown below. The column names are explicitly specified in the following case for the `flights` dataset.

```
> select(flights, carrier, flight, tailnum, origin, dest)
# A tibble: 336,776 x 5
   carrier flight tailnum origin dest
   <chr>    <int> <chr>   <chr>  <chr>
 1 UA        1545 N14228  EWR    IAH
 2 UA        1714 N24211  LGA    IAH
 3 AA        1141 N619AA  JFK    MIA
 4 B6         725 N804JB  JFK    BQN
 5 DL         461 N668DN  LGA    ATL
 6 UA        1696 N39463  EWR    ORD
 7 B6         507 N516JB  EWR    FLL
 8 EV        5708 N829AS  LGA    IAD
 9 B6          79 N593JB  JFK    MCO
10 AA         301 N3ALAA  LGA    ORD
# ... with 336,766 more rows
```

If the columns that are required are in consecutive columns, the expression `start:end` can be used to specify the column to start from and the column to end as in the following example.

```
> select(flights, carrier:dest)
# A tibble: 336,776 x 5
   carrier flight tailnum origin dest
   <chr>    <int> <chr>   <chr>  <chr>
 1 UA        1545 N14228  EWR    IAH
 2 UA        1714 N24211  LGA    IAH
 3 AA        1141 N619AA  JFK    MIA
 4 B6         725 N804JB  JFK    BQN
 5 DL         461 N668DN  LGA    ATL
 6 UA        1696 N39463  EWR    ORD
 7 B6         507 N516JB  EWR    FLL
 8 EV        5708 N829AS  LGA    IAD
 9 B6          79 N593JB  JFK    MCO
10 AA         301 N3ALAA  LGA    ORD
# ... with 336,766 more rows
```

The minus operator can be used to used to select all columns except the specified ones as shown below.

```
> select(flights, -(carrier:dest))
# A tibble: 336,776 x 14
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515      2.00      830
 2  2013     1     1      533            529      4.00      850
 3  2013     1     1      542            540      2.00      923
 4  2013     1     1      544            545     -1.00     1004
 5  2013     1     1      554            600     -6.00      812
 6  2013     1     1      554            558     -4.00      740
 7  2013     1     1      555            600     -5.00      913
 8  2013     1     1      557            600     -3.00      709
 9  2013     1     1      557            600     -3.00      838
10  2013     1     1      558            600     -2.00      753
# ... with 336,766 more rows, and 7 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

The `select` function can also be used to rename the variables. Only the specified columns are included in the result.

```
> select(flights, departure_time = dep_time)
# A tibble: 336,776 x 1
   departure_time
            <int>
 1            517
 2            533
 3            542
 4            544
 5            554
 6            554
 7            555
 8            557
 9            557
10            558
# ... with 336,766 more rows
```

If the rest of the columns should also be included as is, the `rename` function is used for renaming the specified columns and include the rest without any changes, as shown below.

```
> rename(flights, departure_time = dep_time)
# A tibble: 336,776 x 19
    year month   day departure_time sched_dep_time dep_delay arr_time
   <int> <int> <int>          <int>          <int>     <dbl>    <int>
 1  2013     1     1            517            515      2.00      830
 2  2013     1     1            533            529      4.00      850
 3  2013     1     1            542            540      2.00      923
 4  2013     1     1            544            545     -1.00     1004
 5  2013     1     1            554            600     -6.00      812
 6  2013     1     1            554            558     -4.00      740
 7  2013     1     1            555            600     -5.00      913
 8  2013     1     1            557            600     -3.00      709
 9  2013     1     1            557            600     -3.00      838
10  2013     1     1            558            600     -2.00      753
# ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

The helper functions `starts_with` can be used to select all column names that start with the specified value.

```
> select(flights, starts_with("d"))
# A tibble: 336,776 x 5
     day dep_time dep_delay dest  distance
   <int>    <int>     <dbl> <chr>     <dbl>
 1     1      517      2.00 IAH        1400
 2     1      533      4.00 IAH        1416
 3     1      542      2.00 MIA        1089
 4     1      544     -1.00 BQN        1576
 5     1      554     -6.00 ATL         762
 6     1      554     -4.00 ORD         719
 7     1      555     -5.00 FLL        1065
 8     1      557     -3.00 IAD         229
 9     1      557     -3.00 MCO         944
10     1      558     -2.00 ORD         733
# ... with 336,766 more rows
```

The `ends_with` function matches all column names that end with the specified value as shown below.

```
> select(flights, ends_with("time"))
# A tibble: 336,776 x 5
   dep_time sched_dep_time arr_time sched_arr_time air_time
      <int>          <int>    <int>          <int>    <dbl>
 1      517            515      830            819      227
 2      533            529      850            830      227
 3      542            540      923            850      160
 4      544            545     1004           1022      183
 5      554            600      812            837      116
 6      554            558      740            728      150
 7      555            600      913            854      158
 8      557            600      709            723       53.0
 9      557            600      838            846      140
10      558            600      753            745      138
# ... with 336,766 more rows
```

All column names that contain the specified value can be selected as shown below.

```
> select(flights, contains("in"))
# A tibble: 336,776 x 2
   origin minute
   <chr>   <dbl>
 1 EWR      15.0
 2 LGA      29.0
 3 JFK      40.0
 4 JFK      45.0
 5 LGA       0
 6 EWR      58.0
 7 EWR       0
 8 LGA       0
 9 JFK       0
10 LGA       0
# ... with 336,766 more rows
```

The `select` function used along with the `everything` helper function allows the specified columns to be moved to the start of the data frame followed by everything else.

```
> select(flights, flight, origin, dest, everything())
# A tibble: 336,776 x 19
   flight origin dest   year month   day dep_time sched_dep_time dep_delay
    <int> <chr>  <chr> <int> <int> <int>    <int>          <int>     <dbl>
 1   1545 EWR    IAH    2013     1     1      517            515      2.00
 2   1714 LGA    IAH    2013     1     1      533            529      4.00
 3   1141 JFK    MIA    2013     1     1      542            540      2.00
 4    725 JFK    BQN    2013     1     1      544            545     -1.00
 5    461 LGA    ATL    2013     1     1      554            600     -6.00
 6   1696 EWR    ORD    2013     1     1      554            558     -4.00
 7    507 EWR    FLL    2013     1     1      555            600     -5.00
 8   5708 LGA    IAD    2013     1     1      557            600     -3.00
 9     79 JFK    MCO    2013     1     1      557            600     -3.00
10    301 LGA    ORD    2013     1     1      558            600     -2.00
# ... with 336,766 more rows, and 10 more variables: arr_time <int>,
#   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, tailnum <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dttm>
```

Adding New Variables

The `mutate` function adds new columns to the dataset based on the the expressions specified involving the existing columns. The following example adds two new columns, *gain* and *speed*, based on the values of the existing columns as specified below. The results are piped to the `select` function to rearrange the affected columns to the beginning of the dataset.

```
> flights %>%
    mutate(gain = arr_delay - dep_delay,
           speed = (distance / air_time) * 60)  %>%
    select(flight, arr_delay, dep_delay, gain,
           distance, air_time, speed, everything())
# A tibble: 336,776 x 21
   flight arr_delay dep_delay   gain distance air_time speed  year month
    <int>     <dbl>     <dbl>  <dbl>    <dbl>    <dbl> <dbl> <int> <int>
 1   1545     11.0       2.00   9.00     1400    227     370  2013     1
 2   1714     20.0       4.00  16.0      1416    227     374  2013     1
 3   1141     33.0       2.00  31.0      1089    160     408  2013     1
 4    725    -18.0      -1.00 -17.0      1576    183     517  2013     1
 5    461    -25.0      -6.00 -19.0       762    116     394  2013     1
 6   1696     12.0      -4.00  16.0       719    150     288  2013     1
 7    507     19.0      -5.00  24.0      1065    158     404  2013     1
 8   5708    -14.0      -3.00 -11.0       229     53.0   259  2013     1
 9     79    - 8.00     -3.00 - 5.00      944    140     405  2013     1
10    301      8.00     -2.00  10.0       733    138     319  2013     1
# ... with 336,766 more rows, and 12 more variables: day <int>,
#   dep_time <int>, sched_dep_time <int>, arr_time <int>,
#   sched_arr_time <int>, carrier <chr>, tailnum <chr>, origin <chr>,
#   dest <chr>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

The new columns that are being added can also refer to the columns that were just created on the left as shown below. The *speed* column refers to the `air_time_in_hours` column in the following example.

```
> flights %>%
    mutate(air_time_in_hours = air_time/60,
           speed = distance / air_time_in_hours)  %>%
    select(flight, distance, air_time, air_time_in_hours,
           speed, everything())
# A tibble: 336,776 x 21
   flight distance air_time air_time_in_hou… speed  year month   day dep_time
    <int>    <dbl>    <dbl>            <dbl> <dbl> <int> <int> <int>    <int>
 1   1545     1400      227             3.78   370  2013     1     1      517
 2   1714     1416      227             3.78   374  2013     1     1      533
 3   1141     1089      160             2.67   408  2013     1     1      542
 4    725     1576      183             3.05   517  2013     1     1      544
 5    461      762      116             1.93   394  2013     1     1      554
 6   1696      719      150             2.50   288  2013     1     1      554
 7    507     1065      158             2.63   404  2013     1     1      555
 8   5708      229       53.0           0.883  259  2013     1     1      557
 9     79      944      140             2.33   405  2013     1     1      557
10    301      733      138             2.30   319  2013     1     1      558
# ... with 336,766 more rows, and 12 more variables: sched_dep_time <int>,
#   dep_delay <dbl>, arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, tailnum <chr>, origin <chr>, dest <chr>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

The `transmute` function only returns the data set with the new columns as shown below.

```
> flights %>%
.   transmute(gain = arr_delay - dep_delay,
.             speed = (distance / air_time) * 60)
# A tibble: 336,776 x 2
     gain speed
    <dbl> <dbl>
 1   9.00   370
 2  16.0    374
 3  31.0    408
 4 -17.0    517
 5 -19.0    394
 6  16.0    288
 7  24.0    404
 8 -11.0    259
 9 - 5.00   405
10  10.0    319
# ... with 336,766 more rows
```

Grouped Summaries

The `summarize` function is typically used on grouped data to summarize each group as a single row. In the simplest case, the summarize function can be used to collapse the entire data frame into a single row. The following example summarizes the arrival delays in the `flights` data set using the mean of the data. The NA values in the data result in a mean of NA. Those NA values can be ignored with the `na.rm` option.

```
> flights %>%
    summarise(delay = mean(arr_delay))
# A tibble: 1 x 1
  delay
  <dbl>
1    NA

> flights %>%
    summarise(delay = mean(arr_delay, na.rm = TRUE))
# A tibble: 1 x 1
  delay
  <dbl>
1  6.90
```

The `arr_delay` column in the data set contains both positive values and negative values, negative values indicating the early arrival of the flight. If the study is only to look at the delays, the data can first be filtered to select all rows with a positive value for `arr_delay`, and examin the number of such rows and the mean of those delays, as shown below.

```
> flights %>%
    filter(arr_delay > 0) %>%
    summarise(count = n(),
              avg_delay = mean(arr_delay))
# A tibble: 1 x 2
   count avg_delay
   <int>     <dbl>
1 133004      40.3
```

Similarly, for flights that arrived early, the average will be a negative value as shown below.

```
> flights %>%
    filter(arr_delay <= 0) %>%
    summarise(count = n(),
              avg_early = mean(arr_delay))
# A tibble: 1 x 2
   count avg_early
   <int>     <dbl>
1 194342     -16.0
```

The `summarize` function is more used on grouped data and is frequently paired with the `group_by` function. In the following example, in order to examine the average delays of the flights by month, the data is first grouped by *year* and *month*, and summarized. Now the results show the number of flights and average delays by month for

all the flights that arrived late.

```
> flights %>%
    filter(arr_delay > 0) %>%
    group_by(year, month) %>%
    summarise(count = n(),
              avg_delay = mean(arr_delay))
# A tibble: 12 x 4
# Groups:   year [?]
    year month count avg_delay
   <int> <int> <int>     <dbl>
 1  2013     1 11150      34.5
 2  2013     2 10100      33.7
 3  2013     3 10919      40.6
 4  2013     4 12522      42.7
 5  2013     5 10189      41.9
 6  2013     6 12490      53.7
 7  2013     7 13304      54.0
 8  2013     8 11629      39.5
 9  2013     9  6845      38.8
10  2013    10  9823      29.0
11  2013    11  9639      27.5
12  2013    12 14394      39.7
```

In the following scenario, the data is first grouped by the flights *origin* and summarized to show the number of flights for each origin airport and the average delays of the flights from those airports.

```
> flights %>%
    filter(arr_delay > 0) %>%
    group_by(origin) %>%
    summarise(count = n(),
              avg_delay = mean(arr_delay))
# A tibble: 3 x 3
  origin count avg_delay
  <chr>  <int>     <dbl>
1 EWR    50099      41.8
2 JFK    42885      40.0
3 LGA    40020      38.9
```

The following example groups the data by both the flights origin and destination and shows the number of flights in the data set for each such pair and the average delays for that combination.

```
> flights %>%
    filter(arr_delay > 0) %>%
    group_by(origin, dest) %>%
    summarise(count = n(),
              avg_delay = mean(arr_delay))
# A tibble: 219 x 4
# Groups:   origin [?]
   origin dest  count avg_delay
   <chr>  <chr> <int>     <dbl>
 1 EWR    ALB     184      52.1
 2 EWR    ANC       5      12.4
 3 EWR    ATL    2300      41.6
 4 EWR    AUS     317      35.4
 5 EWR    AVL     117      31.2
 6 EWR    BDL     144      48.3
 7 EWR    BNA     985      46.8
 8 EWR    BOS    1683      45.7
 9 EWR    BQN     144      37.7
10 EWR    BTV     396      42.8
# ... with 209 more rows
```

Data Organization with tidyr

A data set is called *tidy* if each variable is saved in its own column and each observation is stored in its own row. A tidy set makes it easy for further data analysis and exploration. The following functions from the `tidyr` package are frequently used in tidying the data.

- gather – takes multiple columns in the data set and collapses them into key-value pairs. The resulting data set is typically known as the long form of the data.
- separate – takes the values in a column and splits them into multiple columns.
- unite – takes the values from multiple columns and combines them into a single column.
- spread – takes the data in a long form and spreads the key-value pairs across multiple columns.

The following sample dataset is used for illustrating the `tidyr` functions. The dataset contains the revenue of three stores per quarter for the years 2014 through 2017. There are 12 rows of measured data in the dataset.

```
> set.seed(123)
> sales <-
    tibble(Store=rep(1:3, each=4),
                Year=rep(2014:2017, 3),
                Qtr_1 = round(runif(12, 10, 30)),
                Qtr_2 = round(runif(12, 10, 30)),
                Qtr_3 = round(runif(12, 10, 30)),
                Qtr_4 = round(runif(12, 10, 30))
                )
> sales
# A tibble: 12 x 6
   Store  Year Qtr_1 Qtr_2 Qtr_3 Qtr_4
   <int> <int> <dbl> <dbl> <dbl> <dbl>
 1     1  2014  16.0  24.0  23.0  25.0
 2     1  2015  26.0  21.0  24.0  14.0
 3     1  2016  18.0  12.0  21.0  16.0
 4     1  2017  28.0  28.0  22.0  15.0
 5     2  2014  29.0  15.0  16.0  13.0
 6     2  2015  11.0  11.0  13.0  18.0
 7     2  2016  21.0  17.0  29.0  18.0
 8     2  2017  28.0  29.0  28.0  17.0
 9     3  2014  21.0  28.0  24.0  13.0
10     3  2015  19.0  24.0  26.0  13.0
11     3  2016  29.0  23.0  10.0  15.0
12     3  2017  19.0  30.0  20.0  19.0
```

The columns *Qtr_1, Qtr_2, Qtr_3,* and *Qtr_4* are not variables but measurement values of the data set. The process of tidying up the data set involves keeping the measured values as key-value pairs, key being the *Quarter* and the value being the measured data. The `gather` function converts the dataset into the tidy format as shown below. The 12 rows of data are now transformed into 48 rows of data – first 12 rows for *Qtr_1*, followed by 12 rows for *Qtr_2,* 12 rows for *Qtr_3,* and finally the last 12 rows for *Qtr_4.*

```
> sales %>%
    gather(Quarter, Revenue, Qtr_1 : Qtr_4) %>%
    head(12)
# A tibble: 12 x 4
   Store  Year Quarter Revenue
   <int> <int> <chr>     <dbl>
 1     1  2014 Qtr_1      16.0
 2     1  2015 Qtr_1      26.0
 3     1  2016 Qtr_1      18.0
 4     1  2017 Qtr_1      28.0
 5     2  2014 Qtr_1      29.0
 6     2  2015 Qtr_1      11.0
 7     2  2016 Qtr_1      21.0
 8     2  2017 Qtr_1      28.0
 9     3  2014 Qtr_1      21.0
10     3  2015 Qtr_1      19.0
11     3  2016 Qtr_1      29.0
12     3  2017 Qtr_1      19.0
```

```
> sales %>%
    gather(Quarter, Revenue, Qtr_1 : Qtr_4) %>%
    tail(12)
# A tibble: 12 x 4
   Store  Year Quarter Revenue
   <int> <int> <chr>     <dbl>
 1     1  2014 Qtr_4      25.0
 2     1  2015 Qtr_4      14.0
 3     1  2016 Qtr_4      16.0
 4     1  2017 Qtr_4      15.0
 5     2  2014 Qtr_4      13.0
 6     2  2015 Qtr_4      18.0
 7     2  2016 Qtr_4      18.0
 8     2  2017 Qtr_4      17.0
 9     3  2014 Qtr_4      13.0
10     3  2015 Qtr_4      13.0
11     3  2016 Qtr_4      15.0
12     3  2017 Qtr_4      19.0
```

The following examples show the alternative approaches for doing the above with different styles of code. The *key* and *value* arguments are explicitly specified in the following case:

```
> sales %>%
    gather(key = Quarter, value = Revenue, Qtr_1 : Qtr_4)
# A tibble: 48 x 4
   Store  Year Quarter Revenue
   <int> <int> <chr>     <dbl>
 1     1  2014 Qtr_1      16.0
 2     1  2015 Qtr_1      26.0
 3     1  2016 Qtr_1      18.0
 4     1  2017 Qtr_1      28.0
 5     2  2014 Qtr_1      29.0
 6     2  2015 Qtr_1      11.0
 7     2  2016 Qtr_1      21.0
 8     2  2017 Qtr_1      28.0
 9     3  2014 Qtr_1      21.0
10     3  2015 Qtr_1      19.0
# ... with 38 more rows
```

An alternative way to express the columns is to exclude the ones that are not measured and include the rest as shown below.

```
> sales %>% gather(Quarter, Revenue, -Store, -Year)
# A tibble: 48 x 4
   Store  Year Quarter Revenue
   <int> <int> <chr>     <dbl>
 1     1  2014 Qtr_1      16.0
 2     1  2015 Qtr_1      26.0
 3     1  2016 Qtr_1      18.0
 4     1  2017 Qtr_1      28.0
 5     2  2014 Qtr_1      29.0
 6     2  2015 Qtr_1      11.0
 7     2  2016 Qtr_1      21.0
 8     2  2017 Qtr_1      28.0
 9     3  2014 Qtr_1      21.0
10     3  2015 Qtr_1      19.0
# ... with 38 more rows
```

The measured columns can also be explicitly specified as shown below.

```
> sales %>% gather(Quarter, Revenue,
                   Qtr_1, Qtr_2, Qtr_3, Qtr_4)
# A tibble: 48 x 4
   Store  Year Quarter Revenue
   <int> <int> <chr>     <dbl>
 1     1  2014 Qtr_1      16.0
 2     1  2015 Qtr_1      26.0
 3     1  2016 Qtr_1      18.0
 4     1  2017 Qtr_1      28.0
 5     2  2014 Qtr_1      29.0
 6     2  2015 Qtr_1      11.0
 7     2  2016 Qtr_1      21.0
 8     2  2017 Qtr_1      28.0
 9     3  2014 Qtr_1      21.0
10     3  2015 Qtr_1      19.0
# ... with 38 more rows
```

Assuming that the converted data above is stored in the variable named `long_data`, the `separate`Quarter column *(Qtr_1, Qtr_2, Qtr_3,* and *Qtr_4*) are split into two separate columns as shown below.

```
> long_data %>%
    separate(Quarter, c("Time_Interval", "Interval_ID"),
             convert = TRUE) -> separate_data
> separate_data
# A tibble: 48 x 5
   Store  Year Time_Interval Interval_ID Revenue
   <int> <int> <chr>              <int>   <dbl>
 1     1  2014 Qtr                    1    16.0
 2     1  2015 Qtr                    1    26.0
 3     1  2016 Qtr                    1    18.0
 4     1  2017 Qtr                    1    28.0
 5     2  2014 Qtr                    1    29.0
 6     2  2015 Qtr                    1    11.0
 7     2  2016 Qtr                    1    21.0
 8     2  2017 Qtr                    1    28.0
 9     3  2014 Qtr                    1    21.0
10     3  2015 Qtr                    1    19.0
# ... with 38 more rows
```

The default separator for splitting is any non-alphanumeric character present in the value. An explicit separator may also be specified as shown below.

```
> long_data %>% separate(Quarter, into = c("Time_Interval", "Interval_ID"),
                         sep = "_", convert = TRUE) -> separate_data
> separate_data
# A tibble: 48 x 5
   Store  Year Time_Interval Interval_ID Revenue
   <int> <int> <chr>              <int>   <dbl>
 1     1  2014 Qtr                    1    16.0
 2     1  2015 Qtr                    1    26.0
 3     1  2016 Qtr                    1    18.0
 4     1  2017 Qtr                    1    28.0
 5     2  2014 Qtr                    1    29.0
 6     2  2015 Qtr                    1    11.0
 7     2  2016 Qtr                    1    21.0
 8     2  2017 Qtr                    1    28.0
 9     3  2014 Qtr                    1    21.0
10     3  2015 Qtr                    1    19.0
# ... with 38 more rows
```

The `unite` function can be used to combine values from multiple columns into a single column as shown below.

The default separator for combining the values is the underscore.

```
> separate_data %>%
.   unite(Quarter, Time_Interval, Interval_ID)
# A tibble: 48 x 4
   Store  Year Quarter Revenue
   <int> <int> <chr>     <dbl>
 1     1  2014 Qtr_1      16.0
 2     1  2015 Qtr_1      26.0
 3     1  2016 Qtr_1      18.0
 4     1  2017 Qtr_1      28.0
 5     2  2014 Qtr_1      29.0
 6     2  2015 Qtr_1      11.0
 7     2  2016 Qtr_1      21.0
 8     2  2017 Qtr_1      28.0
 9     3  2014 Qtr_1      21.0
10     3  2015 Qtr_1      19.0
# ... with 38 more rows
```

An explicit separator may also be specified as the combining character as shown below.

```
> separate_data %>%
    unite(Quarter, Time_Interval, Interval_ID, sep = ".")
# A tibble: 48 x 4
   Store  Year Quarter Revenue
   <int> <int> <chr>     <dbl>
 1     1  2014 Qtr.1      16.0
 2     1  2015 Qtr.1      26.0
 3     1  2016 Qtr.1      18.0
 4     1  2017 Qtr.1      28.0
 5     2  2014 Qtr.1      29.0
 6     2  2015 Qtr.1      11.0
 7     2  2016 Qtr.1      21.0
 8     2  2017 Qtr.1      28.0
 9     3  2014 Qtr.1      21.0
10     3  2015 Qtr.1      19.0
# ... with 38 more rows
```

The `spread` method is the inverse of the `gather` operation used to spread the values back into the respective columns. This is typically used when an observation is spread across multiple rows.

```
> long_data %>% spread(Quarter, Revenue)
# A tibble: 12 x 6
   Store  Year Qtr_1 Qtr_2 Qtr_3 Qtr_4
   <int> <int> <dbl> <dbl> <dbl> <dbl>
 1     1  2014  16.0  24.0  23.0  25.0
 2     1  2015  26.0  21.0  24.0  14.0
 3     1  2016  18.0  12.0  21.0  16.0
 4     1  2017  28.0  28.0  22.0  15.0
 5     2  2014  29.0  15.0  16.0  13.0
 6     2  2015  11.0  11.0  13.0  18.0
 7     2  2016  21.0  17.0  29.0  18.0
 8     2  2017  28.0  29.0  28.0  17.0
 9     3  2014  21.0  28.0  24.0  13.0
10     3  2015  19.0  24.0  26.0  13.0
11     3  2016  29.0  23.0  10.0  15.0
12     3  2017  19.0  30.0  20.0  19.0
```

References

Introduction to dplyr

Tidy data

Tidyverse

Package `nycflights13`

Boston University Metropolitan College