R프로그래밍

6장. 문자열 다루기

박 혜 승 교 수

6장. 문자열 다루기

6.1 문자열 시작하기

6.2 날짜/시간 서식

6.3 정규 표현식 사용하기

6.4 마치며

6.1 문자열 시작하기

6.1 문자열 시작하기

- 문자열 시작하기

  - ✓ R의 문자형 벡터는 텍스트 데이터를 저장하는 데 사용.
  - ✓ R에서 문자형 벡터란 다른 많은 프로그래밍 언어와 달리 a, b, c 같은 단일 문자나 알파벳 기호로 된 벡터가 아니라, 오히려 문자열로 구성된 벡터라고 볼 수 있음
  - ✓ R은 또한 문자형 벡터를 다루는 다양한 내장 함수 제공.
  - ✓ 벡터화 연산을 실행하여 많은 문자열 값을 한꺼번에 처리 가능.

6.1 문자열 시작하기

- 텍스트 출력하기

✓ 텍스트를 콘솔 창에 출력하는 간단한 방법은 큰따옴표를 사용하여 직접 텍스트를 입력하는 것.

```
> "Hello"
[1] "Hello"
```

✓ 이 예제는 첫 번째 위치에 유일한 요소인 “Hello”가 있는 문자형 벡터.

✓ 변수에 저장된 문자열 값도 출력 가능.

```
> str1 <- "Hello"
> str1
[1] "Hello"
```

6.1 문자열 시작하기

- 텍스트 출력하기

✓ 어떤 반복문 안에 다음과 같이 단순히 문자 값을 써넣었다고 해서 이것이 출력되지는 않음.

```
> for (i in 1:3) {
+     "Hello"
+ }
```

✓ R에서는 콘솔 창에 값을 직접 입력할 때만 해당 표현식의 값이 자동으로 출력되기 때문.

✓ for 루프는 명시적으로 어떤 값도 반환하지 않음.

✓ 이러한 성질 때문에 다음 두 함수를 각각 호출했을 때 출력이 서로 다름.

```
> test1 <- function(x) {
+     "Hello"
+     x
+ }
> test1("World")
[1] "World"
```

✓ test1 함수는 “Hello”를 출력하지 않음.

✓ test1("World")는 x값으로 World가 주어진 함수 호출이면서 마지막 표현식인 x를 그 결과로 반환하기 때문에 “World”를 출력.

6.1 문자열 시작하기

- 텍스트 출력하기

✓ 다음과 같이 함수에서 x를 없앤다고 가정하자.

```
> test2 <- function(x) {
+     "Hello"
+ }
> test2("World")
[1] "Hello"
```

✓ test2 함수는 x에 어떤 값이 주어지든 상관없이 언제나 “Hello”를 반환.

✓ 결과적으로 R은 표현식 test2("World”)의 결과값 “Hello”를 자동으로 출력.

6.1 문자열 시작하기

- 텍스트 출력하기

✓ 어떤 객체를 명시적으로 출력하고 싶다면 print() 함수 사용.

```
> print(str1)
[1] "Hello"
```

✓ 위 코드의 결과는 [1]이라는 위치 정보와 함께 문자형 벡터 출력.

✓ 이것은 루프에서도 사용 가능.

```
> for (i in 1:3) {
+     print(str1)
+ }
[1] "Hello"
[1] "Hello"
[1] "Hello"
```

6.1 문자열 시작하기

- 텍스트 출력하기

  ✓ 다음과 같이 함수 안에서 print() 함수 사용 가능.

```
> test3 <- function(x) {
+     print("Hello")
+     x
+ }
> test3("World")
[1] "Hello"
[1] "World"
```

6.1 문자열 시작하기

- 텍스트 출력하기

✓ 텍스트를 문자형 벡터보다는 위치 정보가 없는 형태로 출력할 때는 cat()이나 message() 함수 사용.

```
> cat("Hello")
Hello
```

✓ 다음과 같이 다양한 방법으로 메시지를 만들 수 있음.

```
> name <- "Ken"
> language <- "R"
> cat("Hello,", name, "- a user of", language)
Hello, Ken - a user of R
```

✓ 좀 더 완전한 문장을 만들고자 입력을 조금 수정해 보자.

```
> cat("Hello, ", name, ", a user of ", language, ".")
Hello,  Ken , a user of  R .
```

✓ 위 코드의 결과에는 주어진 문자열을 연결하면서 인수 사이에 불필요한 공백이 존재.

✓ 그 이유는 기본적으로 입력 문자열을 연결할 때 구분 문자로 공백 문자를 사용하기 때문.

6.1 문자열 시작하기

- 텍스트 출력하기

  ✓ sep 인수로 앞 문제를 조절 가능.

  ✓ 다음 예제에서는 기본 공백 문자가 매번 들어가는 것을 방지하고 필요한 부분에만 수동으로 공백을 넣어 올바른 문장을 만듦.

```
> cat("Hello, ", name, ", a user of ", language, ".", sep = "")
Hello, Ken, a user of R.
```

6.1 문자열 시작하기

- 텍스트 출력하기

  ✓ message() 함수는 중요한 이벤트가 발생하는 상황을 알리고 싶을 때 주로 사용.

  ✓ 출력된 결괏값은 눈에 띄는 특징이 있는데, cat()과 달리 입력 문자열을 연결할 때 공백 문자를 사용하지 않음.

```
> message("Hello, ", name, ", a user of ", language, ".")
Hello, Ken, a user of R.
```

  ✓ message() 함수를 cat() 함수의 결과와 같게 출력하려면 구분 기호를 수동으로 넣어야 함.

6.1 문자열 시작하기

- 텍스트 출력하기

✓ message()는 cat()과 달리 텍스트를 끝낼 때 자동으로 줄 바꿈을 실행.
✓ 다음 두 예제는 이 차이를 잘 보여줌.

```
> for (i in 1:3) {
+     cat(letters[[i]])
+ }
abc
> for (i in 1:3) {
+     message(letters[[i]])
+ }
a
b
c
```

✓ cat() 함수를 호출할 때는 매번 줄 바꿈 하지 않고 입력 문자열을 출력.
✓ 반면, message() 함수를 호출할 때는 입력 문자열마다 줄 바꿈.

6.1 문자열 시작하기

- 텍스트 출력하기

  ✓ cat() 함수를 사용하여 줄 바꿈을 하려면 명시적으로 입력마다 개행 문자를 삽입.
  ✓ 다음 코드는 message() 함수를 사용할 때와 같은 결과를 출력.

```
> for (i in 1:3) {
+     cat(letters[[i]], "\n", sep = "")
+ }
a
b
c
```

6.1 문자열 시작하기

- 문자열 연결하기

✓ paste() 함수는 여러 문자형 벡터를 연결하여 새로운 문자열을 생성할 때 사용.
✓ paste() 함수 역시 공백 문자를 기본 구분 문자로 사용.

```
> paste("Hello", "world")
[1] "Hello world"
> paste("Hello", "world", sep = "-")
[1] "Hello-world"
```

✓ 구분 문자를 사용하지 않으려면 paste() 함수에서 sep = ""로 설정하거나 paste0() 함수를 대신 사용.

```
> paste0("Hello", "world")
[1] "Helloworld"
```

6.1 문자열 시작하기

- 문자열 연결하기

  ✓ cat() 함수가 콘솔 창에 결과 문자열을 단순히 출력하는 것과 달리 paste() 함수는 나중을 위해 이 문자열을 결과로 반환.

  ✓ 다음 코드는 cat() 함수가 문자열을 출력하기는 하지만 NULL을 반환하는 것을 보여줌.

```
> value1 <- cat("Hello", "world")
Hello world
> value1
NULL
```

  ✓ cat() 함수는 문자열을 출력만 하고, paste() 함수는 새로운 문자형 벡터를 생성.

6.1 문자열 시작하기

- 문자열 연결하기

✓ 이전 예제는 paste() 함수의 입력으로 단일 요소 문자형 벡터가 주어진 경우를 다룸.

✓ 다중 요소 벡터가 주어진다면 어떻게 동작할까?

```
> paste(c("A", "B"), c("C", "D"))
[1] "A C" "B D"
```

✓ paste() 함수가 요소별로 동작하는 것을 볼 수 있음.

✓ 즉, paste("A", "C")를 먼저 실행하고 paste("B", "D")를 나중에 실행하여 요소를 2개 갖는 문자형 벡터를 출력.

6.1 문자열 시작하기

- 문자열 연결하기

✓ 한 문자열에 두 요소를 출력하고 싶다면 두 요소를 연결하는 collapse 인수를 설정.

```
> paste(c("A", "B"), c("C", "D"), collapse = ", ")
[1] "A C, B D"
```

✓ 각 요소를 서로 다른 줄에 표시하고 싶다면 collapse에 개행 문자 \n을 추가.

```
> result <- paste(c("A", "B"), c("C", "D"), collapse = "\n")
> result
[1] "A C\nB D"
```

```
> cat(result)
A C
B D
```

✓ 이제 텍스트가 의도한 대로 콘솔 창에 두 줄로 출력.

✓ paste0() 함수 역시 paste()와 동작이 비슷.

6.1 문자열 시작하기

- 텍스트 변환하기

✓ 텍스트를 다른 형식으로 변환하는 것이 유용할 때가 많음.
✓ 텍스트를 변환하는 여러 가지 기본 유형이 있으며, 쉽게 사용이 가능.

대·소문자 변경하기

✓ 텍스트를 활용하여 데이터를 처리할 때 가정한 표준에 맞지 않게 많이 입력하곤 함.
✓ 예를 들어 성적과 관련한 데이터가 있을 때 A에서 F까지 대문자로 구분해서 입력되길 기대하지만, 실제로는 대·소문자 구분 없이 입력될 수 있음.
✓ 대·소문자를 변환하면 입력 문자열의 일관성을 유지하는데 도움.

6.1 문자열 시작하기

- 텍스트 변환하기

  ✓ tolower() 함수는 텍스트를 소문자로 변경.
  ✓ toupper() 함수는 텍스트를 대문자로 변경.

```
> tolower("Hello")
[1] "hello"
> toupper("Hello")
[1] "HELLO"
```

  ✓ 소문자나 대문자로 변경하는 함수들은 문자를 입력으로 받는 함수에서 특히 유용.

6.1 문자열 시작하기

- 텍스트 변환하기

  ✓ 예를 들어 주어진 type이 대·소문자에 상관 없이 “add” (예: ADD”, “AdD”) 이면 x + y, “times”이면 x * y를 계산한다고 하자.

  ✓ 이때 가장 좋은 방법은 type을 항상 소문자 혹은 대문자로 변경하여 입력이 대문자인지 소문자인지 상관없게 하는 것.

```
> calc <- function(type, x, y) {
+     type <- tolower(type)
+     if (type == "add") {
+         x + y
+     } else if (type == "times") {
+         x * y
+     } else {
+         stop("Not supported type of command")
+     }
+ }
> c(calc("add", 2, 3), calc("Add", 2, 3), calc("TIMES", 2, 3))
[1] 5 5 6
```

6.1 문자열 시작하기

- 텍스트 변환하기

  ✓ 서로 대·소문자가 다른 비슷한 입력에서 좀 더 안정적인 동작을 보장.
  ✓ type은 대문자와 소문자를 구별하지 않음.
  ✓ 추가로 이 두 함수는 벡터화되어 있음.
  ✓ 즉, 문자형 벡터가 주어질 때 각 문자열 요소의 대·소문자가 변경.

```
> toupper(c("Hello", "world"))
[1] "HELLO" "WORLD"
```

6.1 문자열 시작하기

- 텍스트 변환하기

문자 개수 세기

✓ nchar() 함수는 문자형 벡터에서 각 요소의 문자 개수를 단순히 계산.

```
> nchar("Hello")
[1] 5
```

✓ toupper()나 tolower()와 마찬가지로 nchar()도 벡터화.

```
> nchar(c("Hello", "R", "User"))
[1] 5 1 4
```

✓ nchar() 함수는 인수에 공급되는 문자열이 유효한지 여부를 확인하는 데 자주 사용.

✓ 예를 들어 다음 함수는 학생의 개인 정보를 데이터베이스에 저장.

```
> store_student <- function(name, age) {
+     stopifnot(length(name) == 1, nchar(name) >= 2, is.numeric(age), age > 0)
+     # 데이터베이스에 정보를 저장한다
+ }
```

6.1 문자열 시작하기

- 텍스트 변환하기

  - ✓ store_student() 함수는 정보를 데이터베이스에 저장하기 전에 먼저 stopifnot() 함수로 name과 age의 값이 유효한지 확인.
  - ✓ 사용자가 적합하지 않은 이름(예를 들어 두 글자 미만)을 입력하면 이 함수는 오류를 발생시키며 실행을 멈춤.

```
> store_student("James", 20)
> store_student("P", 23)
Error in store_student("P", 23) : nchar(name) >= 2 is not TRUE
```

  - ✓ nchar(x) == 0은 x == ""와 의미가 같음.
  - ✓ 빈 문자열을 확인하는 데 두 가지 방법 모두 가능.

6.1 문자열 시작하기

- 텍스트 변환하기

선행과 후행 공백 문자 제거하기

✓ 이전 예제에서 배운 것처럼 nchar() 함수를 사용하여 name이 유효한지 확인.
✓ 때로는 입력 데이터에 쓸모없는 공백이 있을 수 있음.
✓ 데이터에 더 많은 노이즈가 추가되고 문자열 인수를 주의 깊게 검사해야 함.
✓ 예를 들어 store_student()는 name으로 "P"를 허용하지 않은 반면, 이름이 같은 " P "라는 이름은 통과.
✓ nchar(" P ")의 결과는 3이기 때문.

```
> store_student(" P ", 23)
```

6.1 문자열 시작하기

- 텍스트 변환하기

✓ 쓸모없는 공백이 들어갈 가능성을 고려하려면 store_student 함수를 수정해야 함.
✓ R 3.2.0에서는 주어진 문자열의 앞뒤 공백을 제거하려고 trimws() 함수를 도입.

```
> store_student2 <- function(name, age) {
+     stopifnot(length(name) == 1, nchar(trimws(name)) >= 2, is.numeric(age), age > 0)
+     # 데이터베이스에 정보를 저장한다
+ }
```

✓ 공백 문자와 같은 노이즈에 더욱 강건하게 변경.

```
> store_student2(" P ", 23)
Error in store_student2(" P ", 23) : nchar(trimws(name)) >= 2 is not TRUE
```

✓ trimws() 함수는 기본적으로 문자열의 앞뒤에 올 수 있는 공백 문자(스페이스 혹은 탭 문자)를 모두 제거.
✓ 공백을 제거하고자 하는 방향을 "left"(앞) 또는 "right"(뒤)로 지정 가능.

```
> trimws(c(" Hello", "World "), which = "left")
[1] "Hello"  "World "
```

6.1 문자열 시작하기

- 텍스트 변환하기

부분 문자열 추출하기

✓ substr() 함수를 사용하여 문자형 벡터에서도 서브세팅 가능.

✓ 다음 형식의 날짜 정보가 있다고 가정하자.

```
> dates <- c("Jan 3", "Feb 10", "Nov 15")
```

✓ 월을 나타내는 문자는 축약된 형태로 알파벳 3개로 구성.

✓ substr(str, start idx, end idx) 함수를 사용하여 다음과 같이 월 정보 추출 가능.

```
> substr(dates, 1, 3)
[1] "Jan" "Feb" "Nov"
```

✓ 일 정보를 추출하려면 substr()과 nchar() 함수를 함께 사용.

```
> substr(dates, 5, nchar(dates))
[1] "3"  "10" "15"
```

✓ 이제 입력 문자열에서 월/일 정보를 추출할 수 있게 됨

✓ 문자열을 같은 날짜를 의미하는 수치형 값으로 변형하는 함수가 있다면 유용할 것

6.1 문자열 시작하기

- 텍스트 변환하기

✓ 다음 코드는 이전에 배운 다양한 함수를 활용하여 만든 예제

```
> get_month_day <- function(x) {
+     months <- vapply(substr(tolower(x), 1, 3), function(md) {
+         switch(md, jan = 1, feb = 2, mar = 3, apr = 4, may = 5,
+                jun = 6, jul = 7, aug = 8, sep = 9, oct = 10, nov = 11, dec = 12)
+     }, numeric(1), USE.NAMES = FALSE)
+     days <- as.numeric(substr(x, 5,nchar(x)))
+     data.frame(month = months, day = days)
+ }
> get_month_day(dates)
    month    day
1       1      3
2       2     10
3      11     15
```

❖ vapply(list, func)는 sapply()의 안전한 버전. 각 반복 연산이 서로 독립적일 때 유용.

❖ USE.NAMES=FALSE : 이름 속성 없이 반환.

6.1 문자열 시작하기

- 텍스트 변환하기

  ✓ substr() 함수는 부분 문자열을 주어진 문자형 벡터로 대체하는 기능도 있음.

```
> substr(dates, 1, 3) <- c("Feb", "Dec", "Mar")
> dates
[1] "Feb 3"  "Dec 10" "Mar 15"
```

6.1 문자열 시작하기

- 텍스트 변환하기

텍스트 분할하기

✓ 많은 경우 추출할 문자열 길이는 고정되어 있지 않음.
✓ 예를 들어 "Mary Johnson" 또는 "Jack Smiths"처럼 사람마다 이름과 성의 길이가 모두 다름.
✓ substr() 함수를 사용하여 이름을 두 부분으로 분리하고 각각을 추출하기는 쉽지 않음.
✓ 이러한 형태의 텍스트에는 공백이나 쉼표처럼 구분 기호가 있음.
✓ 유용한 부분을 추출하려면 텍스트를 분할하고 각 부분에 접근할 수 있는 방법이 필요.
✓ strsplit() 함수는 문자형 벡터가 주어졌을 때 특정 구분 기호로 텍스트를 분할하는 데 사용.

```
> strsplit("a,bb,ccc", split = ",")
[[1]]
[1] "a"   "bb"  "ccc"
```

6.1 문자열 시작하기

- 텍스트 변환하기

  ✓ strsplit() 함수는 리스트를 결과로 반환.
  ✓ 이 리스트의 각 요소는 입력된 문자형 벡터에서 분할되어 나온 요소들로 구성된 문자형 벡터.
  ✓ 앞에서 소개한 다른 문자열 관련 함수들과 마찬가지로 strsplit() 함수 역시 벡터화.
  ✓ 즉, 다음 예에서 보듯이 분할 결과는 문자형 벡터의 리스트.

```
> students <- strsplit(c("Tony, 26, Physics", "James, 25, Economics"), split = ", ")
> students
[[1]]
[1] "Tony"    "26"      "Physics"

[[2]]
[1] "James"     "25"        "Economics"
```

6.1 문자열 시작하기

- 텍스트 변환하기

✓ strsplit() 함수는 주어진 문자형 벡터를 요소별로 처리하여 각 문자열을 분할한 문자형 벡터들로 된 리스트 객체를 반환.

✓ 실제로 어떤 정보를 추출하거나 인식할 때 제일 먼저 하는 과정 중 하나가 바로 분할.

✓ 이렇게 분할한 데이터를 행렬에 넣어 각 열에 적당한 이름을 붙여 주자.

```
> students_matrix <- do.call(rbind, students)
> colnames(students_matrix) <- c("name", "age", "major")
> students_matrix
          name     age         major
[1,]    "Tony"   "26"      "Physics"
[2,]   "James"   "25"    "Economics"
```

✓ 이 행렬을 데이터 프레임으로 변형한 후 각 열을 적합한 타입으로 변경해 보자.

```
> students_df <- data.frame(students_matrix, stringsAsFactors = FALSE)
> students_df$age <- as.numeric(students_df$age)
> students_df
      name    age        major
1     Tony     26      Physics
2    James     25    Economics
```

6.1 문자열 시작하기

- 텍스트 변환하기

✓ 전체 문자열을 글자마다 분할하고 싶을 때는 다음과 같이 split 인수에 아무것도 넣지 않음.

```
> strsplit(c("hello", "world"), split = "")
[[1]]
[1] "h" "e" "l" "l" "o"

[[2]]
[1] "w" "o" "r" "l" "d"
```

6.1 문자열 시작하기

- 텍스트 서식 지정하기

  ✓ 텍스트를 조각 내어 생각해야 하고 형식이 길어질수록 코드를 읽기가 어렵기 때문에 paste()로 연결하는 것이 좋지만은 않을 때가 있음.

  ✓ 예를 들어 students_df의 각 레코드를 다음 형식으로 출력해야 한다고 가정해 보자.

  ```
  #1, name: Tony, age: 26, major: Physics
  ```

  ✓ 이를 paste() 함수를 사용하여 한다면 정말 쉽지 않을 것.

  ```
  > cat(paste("#", 1:nrow(students_df), ", name: ", students_df$name, ", age: ", students_
  df$age, ", major: ", students_df$major, sep = ""), sep = "\n")
  #1, name: Tony, age: 26, major: Physics
  #2, name: James, age: 25, major: Economics
  ```

6.1 문자열 시작하기

- 텍스트 서식 지정하기

✓ sprintf() 함수는 텍스트 서식을 지원하여 문제를 더 나은 방법으로 해결.

```
> cat(sprintf("#%d, name: %s, age: %d, major: %s",
+     1:nrow(students_df), students_df$name, students_df$age, students_df$major),
+     sep = "\n")
#1, name: Tony, age: 26, major: Physics
#2, name: James, age: 25, major: Economics
```

✓ #%d, name: %s, age: %d, major: %s 부분은 서식 템플릿.

✓ %d와 %s는 문자열에 나타나는 입력 인수들이 위치할 자리.

✓ sprintf() 함수는 템플릿 문자열이 끊어지는 것을 방지하고 대입하려는 부분을 함수 인수로 지정하기 때문에 사용하기 쉬움.

✓ 앞 예제에서 %s는 문자열을 나타내고, %d는 숫자(정수)를 나타냄.

✓ sprintf() 함수는 %f를 사용하여 수치형 값의 형식을 지정하는 데 매우 유연.

✓ 예를 들어 %.1f는 숫자를 소수점 첫째 자리로 반올림.

```
> sprintf("The length of the line is approximately %.1fmm", 12.295)
[1] "The length of the line is approximately 12.3mm"
```

6.1 문자열 시작하기

- 텍스트 서식 지정하기

  ✓ 사실 다양한 타입의 값에 서식 지정 구문이 존재.
  ✓ 가장 일반적으로 사용하는 구문은 다음과 같음.

  [표 6-1] 타입에 따른 서식 지정 구문

형식	출력
sprintf("%s", "A")	A
sprintf("%d", 10)	10
sprintf("%04d", 10)	0010
sprintf("%f", pi)	3.141593
sprintf("%.2f", pi)	3.14
sprintf("%1.0f", pi)	3
sprintf("%8.2f", pi)	3.14
sprintf("%08.2f", pi)	00003.14
sprintf("%+f", pi)	+3.141593
sprintf("%e", pi)	3.141593e+00
sprintf("%E", pi)	3.141593E+00

6.1 문자열 시작하기

- 텍스트 서식 지정하기

  ✓ 서식 텍스트에서 %는 특수 문자로, 자리를 표시하는 시작 문자로 해석.
  ✓ 문자열에서 %를 실제 의미(퍼센트 기호)로 사용한다면 어떨까?
  ✓ 서식 텍스트로 해석되는 것을 피하려면 %%를 사용하여 문자 그대로의 %를 구분할 필요가 있음

```
> sprintf("The ratio is %d%%", 10)
[1] "The ratio is 10%"
```

6.1 문자열 시작하기

- 텍스트 서식 지정하기

R에서 파이썬 문자열 함수 사용하기

✓ sprintf() 함수가 강력하지만, 그렇다고 모든 경우에 완벽한 것은 아님.
✓ 예를 들어 일부 변수가 서식에서 여러 번 나와야 할 때는 동일한 인수를 여러 번 써야 함.
✓ 이렇게 하면 코드가 더 복잡하고 수정하기가 어려움.

```
> sprintf("%s, %d years old, majors in %s and loves %s.", "James", 25, "Physics",
"Physics")
[1] "James, 25 years old, majors in Physics and loves Physics."
```

6.1 문자열 시작하기

- 텍스트 서식 지정하기

  ✓ glue 패키지는 R에서 문자열을 다루려고 현재 많이 사용하는 여러 패키지 중 하나.

  ✓ tidyverse(https://www.tidyverse.org/)라는 R 패키지 컬렉션에 포함하여 현재도 활발하게 개발되고 있음.

  ✓ 다음과 같이 손쉽게 설치 가능.

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

  ✓ 다음과 같이 R 변수를 바로 문자열에 전달 가능.

```
> library(glue)
> name <- "Lee"
> glue('My name is {name}.')
My name is Lee.
```

6.1 문자열 시작하기

- 텍스트 서식 지정하기

  ✓ 긴 문자열도 다음과 같이 줄 바꿈을 통해 쉽게 분리하여 작성 가능.
  ✓ 결과적으로 이 문자열들을 하나로 합친 긴 문자열을 생성.

```
> age <- 20
> anniversary <- as.Date("2020-01-01")
> glue('My name is {name},',
+      ' My age next year is {age + 1},',
+      ' My anniversary is {format(anniversary, "%A, %B %d, %Y")}.')
My name is Lee, My age next year is 21, My anniversary is 수요일, 1월, 01, 2020.
```

6.1 문자열 시작하기

- 텍스트 서식 지정하기

  ✓ 이름이 있는 인수를 설정하여 임시 변수도 할당 가능.
  ✓ 즉, 이름으로 된 자리 표시를 사용하여 파이썬 같은 서식 스타일의 문자열 포맷 가능.

```
> glue('My name is {name},',
+      ' My age next year is {age + 1},',
+      ' My anniversary is {format(anniversary, "%A, %B %d, %Y")}.',
+      name = "Lee",
+      age = 20,
+      anniversary = as.Date("2020-01-01"))
My name is Lee, My age next year is 21, My anniversary is 수요일, 1월 01, 2020.
> glue("{name}, {age} years old,",
+      " majors in {major} and loves {major}.",
+      name = "Lee", age = 20, major = "Math")
Lee, 20 years old, majors in Math and loves Math.
```

6.2 날짜/시간 서식

6.2 날짜/시간 서식

- 날짜/시간 서식
  - ✓ Sys.Date() 함수: 현재 날짜를 반환.

```
> Sys.Date()
[1] "2016-02-26"
```

  - ✓ Sys.time() 함수: 현재 시간을 반환.

```
> Sys.time()
[1] "2016-02-26 22:12:25 KST"
```

6.2 날짜/시간 서식

- 날짜/시간 서식

✓ 날짜와 시간에 대한 결과가 마치 문자형 벡터처럼 보이지만, 실제로는 그렇지 않음.

```
> current_date <- Sys.Date()
> as.numeric(current_date)
[1] 16857
> current_time <- Sys.time()
> as.numeric(current_time)
[1] 1456495945
```

✓ 본질적으로 이 결과는 어떤 원점을 기준으로 하는 숫자 값으로, 날짜/시간을 계산하는 특별한 방법이 있음.

✓ 날짜에서 숫자 값은 1970- 01- 01 이후에 경과된 일 수를 의미.

✓ 시간에서 숫자 값은 1970- 01- 01 00:00.00 UTC에서 경과된 초 수를 의미.

6.2 날짜/시간 서식

- 텍스트에서 날짜/시간 분석하기

  ✓ 사용자가 정의한 원점을 기준으로 상대적인 날짜 지정 가능.

```
> as.Date(1000, "1970-01-01")
[1] "1972-09-27"
```

  ✓ 텍스트 형식으로 날짜와 시간 지정 가능.

```
> my_date <- as.Date("2016-02-10")
> my_date
[1] "2016-02-10"
```

6.2 날짜/시간 서식

- 텍스트에서 날짜/시간 분석하기

  ✓ 2016-02-10처럼 문자열로 시간을 표현할 수 있다면 왜 굳이 Date 객체를 만들어야 할까?
  ✓ 이는 Date 객체가 기능이 더 많기 때문.
  ✓ Date 객체는 몇 가지 연산이 가능.
  ✓ 날짜 객체를 가지고 있다 가정할 때 며칠을 더하거나 빼면 새로운 날짜를 얻을 수 있음.

```
> my_date + 3
[1] "2016-02-13"
> my_date + 80
[1] "2016-04-30"
> my_date - 65
[1] "2015-12-07"
```

6.2 날짜/시간 서식

- 텍스트에서 날짜/시간 분석하기

  ✓ 어떤 날짜에서 다른 날짜를 직접 빼면 두 날짜 사이에 며칠이 있는지 알 수 있음.

```
> date1 <- as.Date("2014-09-28")
> date2 <- as.Date("2015-10-20")
> date2 - date1
Time difference of 387 days
```

6.2 날짜/시간 서식

- 텍스트에서 날짜/시간 분석하기

✓ 앞 예제에서 date2 - date1 결과가 메시지처럼 보이지만, 실제로는 숫자 값.
✓ as.numeric()을 사용하여 명시적으로 숫자로 표시 가능.

```
> as.numeric(date2 - date1)
[1] 387
```

✓ 시간도 이와 비슷하지만, as.Time()이라는 함수는 없음.
✓ 텍스트 표현에서 날짜/시간을 만들려면 as.POSIXct()나 as.POSIXlt() 함수 사용 가능.
✓ 두 함수는 포직스(POSIX) 표준에 따른 날짜/시간 객체를 다르게 구현한 것.

```
> my_time <- as.POSIXlt("2016-02-10 10:25:31")
> my_time
[1] "2016-02-10 10:25:31 KST"
```

✓ 이 타입의 객체도 +와 -를 사용하여 간단한 시간 계산이 가능.
✓ 다만 날짜 객체와 달리 일 단위가 아닌 초 단위로 계산.

```
> my_time + 10
[1] "2016-02-10 10:25:41 KST"
> my_time + 12345
[1] "2016-02-10 13:51:16 KST"
> my_time - 1234567
[1] "2016-01-27 03:29:24 KST"
```

6.2 날짜/시간 서식

- 텍스트에서 날짜/시간 분석하기

  - ✓ 데이터에 날짜나 시간에 관한 문자열 표현이 주어지면 이를 날짜나 시간 객체로 변환해야만 계산에 활용 가능.
  - ✓ 하지만 데이터에서 주어진 원본 데이터가 항상 as.Date()나 as.POSIXlt() 함수에서 바로 인식할 수 있는 형태는 아닐 것.
  - ✓ 이럴 경우 sprintf() 함수와 같이 날짜나 시간의 특정 부분을 나타내는 데 특수 문자를 사용하여 자리를 표시할 필요가 있음.
  - ✓ 예) 입력이 2015.07.25인 경우 문자열 서식이 제공되지 않으면 as.Date()는 오류 발생.

```
> as.Date("2015.07.25")
Error in charToDate(x) : 문자열이 표준서식을 따르지 않습니다
```

  - ✓ 서식 문자열을 템플릿으로 사용하여 as.Date() 함수에 주어지는 문자열을 날짜로 분석하는 방법 지정 가능.

```
> as.Date("2015.07.25", format = "%Y.%m.%d")
[1] "2015-07-25"
```

  - ✓ 날짜/시간에 대한 문자열도 입력이 표준 서식을 따르지 않을 때는 as.POSIXlt() 함수가 인식 가능하도록 서식 문자열을 지정해 주어야 함.

```
> as.POSIXlt("7/25/2015 09:30:25", format = "%m/%d/%Y %H:%M:%S")
[1] "2015-07-25 09:30:25 KST"
```

6.2 날짜/시간 서식

- 텍스트에서 날짜/시간 분석하기

  ✓ strptime() 함수: 문자열을 날짜/시간으로 변형하는 데 사용하는(좀 더 직접적인) 함수.

```
> strptime("7/25/2015 09:30:25", "%m/%d/%Y %H:%M:%S")
[1] "2015-07-25 09:30:25 KST"
```

  ✓ as.POSIXlt() 함수는 문자를 입력하는 strptime() 함수의 래퍼(wrapper)라고 할 수 있음.

  ❖ 래퍼 함수는 함수 내부에 실제 코드를 구현한 것이 아니라, 원형 함수를 호출하는 코드로 되어 있음.

  ✓ strptime() 함수는 항상 서식 문자열이 필요.

  ✓ as.POSIXlt() 함수는 특별한 서식 문자열이 주어지지 않으면 표준 서식으로 처리.

6.2 날짜/시간 서식

- 텍스트에서 날짜/시간 분석하기

  ✓ 날짜 혹은 날짜/시간 객체 역시 벡터.
  ✓ as.Date() 함수에 문자형 벡터를 입력할 수 있는데, 이때는 날짜 벡터를 결과로 얻음.

```
> as.Date(c("2015-05-01", "2016-02-12"))
[1] "2015-05-01" "2016-02-12"
```

  ✓ 관련 연산 역시 벡터화.
  ✓ 다음 코드에서 연속된 정수를 날짜 객체에 더하면 연속된 날짜를 얻을 수 있음.

```
> as.Date("2015-01-01") + 0:2
[1] "2015-01-01" "2015-01-02" "2015-01-03"
```

6.2 날짜/시간 서식

- 텍스트에서 날짜/시간 분석하기

  ✓ 같은 기능이 날짜/시간 객체에도 적용.

```
> strptime("7/25/2015 09:30:25", "%m/%d/%Y %H:%M:%S") + 1:3
[1] "2015-07-25 09:30:26 KST" "2015-07-25 09:30:27 KST" "2015-07-25 09:30:28 KST"
```

  ✓ 때로는 날짜나 시간을 정수 형태로 표시하는 경우도 있음.

  ✓ 예를 들어 다음 코드는 20150610을 분석.

```
> as.Date("20150610", format = "%Y%m%d")
[1] "2015-06-10"
```

  ✓ 다음은 20150610093215를 구문 분석하는 서식을 나타내는 템플릿을 지정.

```
> strptime("20150610093215", "%Y%m%d%H%M%S")
[1] "2015-06-10 09:32:15 KST"
```

6.2 날짜/시간 서식

- 텍스트에서 날짜/시간 분석하기

✓ 다음과 같은 데이터 프레임에서 날짜/시간 객체를 분석하는 예제를 생각해 보자.

```
> datetimes <- data.frame(
+     date = c(20150601, 20150603),
+     time = c(92325, 150621))
```

✓ datetimes 열에서 paste0()을 사용하고, 앞 예제에서 이용한 템플릿으로 strptime()을 직접 호출하면 첫 번째 요소가 서식에 맞지 않아 실측값이 표시.

```
> dt_text <- paste0(datetimes$date, datetimes$time)
> dt_text
[1] "2015060192325"  "20150603150621"
> strptime(dt_text, "%Y%m%d%H%M%S")
[1] NA                        "2015-06-03 15:06:21 KST"
```

✓ 문제는 92325에 있으며 이를 092325로 변경해야 함.

✓ 앞에 0이 있는지 확인하려면 sprintf()를 사용.

```
> dt_text2 <- paste0(datetimes$date, sprintf("%06d", datetimes$time))
> dt_text2
[1] "20150601092325" "20150603150621"
> strptime(dt_text2, "%Y%m%d%H%M%S")
[1] "2015-06-01 09:23:25 KST" "2015-06-03 15:06:21 KST"
```

6.2 날짜/시간 서식

- 날짜/시간을 문자열로 서식 변환하기

  ✓ 날짜 객체를 만들고 그것을 출력하면 항상 표준 형식으로 표현.

```
> my_date
[1] "2016-02-10"
```

  ✓ as.character() 함수: 날짜를 표준 형식의 문자열로 변환.

```
> date_text <- as.character(my_date)
> date_text
[1] "2016-02-10"
```

  ✓ 출력 결과는 my_date와 같지만, 이 문자열은 이제 단순한 텍스트이며 더 이상 날짜 계산을 지원하지 않음.

```
> date_text + 1
Error in date_text + 1 : non-numeric argument to binary operator
```

6.2 날짜/시간 서식

- 날짜/시간을 문자열로 서식 변환하기

✓ 때로는 날짜를 표준 형식이 아닌 다른 방법으로 표현해야 할 때가 있음.

```
> as.character(my_date, format = "%Y.%m.%d")
[1] "2016.02.10"
```

✓ 사실 as.character() 함수 내부에서는 format() 함수를 직접 호출.

✓ format() 함수를 사용하면 정확히 같은 결과를 얻을 것

✓ 대부분 이 방식을 추천.

```
> format(my_date, "%Y.%m.%d")
[1] "2016.02.10"
```

✓ 날짜/시간 객체에도 마찬가지로 적용이 가능.

✓ 원래 자리 표시자 외에 더 많은 텍스트를 포함하도록 사용자가 서식 정의 가능.

```
> my_time
[1] "2016-02-10 10:25:31 PST"
> format(my_time, "date: %Y-%m-%d, time: %H:%M:%S")
[1] "date: 2016-02-10, time: 10:25:31"
```

6.3 정규 표현식 사용하기

6.3 정규 표현식 사용하기

- 정규 표현식 사용하기

  - ✓ 개방형 웹 사이트나 인증을 요구하는 데이터베이스에서 데이터를 내려받아야 할 때가 있음.
  - ✓ 데이터 소스는 다양한 형식의 데이터를 제공하며, 대부분의 데이터는 잘 구성되어 있음.
  - ✓ 예를 들어 많은 경제 및 금융 데이터베이스는 CSV 형식의 데이터를 제공.
  - ✓ CSV 형식은 테이블 형태의 데이터를 표현하는 데 널리 지원되는 텍스트 형식.
  - ✓ 일반적인 CSV 형식은 다음과 같음.

```
아이디, 이름, 점수
1, A, 20
2, B, 30
3, C, 25
```

  - ✓ R에서 이러한 형식은 데이터 프레임으로 표현.
  - ✓ CSV 파일을 올바른 헤더와 데이터 타입을 가진 데이터 프레임으로 읽어 들일 때는 `read.csv()` 함수를 호출하면 편리.

6.3 정규 표현식 사용하기

- 정규 표현식 사용하기

  - ✓ 모든 데이터 파일이 잘 조직되어 있는 것은 아님.
  - ✓ 제대로 구성되지 않은 데이터를 처리 하기는 매우 어려움.
  - ✓ read.table()이나 read.csv() 함수 같은 내장 함수는 대부분 잘 동작하지만, 이렇게 형식이 없는 데이터에는 전혀 도움이 되지 않을 수 있음.
  - ✓ 예를 들어 다음과 같이 CSV와 유사한 형식으로 구성된 원시 데이터(messages.txt)를 분석하려고 read.csv() 함수를 호출할 때는 조심해야 함.

messages.txt

```
2014-02-01,09:20:25,James,Ken,Hey, Ken!
2014-02-01,09:20:29,Ken,James,Hey, how are you?
2014-02-01,09:20:41,James,Ken, I'm ok, what about you?
2014-02-01,09:21:03,Ken,James,I'm feeling excited!
2014-02-01,09:21:26,James,Ken,What happens?
```

6.3 정규 표현식 사용하기

- 정규 표현식 사용하기

✓ 아마 이 파일을 읽을 때는 다음과 같이 잘 구성된 형태의 데이터 프레임을 기대할 것.

	Date	Time	Sender	Receiver	Message
1	2014-02-01	09:20:25	James	Ken	Hey, Ken!
2	2014-02-01	09:20:29	Ken	James	Hey, how are you?
3	2014-02-01	09:20:41	James	Ken	I'm ok, what about you?
4	2014-02-01	09:21:03	Ken	James	I'm feeling excited!
5	2014-02-01	09:21:26	James	Ken	What happens?

6.3 정규 표현식 사용하기

- 정규 표현식 사용하기

✓ 아무 정보 없이 read.csv() 함수를 사용하면 결과가 원하는 대로 나오지 않음.
✓ 이 csv 파일은 구분 기호로 잘못 해석될 수 있는 쉼표가 추가로 들어 있음
✓ 다음은 이 원시 텍스트로부터 변환된 데이터 프레임.

```
> read.csv("data/messages.txt", header = FALSE)
        V1       V2      V3    V4                    V5                 V6
1  2/1/14  9:20:25   James   Ken                   Hey               Ken!
2  2/1/14  9:20:29     Ken James                   Hey      how are you?
3  2/1/14  9:20:41   James   Ken                 I'm ok  what about you?
4  2/1/14  9:21:03     Ken James I'm feeling excited!
5  2/1/14  9:21:26   James   Ken        What happens?
```

6.3 정규 표현식 사용하기

- 정규 표현식 사용하기

  - ✓ 이 문제를 해결할 수 있는 방법은 다양.
  - ✓ 각 줄마다 strsplit() 함수를 사용하여 처음 요소 몇 개는 수동으로 제외하고, 여러 조각으로 분리된 다른 부분은 하나로 붙일 수 있음.
  - ✓ 가장 강력한 방법은 정규 표현식(https://en.wikipedia.org/wiki/Regular_expression)을 사용하는 것.
  - ✓ 정규 표현식은 텍스트와 일치하는 패턴을 서술하고, 텍스트에서 원하는 부분을 추출.
  - ✓ 어떤 과일의 개수나 상태를 나타내는 텍스트(fruits.txt)를 처리한다고 가정하자.

fruits.txt

```
apple: 20
orange: missing
banana: 30
pear: sent to Jerry
watermelon: 2
blueberry: 12
strawberry: sent to James
```

6.3 정규 표현식 사용하기

- 정규 표현식 사용하기

  - ✓ 이제 상태 정보가 아닌 개수 정보를 갖는 과일을 고른다고 하자.
  - ✓ 숫자가 있는 과일과 숫자가 없는 과일로 구분해볼까?
  - ✓ 일반적으로 특정 패턴과 일치하는 텍스트와 그렇지 않은 텍스트를 구분할 필요가 있음.
  - ✓ 정규 표현식은 확실히 여기에 적합한 기술.
  - ✓ 정규 표현식은 두 단계를 사용하여 문제를 해결.
    - ❖ 첫 번째로 텍스트와 일치하는 패턴을 탐색.
    - ❖ 두 번째로 패턴을 그룹화하여 필요한 정보를 추출.

6.3 정규 표현식 사용하기

- 문자열 패턴 찾기

  - ✓ 콜론(:)과 공백이 뒤에 오는 단어로 시작하고, 단어나 다른 기호가 아닌 정수로 끝나는 모든 라인 탐색.
  - ✓ 정규 표현식은 패턴을 표현하는 일련의 기호를 제공.
  - ✓ 앞에서 설명한 패턴은 `^\w+:\s\d+$`처럼 표현 가능.
  - ✓ 여기서는 기호의 클래스를 나타내는 데 메타 기호를 사용.

    ^ 줄이 시작하는 곳.
    \w 단어 글자를 나타냄.
    \s 공백 문자.
    \d 숫자 문자.
    $ 줄이 끝나는 곳.
    \w+ 한 단어 이상의 글자.
    : 정확히 단어 다음에 나올 것으로 예상되는 기호.
    \d+ 하나 이상의 숫자 문자.

6.3 정규 표현식 사용하기

- 문자열 패턴 찾기

✓ 좀 더 구체적으로 이 패턴은 abc: 123 같은 형태의 줄은 찾고, 그렇지 않은 줄은 제외.
✓ R에서 원하는 경우를 골라내려면 grep() 함수로 패턴과 일치하는 문자열을 구함.

```
> fruits <- readLines("data/fruits.txt")
> fruits
[1] "apple: 20"                "orange: missing"          "banana: 30"
[4] "pear: sent to Jerry"      "watermelon: 2"            "blueberry: 12"
[7] "strawberry: sent to James"
> matches <- grep("^\\w+:\\s\\d+$", fruits)
> matches
[1] 1 3 5 6
```

✓ 주의) \를 사용할 때 이스케이프 문자와 구분하고자 \\라고 써야 함.
✓ 이제 fruits를 matches 객체로 필터링할 수 있음.

```
> fruits[matches]
[1] "apple: 20"     "banana: 30"    "watermelon: 2" "blueberry: 12"
```

✓ 이제 원하는 패턴의 줄과 그렇지 않은 줄을 성공적으로 구분 가능.
✓ 패턴과 일치하는 줄은 선택되고, 패턴과 일치하지 않는 줄은 생략.

6.3 정규 표현식 사용하기

- 문자열 패턴 찾기

✓ 주의) ^로 시작하고 $로 끝나는 패턴을 지정.
✓ 부분적으로 일치하는 것은 원하지 않기 때문.
✓ 정규 표현식은 기본적으로 부분 패턴 매칭을 수행.
✓ 즉, 문자열 일부가 패턴과 일치하면 전체 문자열이 패턴과 일치하는 것으로 간주.
✓ 예를 들어 다음 코드는 패턴 2개와 각각 일치하는 문자열을 찾음.

```
> grep("\\d", c("abc", "a12", "123", "1"))
[1] 2 3 4
> grep("^\\d$", c("abc", "a12", "123", "1"))
[1] 4
```

✓ 첫 번째 패턴은 임의의 숫자(부분 매칭)를 포함하는 문자열과 일치.
✓ ^나 $가 있는 두 번째 패턴은 숫자가 한 자리만 있는 문자열과 일치.
✓ 일단 패턴이 올바르게 동작하면 그룹을 사용하여 데이터를 추출하는 다음 단계로 넘어감.

6.3 정규 표현식 사용하기

- 그룹을 사용하여 데이터 추출하기

  ✓ 패턴 문자열에서 괄호를 사용하여 텍스트에서 추출할 부분을 구분하는 표시를 만들 수 있음.
  ✓ 이 문제는 패턴을 그룹 2개가 있는 (\w+):\s(\d+)로 수정하여 해결 가능.
  ✓ 하나는 \w+와 일치하는 부분, 즉 과일 이름을 나타내는 부분.
  ✓ 다른 하나는 \d+와 일치하는 과일 개수를 나타내는 부분.

6.3 정규 표현식 사용하기

- 그룹을 사용하여 데이터 추출하기

  ✓ stringr 패키지를 활용하면 정규 표현식을 훨씬 쉽게 사용 가능.
  ✓ 그룹이 있는 패턴을 사용하여 str_match() 함수를 호출.

```
> library(stringr)
> matches <- str_match(fruits, "^(\\w+):\\s(\\d+)$")
> matches
```

```
                [,1]          [,2]    [,3]
[1,]     "apple: 20"       "apple"    "20"
[2,]              NA            NA      NA
[3,]    "banana: 30"      "banana"    "30"
[4,]              NA            NA      NA
[5,] "watermelon: 2"  "watermelon"     "2"
[6,] "blueberry: 12"   "blueberry"    "12"
[7,]              NA            NA      NA
```

6.3 정규 표현식 사용하기

- 그룹을 사용하여 데이터 추출하기

  ✓ 결과가 한 행 이상 있는 행렬 형태.
  ✓ 텍스트에서 괄호 안의 그룹을 추출하여 2~3열에 배치.
  ✓ 이 문자형 행렬을 올바른 헤더와 데이터 유형을 가진 데이터 프레임으로 쉽게 변형 가능.

```
> # 데이터 프레임으로 변형한다
> fruits_df <- data.frame(na.omit(matches[, -1]), stringsAsFactors = FALSE)
> # 헤더를 추가한다
> colnames(fruits_df) <- c("fruit","quantity")
> # quantity의 문자를 정수로 변환한다
> fruits_df$quantity <- as.integer(fruits_df$quantity)
```

```
> fruits_df
          fruit    quantity
1         apple          20
2        banana          30
3    watermelon           2
4     blueberry          12
```

6.3 정규 표현식 사용하기

- 그룹을 사용하여 데이터 추출하기

  ✓ 앞에서 언급한 메타 기호 외에 다음과 같은 것들도 유용.

  [0-9] 0에서 9까지 단일 정수.
  [a-z] a에서 z까지 소문자 하나.
  [A-Z] A에서 Z까지 대문자 하나.
  . 임의의 단일 기호.
  * 0번 혹은 한 번 이상 나올 수 있는 패턴.
  + 한 번 이상 나오는 패턴.
  {n} n번 나오는 패턴.
  {m, n} 최소 m번, 최대 n번까지 나오는 패턴.

6.3 정규 표현식 사용하기

- 그룹을 사용하여 데이터 추출하기

  ✓ 메타 기호를 사용하면 문자열 데이터를 쉽게 확인하거나 필터링할 수 있음.
  ✓ 예를 들어 두 나라의 전화번호가 서로 섞여 있다고 가정하자.
  ✓ 한 국가의 전화번호 패턴이 다른 국가의 전화번호 패턴과 다를 때는 정규 표현식을 사용하여 두 가지 범주로 나눌 수 있음.

```
> telephone <- readLines("data/telephone.txt")
> telephone
[1] "123-23451" "1225-3123" "121-45672" "1332-1231" "1212-3212" "123456789"
```

  ✓ 데이터에 예외가 있을 수 있다는 점을 기억하자.
  ✓ 예외인 번호에는 중간에 -기호가 없음.
  ✓ 예외가 아닐 때는 두 가지 유형의 전화번호 패턴이 있다는 것을 쉽게 파악할 수 있음.

```
> telephone[grep("^\\d{3}-\\d{5}$", telephone)]
[1] "123-23451" "121-45672"
> telephone[grep("^\\d{4}-\\d{4}$", telephone)]
[1] "1225-3123" "1332-1231" "1212-3212"
```

6.3 정규 표현식 사용하기

- 그룹을 사용하여 데이터 추출하기

  - ✓ grepl() 함수: 각 요소가 패턴과 일치하는지 여부를 나타내는 논리 벡터를 반환하므로 예외인 경우를 찾는 데 매우 유용.
  - ✓ grepl() 함수를 사용하여 주어진 패턴과 일치하지 않는 모든 레코드 선택 가능.

```
> telephone[!grepl("^\\d{3}-\\d{5}$", telephone) & !grepl("^\\d{4}-\\d{4}$", telephone)]
[1] "123456789"
```

  - ✓ 앞 코드에서는 기본적으로 두 패턴과 일치하지 않는 모든 레코드를 예외로 간주.
  - ✓ 확인해야 할 레코드가 수백만 개 있다고 하자.
  - ✓ 예외적인 경우는 어떤 임의의 형식이든 가능하기 때문에 유효한 모든 레코드를 제외하고 잘못된 레코드를 찾는 방법을 사용하면 더욱 안정적.

6.3 정규 표현식 사용하기

- 사용자 정의 방식으로 데이터 읽기

✓ 먼저 원형 데이터에서 전형적인 어느 한 줄을 살펴보자.

```
2014-02-01,09:20:29,Ken,James,Hey, how are you?
```

✓ 즉, 날짜와 시간, 보낸 사람, 받는 사람, 메시지가 쉼표로 구분.
✓ 메시지 안의 쉼표는 구분 문자로 해석해서는 안 됨.
✓ 정규 표현식은 이전 예제처럼 이러한 목적에 부합.
✓ 하나 이상의 기호에서 패턴이 같다는 것을 나타내려면 기호 식별자 뒤에 + 기호 추가.
✓ 예를 들어 \d+는 "0"과 "9" 사이의 숫자가 하나 이상인 문자열을 나타냄.
✓ "1", "23", "456" 모두 이 패턴과 일치하지만, "word"는 그렇지 않음.
✓ 어떤 패턴이 나올 수도 있고(+) 나오지 않을 수(0)도 있음.
✓ 특정 패턴이 한 번 이상(+) 나올 수도 있고, 아예 나오지 않을 수(0)도 있다는 것을 나타내려고 기호 식별자 뒤에 * 기호 배치.

6.3 정규 표현식 사용하기

- 사용자 정의 방식으로 데이터 읽기

✓ 다음은 우리가 파악할 수 있는 그룹 패턴.

```
(\d+-\d+-\d+),(\d+:\d+:\d+),(\w+),(\w+),\s*(.+)
```

✓ 먼저 과일 예제에서 했던 방식으로 readLines() 함수를 사용해서 원형 텍스트를 읽어야 함.

```
> messages <- readLines("data/messages.txt")
```

✓ 추출하고자 하는 텍스트와 정보를 나타내는 패턴을 사용해야 함.

```
> pattern <- "^(\\d+-\\d+-\\d+),(\\d+:\\d+:\\d+),(\\w+),(\\w+),\\s*(.+)$"
> matches <- str_match(messages, pattern)
> messages_df <- data.frame(matches[, -1])
> colnames(messages_df) <- c("Date", "Time", "Sender", "Receiver", "Message")
```

```
> messages_df
        Date      Time    Sender    Receiver                  Message
1 2014-02-01  09:20:25     James         Ken                Hey, Ken!
2 2014-02-01  09:20:29       Ken       James          Hey, how are you?
3 2014-02-01  09:20:41     James         Ken    I'm ok, what about you?
4 2014-02-01  09:21:03       Ken       James       I'm feeling excited!
5 2014-02-01  09:21:26     James         Ken              What happens?
```

6.4 마치며

6.4 마치며

- 마치며

  ✓ 이 장에서는 문자형 벡터를 조작하고 날짜/시간 객체와 문자열 표현 간 변환을 하는 여러 가지 내장 함수를 학습했다.

  ✓ 문자열 데이터를 확인하고 필터링하고 원본 텍스트에서 원하는 정보를 추출하려고 매우 강력한 도구인 정규 표현식의 기본 개념을 배웠다.

  ✓ 이제 지금까지 배운 방법들을 사용하여 기본 데이터 구조로 작업할 수 있다.