Chapter 1

기초 프로그래밍

우리는 독자가 R을 과학적 분석을 위해 필요한 일련의 프로세스를 진행하는 프로그래밍 언어로서 이해하기를 권장합니다. 따라서, 이 챕터에서는 이러한 프로세스를 수행하는데 필요한 프로그래밍적 요소들에 대해서 다룰 것입니다.

1.1 에러 그리고 정확성

우리는 이러한 내용들을 설명하기 전에 에러와 경고에 대한 내용을 먼저 다룰 것입니다. 그 이유는 에러와 경고는 프로세스의 수행중 연구자가 원하는 방향과는 다른 방향으로 진행되고 있음을 알려주는 표시이기 때문입니다. 또다른 이유는 분석을 위한 프로그래밍은 논리적인 절차를 어떻게 잘 구성하는가에 따라서 그 효율성과 프로그램의 가독성이 달라지기 때문입니다.

먼저 아래와 같이 0이라는 값을 0으로 나누어 보는 간단한 예를 들어봅니다.

```
> 0/0
[1] NaN
```

아하! 이런 경우에는 NaN (즉, Not a number 라는 약자)를 알려줍니다. 그 이유는 수학적으로 0으로 어떠한 값도 나눌 수 없기 때문입니다. 그리고, 이렇게 NaN 이라는 값이 포함된 어떠한 연산도 수행될 수 없기 때문에 NaN이라는 값을 출력하게 됩니다.

```
> x <- 0/0
> x + 3
[1] NaN
```

수학적 정의하에 올바른 계산을 하고자 한다면 아래와 같이 해야 할 것입니다.

```
> 0/1
[1] 0
>
```

그런데, 사용자의 실수로 인하여 아래와 같이 3을 0으로 나누었다고 가정합니다. 조금더 고급수학을 다루게 된다면 $n/0$ 이라는 것은 ∞ (무한값)이라는 것을 알고 계실 것입니다. R은 이러한 연산을 기본값으로 하고 있습니다.

```
> 3/0
[1] Inf
> pi/0
[1] Inf
```

이렇게 무한값을 포함하고 있는 어떠한 연산 역시 무한값을 돌려줍니다.

```
> x <- 3/0
> x + 1.0
[1] Inf
```

따라서, 이러한 에러와 관계된 부분을 정확히 알고 R을 사용해야 합니다.

```
> 1/0 + 1/0
[1] Inf
> 1/0 - 1/0
[1] NaN
```

NaN 이라는 약자가 이전에 설명한 NA와 혼돈스러울 수 있기 때문에 한 번 설명합니다. NaN은 숫자가 아닌 것을 말하는 것이고, NA는 존재하지 않는 값을 의미하는 것입니다.

그럼, R은 어느정도로 정확한 수치를 돌려줄까요? 아래와 같이 1을 어떠한 수로 나누어 봅니다.

```
> 1/1
[1] 1
> 1/10
[1] 0.1
> 1/100
[1] 0.01
```

계산이 잘 되는것 같습니다. 이제 좀 더 큰수를 사용해봅니다.

```
> 1/1e300
[1] 1e-300
```

여기에서 1e300 이라는 표현은 10의 300자승이라는 사이언티픽 표현입니다.

```
> 1/1e309
[1] 0
> 1/1e308
[1] 1e-308
```

아하! 1e309 이상의 숫자를 사용하면 이 연산은 0 으로 간주됨을 알 수 있습니다. 따라서, 사용자는 이보다 큰 숫자를 사용할 때 주의를 기울여야 합니다. 일반적으로는 R 은 아래와 같은 범위내에서 안정적인 연산을 하게 됩니다.

```
> .Machine$double.xmin
[1] 2.225074e-308
> .Machine$double.xmax
[1] 1.797693e+308
```

그럼 아래의 경우에는 어떤 것일까요?

```
> log(0)
[1] -Inf
```

위의 연산은 수학적으로 log의 값이 0 일때 음의 무한값을 가지도록 정의되어 있기 때문에 올바른 연산입니다. 그러나, 아래와 같이 log에 -1의 값을 넣으면 어떻게 될까요?

```
> log(-1)
[1] NaN
Warning message:
In log(-1) : NaNs produced
>
```

(본 메시지는 현 한국어 3.0.0 버전으로는 아래와 같으며, R-3.1.0에 더 정확한 메시지로 교정하겠습니다 – 사용에 불편을 드려 죄송합니다).

```
> log(-1)
[1] NaN
경고 메세지가 손실되었습니다
In log(-1): NaNs가 생성되었습니다
>
```

R은 이렇게 수학적으로 정의되어 있지 않은 연산에 대해서 NaN을 표시하고 경고메시지를 보여줍니다. 이러한 경고메시지가 왜 발생했는지 아는 것은 매우 중요합니다.

만약, 어떠한 연산을 수행했을때 아주 많은 에러가 생겨서 이들을 확인해보고자 한다면 아래와 같이 warnings() 함수를 이용하시길 바랍니다.

```
> warnings()
Warning message:
In log(-1) : NaNs produced
>
```

(본 메시지는 현 한국어 3.0.0 버전으로는 아래와 같으며, R-3.1.0에서 오탈자를 교정하겠습니다 – 사용에 불편을 드려 죄송합니다).

```
> warnings()
경고 메세지:
In log(-1): NaNs가 생성되었습니다
>
```

그렇다면, 이번에는 수학연산을 하는 함수에 문자를 입력해 보면 어떤 현상이 발생하나요? 아래의 결과를 살펴보세요.

```
> a <- "a"
> a
[1] "a"
> log(a)
Error in log(a) : Non-numeric argument to mathematical function
>
```

수학함수에 요구되어지는 숫자값이 입력되지 않다면서 연산자체를 거부합니다. 이를 에러라고 합니다.

여기에서 보았듯이 경고와 에러는 서로 다른 것입니다. 요약하면, 경고라는 것은 연산은 수행되지만 어떠한 수학적 정의에 맞지 않는 것이며, 에러라는 것은 발생하지 말아야할 일이 일어났다는 것을 의미합니다.

이러한 경고와 에러를 다루는 것은 실제적인 프로그래밍을 할 때 매우 중요합니다. R은 이러한 것들을 다루기 위한 디버거라는 도구와 에러핸들링이라는 기능을 제공하고 있습니다. 디버거의 사용에 대해서는 현재 익숙해지기 위한 기초단계에 해당되지 않는 기술적 요소이기 때문에 이곳에서는 다루지 않고, "패키지 제작"이라는 챕터에서 자세히 설명하도록 하겠습니다. 그러나, 에러핸들링 기능은 이곳 기초 프로그래잉에서 설명할 것입니다.

본 섹션의 내용을 이해했다면 여러분은 이제 프로그래밍을 시작할 준비가 되었습니다.

1.2 프로그래밍과 함수

우리가 중학교에서 수학시간에 함수라는 개념을 배울때 블랙박스에 비유하는 것은 잘 알고 계실 것입니다. 이 블랙박스에 어떠한 입력을 넣어주면, 어떤 특정한 프로세스에 의하여 처리된 결과가 블랙박스의 출력으로 나오게 됩니다. 예를들어 x 라는 것은 3.532 이라는 값을 가지고 있는데, 이를 반올림하는 경우를 생각해 봅니다. R은 round()라는 함수를 제공합니다.

```
> x <- 3.532
> round(x)
[1] 4
```

그런데, 내가 원하고자 했던 것은 실제로 소수점 두번째 자리에서 반올림하는 것입니다.

```
> round(x, 1)
[1] 3.5
```

흠... 어쩌다 보니 소수점 세번째 자리에서 반올림 한 숫자를 쓰는 것이 필요하게 되어 아래와 같이 합니다.

```
> round(x, 2)
[1] 3.53
```

내가 수행하고자 하는 어떤 것을 round()라는 블랙박스와 이 블랙박스에 반올림 자리수라는 인자를 이용하여 조절할 수 있었습니다.

여기에서 보는 것과 같이 함수를 작성하기 위해서는 입력, 목적, 출력이라는 세가지 요소가 필요합니다. R에서 제공하는 다양한 함수들은 실제로 이러한 방식으로 작성되어 있으며, 단지 사용자의 편의를 고려하여 라이브러리라고 불리는 함수들의 묶음을 제공하는 것입니다. 즉, 반올림을 해주는 round(), 어떤 구간을 동일하게 나누어 주는 cut()이라는 함수, 작업디렉토리에 어떤 파일들이 있는가를 보여주는 ls()함수들이 이렇게 미리 작성되어 제공되어지는 내장함수들이라고 합니다.

그러나, 사용자의 목적에 따라서 내장함수만으로는 더 이상 작업을 할 수 없는 경우가 있습니다. 이럴때 사용자 자신이 원하는 함수를 작성하여 사용하는 블랙박스를 사용자정의 함수라고 합니다. 이렇게 사용자정의 함수를 정의하는데 사용되는 문법은 아래와 같습니다.

```
fn <- function(args){
        expressions
}
```

예를들어, 내가 만드는 함수의 목적이 주어진 두개의 숫자들을 이용하여 더하기를 수행한 뒤 그 결과를 돌려준다고 가정합니다. R을 이용하여 아래와 같이 할 수 있는데, 내가 원하는 것은 plus()라는 사용자정의 함수를 만들어 보는 것입니다.

```
> x <- 1
> y <- 3
> x + y
[1] 4
```

그렇다면 콘솔상에서 아래와 같이 입력합니다.

```
> file.create("plus.R")
> file.edit("plus.R")
# 여기부터는 R이 아닌 R이 열어준 plus.R이라는 파일명을 가지는 외부 파일임.

plus <- function(x1, x2){
        y <- x1 + x2
        return(y)
}

# 저장후 종료 하면 다시 R로 돌아감.
> # 저장한 코드를 실행하기 위해서 아래와 같이 함.
> source("plus.R")
>
```

이제 R 콘솔에서 내가 만든 plus()라는 함수를 사용해 봅니다.

```
> plus(3,2)
[1] 5
> plus(10,30)
[1] 40
```

아하! 잘 됩니다. 그럼 아래와 같이 따라해봅니다.

```
> plus(x1=3, x2=5)
[1] 8
```

이렇게 사용자가 함수의 정의부분에 사용한 인자를 보통 formal argument (형식 인자)라고 합니다. 그리고 위와 같이 정의된 인자의 이름을 함께 이용하여 사용할 수 있습니다. 이 때 이렇게 사용하는 인자를 named argument (지시된 인자)와 함께 사용한다고 합니다. 지시된 인자를 사용하는 이유는 함수의 정의에 사용되는 인자의 개수는 매우 많을 수도 있기 때문에 사용의 혼돈을 피하기 위해서 입니다. 따라서, 우리는 어떤 함수를 사용할 때 지시된 인자명과 함께 사용하기를 권장합니다.

그런데, 이 함수는 정의할 때 두개의 인자를 필요로 했습니다. 따라서, 아래와 같이 하나만 사용이 되거나 아무것도 입력되지 않는다는 아래와 같은 메시지를 보여주게 됩니다.

```
> plus(x1=3)
Error in x1 + x2 : 'x2' is missing
> plus()
Error in x1 + x2 : 'x1' is missing
```

이렇게 어떤 함수에서 사용되는 인자들의 목록을 확인해 보고 싶다면 arg() 함수를 이용하길 바랍니다.

```
> args(plus)
function (x1, x2)
NULL
```

만약 아래와 같이 함수의 이름만 입력하게 된다면 아래와 같이 작성된 사용자함수의 소스코드를 볼 수 있습니다.

```
> plus
function(x1, x2){
y <- x1 + x2
return(y)
}
>
```

따라서, 사용자가 R의 소스코드가 어떻게 작성되어 있는지 확인하고 싶다면 사용되는 함수명을 입력하면 됩니다. 예를들어, 현재 세션의 있는 R의 객체를 삭제해주는 명령어인 rm() 이라는 함수의 소스코드를 확인해 보고 싶다면 아래와 같이 단순히 rm이라고 입력합니다.

```
> rm
function (..., list = character(), pos = -1, envir = as.environment(pos),
    inherits = FALSE)
{
    dots <- match.call(expand.dots = FALSE)$...
    if (length(dots) && !all(sapply(dots, function(x) is.symbol(x) ||
```

```
        is.character(x))))
        stop("... must contain names or character strings")
    names <- sapply(dots, as.character)
    if (length(names) == 0L)
        names <- character()
    list <- .Primitive("c")(list, names)
    .Internal(remove(list, envir, inherits))
}
<bytecode: 0x9fd5864>
<environment: namespace:base>
>
```

이렇게 소스를 볼 수 있는 기능은 추후에 전산통계를 배우고자 하는 학생 또는 소프트웨어 개발자들에게 매우 도움이 될 것입니다.

참고사항 어떤 함수의 소스코드를 알아보고자 kk, mean, sapply를 각각 입력했다고 한다면 각각의 메세지가 다르게 나타나는 것을 확인할 수 있습니다. 그러나, 실제로는 객체의 유형이 무엇인지 잘 알아야 합니다. 현재 kk 라는 것은 한 번도 사용한 적이 없는 객체의 이름이기 때문에 아래와 같이 나옵니다.

```
> kk
에러: 객체 'kk'를 찾을 수 없습니다
>
```

그러나, sapply()와 같이 R에서 미리 제공하는 내장함수의 경우에는 이미 함수가 정의되어 있는 것이기 때문에 아래와 같이 나옵니다.

```
> sapply
function (X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
{
   FUN <- match.fun(FUN)
   answer <- lapply(X = X, FUN = FUN, ...)
   if (USE.NAMES && is.character(X) && is.null(names(answer)))
       names(answer) <- X
   if (!identical(simplify, FALSE) && length(answer))
       simplify2array(answer, higher = (simplify == "array"))
   else answer
}
<bytecode: 0x00000000147a3290>
<environment: namespace:base>
>
```

본 섹션에서는 아주 단순한 예제를 이용하여 함수의 개념과 사용자 정의함수를 작성하는 방법을 설명했습니다. 이렇게 단위의 함수들이 모이고 모여서 어떤 하나의 큰 작업을 수행하도록 할 수 있습니다. 그리고, 몇 개의 블랙박스가 서로 어떻게 연결 및 구성되며, 형상관리는 어떻게 할 것인지, 사용자의 편의를

고려해야 하는지, 얼마나 프로그램을 자잘하게 쪼개야 하는 등의 다양한 관점을 살리는 부분은 프로그램을 하면서 경험속에서 얻어지게 됩니다. 여기에서는 소프트웨어의 개발을 이야기 하는 것이 아니기 때문에 단순히 프로그램이라는 것을 어떤 분석을 수행하기 위한 일련의 과정을 컴퓨터가 이해할 수 있도록 하는 것으로 한정하도록 합니다.

1.3 조건문과 논리연산자

if 그리고 else 우리는 어떤 프로세스를 설계할 때 이런 경우에는 A라는 것을 수행하고, 저런 경우에는 B라는 것을 수행한다라고 하는 로직을 사용합니다. R은 이러한 논리를 표현하기 위해서 if와 else 라는 "만약 그렇다면" 이것을 수행하고 "그렇지 않다면" 이것을 수행한다라는 조건문을 제공하고 있습니다. 이것에 대한 문법적 형식은 아래와 같습니다.

```
if(condition){
        expressions
}
else {
        expressions
}
```

이러한 조건문을 이해하기 위해서 위에서 사용한 plus()라는 사용자 정의 함수를 아래와 같이 확장해 보도록 합니다.

- 만약 x1에 입력받은 숫자가 10 보다 크다면 "x1 is greater than 10" (x1은 10보다 큽니다)라는 메시지를 함수의 결과값을 보여주기 전에 출력하고,
- 그렇지 않다면 "x1 is less than or equal to 10" (x1은 10보다 작거나 같습니다) 라는 메시지를 함수의 결과값을 보여주기 전에 출력하도록 합니다.

이를 표현하자면, 아래와 같습니다. f

```
plus <- function(x1, x2){
        if(x1 <= 10) print("x1 is less than or equal to 10")
        if(x1 > 10) print("x1 is greater than 10")
        y <- x1 + x2
        return(y)
}
```

조금 바꿔봅니다.

```
plus <- function(x1, x2){
        if(x1 <= 10) print("x1 is less than or equal to 10")
        else print("x1 is greater than 10")
        y <- x1 + x2
        return(y)
}
```

그 결과는 아래와 같습니다.

```
> plus(3,5)
[1] "x1 is less than or equal to 10"
[1] 8
> plus(15, 4)
[1] "x1 is greater than 10"
[1] 19
>
```

프로그램이 잘 수행됩니다.

논리연산 이제 다시 위의 문법을 살펴보도록 합니다. 조건문 if 의 사용에 꼭 필요한 것은 조건문에 대한 판단입니다. 문법적 요소인 condition 이라는 것은 어떤 주어진 조건에 대한 참과 거짓을 판단하는 부분입니다.

위의 plus 함수에서 이 condition 에 해당하는 부분이 바로 $x1 < 10$ 이라는 표현입니다. 그럼, 이 조건문은 어떻게 수행되는지 알기 위해서 $x1 < 10$의 결과를 출력하도록 print($x1 < 10$)으로 변경합니다.

```
plus <- function(x1, x2){
        if(print(x1 <= 10)) print("x1 is less than or equal to 10")
        else print("x1 is greater than 10")
        y <- x1 + x2
        return(y)
}
```

이를 확인해 보기 위해서 실행해 봅니다.

```
> plus(3,5)
[1] TRUE
[1] "x1 is less than or equal to 10"
[1] 8
```

아하! R은 이렇게 condition 부분에서 참과 거짓을 판단한 후 그 결과로서 돌려주는 TRUE 또는 FALSE 라는 값에 의해서 조건문을 실행하게 된다는 것을 알 수 있습니다. 이러한 조건에 대한 수행적 결과를 우리는 논리적 연산을 수행한다고 합니다.

아래의 예를 살펴봅니다. 먼저 1부터 10사이에 1단위로의 수열을 생성합니다.

```
> x <- 1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

이제 벡터 x의 구성요소중 5 보다 큰 요소들에 대한 결정을 내리도록 하겠습니다.

```
> x > 5
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

그럼, $x > 5$와 $x < 7$ 이라는 두개의 조건을 만족하는 결과는 어떻게 표현할까요? 두개의 조건을 서로 결합하는 것은 & (엠퍼센트) 라는 기호를 이용하여 표기합니다.

```
> x > 5 & x < 7
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
```

위와 같은 표현이 가능하다면 $x < 3$ 또는 $x > 5$ 라는 논리는 어떻게 표현될까요? 이는 | 를 이용하여 표시합니다.

```
> x < 3 | x > 5
 [1]  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

논리연산에는 다음과 같은 것들이 있습니다

`<, >, <=, >=, &, &&, |, ||,`

이 논리연산은 추후에 설명하게 될 수학의 집합연산과도 관계가 깊습니다. 이에 대한 설명은 "수학/확률/수치연산" 이라는 챕터에서 다루도록 하고, 여기에서는 단순한 논리연산만을 다루겠습니다.

plus() 예제 확장: 조건을 하나만 사용하는 것이 아니라 여러개를 사용할 수 있습니다. 예를들어, 우리가 수집하는 두 개의 데이터가 x 와 y 에 저장을 하는데, 실제적으로는 0 보다 큰 값들만이 존재합니다. 따라서, plus() 함수를 사용할때,

- 두개의 입력인자가 모두 양수인 경우에만 plus()연산을 수행하고,
- 만약 그렇지 않다면 "Both x1 and x2 must be positive" 라는 메시지를 출력하는 것입니다.

아래와 같이 프로그램을 살짝 변경합니다.

```
plus <- function(x1, x2){
        if( (x1 > 0) & (x2 > 0) ){
                y <- x1 + x2
                return(y)
        }
        else print("Both x1 and x2 must be positive")
}
```

이제 그 결과를 확인해 봅니다.

```
> plus(3,2)
[1] 5
> plus(-3,2)
[1] "Both x1 and x2 must be positive"
> plus(3,-2)
[1] "Both x1 and x2 must be positive"
```

사용자가 예상한대로 프로그램이 잘 짜지고 있음을 확인할 수 있습니다.

이제 한 단계 더 업그레이드 합니다. 실제로 더하기라는 개념은 두 개의 값을 필요로 합니다. 그래서 plus()라는 함수를 사용할때 두 개의 인자들이 입력되어야 합니다. 따라서, 아래와 같이 해봅니다.

- 만약, 입력된 인자의 개수가 2개 아니라면 "Only two arguments are needed for this computation" (이 연산을 수행하기 위해서는 두개의 인자가 필요합니다) 라는 메시지를 출력하고,
- 입력된 인자의 개수가 2개라면 연산을 수행하되, 오로지 두개의 인자값들이 양수일때만 연산을 수행합니다.

이를 수행하기 위해서는 여러분들은 missing()이라는 함수와 stop()이라는 함수를 알아야 합니다. missing()이라는 함수는 사용된 변수의 값이 있는지 없는지를 판단하며, stop()이라는 함수는 메시지를 보여준뒤 그 뒤에 있는 프로세스들은 모두 중단한 채로 함수를 빠져나가는 것입니다.

```
plus <- function(x1, x2){
        if(missing(x1) | missing(x2)) stop("Two arguments are needed")

        if( (x1 > 0) & (x2 > 0) ){
                y <- x1 + x2
                return(y)
        }
        else print("Both x1 and x2 must be positive")
}
```

이제 확인을 해 봅니다.

```
> plus(3)
Error in plus(3) : Two arguments are needed
> plus(-3,2)
[1] "Both x1 and x2 must be positive"
> plus(3,5)
[1] 8
```

모두 사용자의 로직대로 프로그램이 작성되었음을 확인할 수 있습니다.

그런데 가끔 이런 경우가 있습니다. 조건문이 어떤 특정한 값을 받고 이에 해당하는 경우만을 실행하고자 할때입니다. 예를들면, 아래와 같습니다.

- 만약, 사용자가 "A"를 입력하면 "Hi! What's up?" 을 출력합니다.
- 만약, 사용자가 "B"를 입력하면 "Hello World! I am R" 을 출력합니다.

이 경우에 어떤 독자는 if문을 배웠으니 아래와 같이 할 것입니다.

```
urInput <- function(x){
        if(x=="A") print("Hi! What's up?")
        if(x=="B") print("Hello~ World! I am R")
}
```

아래와 같이 결과를 확인해 봅니다.

```
> urInput("A")
[1] "Hi! What's up?"
> urInput("B")
[1] "Hello~ World! I am R"
> urInput("C")
> urInput(3)
```

여기에서 "C" 또는 3 을 입력 했을 때는 아무것도 수행하지 않았음을 알 수 있습니다. 이와 동일한 기능을 R은 switch()라는 함수를 통하여 수행할 수도 있습니다.

```
urInput2 <- function(x){
        switch(x,
                "A" = print("Hi! What's up?"),
                "B" = print("Hello~ World! I am R")
        )
}
```

이제 동일한 결과를 얻는지 확인해 봅니다.

```
> urInput2("A")
[1] "Hi! What's up?"
> urInput2("B")
[1] "Hello~ World! I am R"
> urInput2("C")
>
```

이러한 조건적 실행을 수행하는 분기문에 대해서는 프로그래밍의 경험이 쌓여 가면서 자연스레 더욱 많은 것을 알게 됩니다. 본 섹션에서는 분기문에 대해서 이정도로 마무리하고 다음 장으로 넘어가겠습니다.

1.4 반복문

이전 섹션에서는 간단하게 2개의 인자들을 입력받아 더하기를 수행하는 함수 plus()를 작성하였습니다. 이번 섹션에서는 반복문이라는 개념을 이해하기 위하여 임의의 벡터 x를 입력받아 그 객체의 구성요소들의 합을 구하는 mySum()이라는 함수를 만들어 보도록 합니다.

먼저 1부터 10까지의 정수로 이루어진 수열을 생성해 봅니다.

```
> x <- 1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
>
```

R은 객체 x의 구성요소들을 더하고자 할때 sum()이라는 함수를 이용합니다.

```
> sum(x)
[1] 55
```

실제 연산은 아래와 같이 표현할 수 있을 것입니다.

```
> x[1] + x[2] + x[3] + x[4] + x[5] + x[6] + x[7] + x[8] + x[9] + x[10]
[1] 55
```

for() 이용하기 만약, 벡터의 구성요소가 100개라면 이렇게 모든 구성요소를 일일이 작성하여 합을 구하는것은 비효율적일 것입니다. 이런 경우 for()라는 반복문을 이용합니다.

합계를 구하는데 있어서 반복문이 사용되는 근본 원리는 벡터의 첫번째 구성요소부터 마지막 구성요소까지의 각각의 구성요소가 가지는 값을 순차적으로 얻을 수 있는가 입니다. 따라서, 몇번째 구성요소를 표시하는 지시자가 필요하며, 이 지시자에 해당하는 값을 불러오는 과정이 필요합니다. 마지막으로 지시자에 해당하는 값을 불러왔을 때, 이 값은 이전의 합계에 더해짐으로서 현재의 합계를 얻게 되는 것입니다. 이를 R 로 표현하면 아래와 같습니다.

```
> x <- 1:10
> val <- numeric(1)    # 합계를 저장할 변수를 미리 생성합니다.
> for(idx in x){       # idx는 x의 % 몇번째 구성요소를 나타내는 지시자입니다
+ print(x[idx])        # % idx번째에 해당하는 x의 값을 출력합니다
+ val <- val + x[idx]  # 합계를 위해서 생성해 놓은 변수에 idx 번째에 해당하는 x의 값을 더함으로서 합계의
+ }

% 첫 번째 줄에서 numeric()의 역할

[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
> val                  # 총계를 확인합니다.
[1] 55
>
```

이렇게 합계를 구하는 기능을 함수로 만드는 것은 어렵지 않습니다.

```
> mySum <- function(x){
+ val <- numeric(1)
+ for(idx in x){
```

```
+ val <- val + x[idx]
+ }
+ return(val)
+ }
> x <- 1:10
> mySum(x)
[1] 55
```

while() 사용하기 R은 for()라는 반복문과 동일한 기능을 가지는 while()이라는 다른 형식의 반복문을 제공합니다. for()라는 반복문이 지시자를 이용하여 중괄호 안의 표현식들을 반복하게 되지만, while()은 조건문을 이용하여 중괄호 안의 표현식을 반복하게 됩니다. for()에 대한 이해를 돕기 위해 사용되었던 1부터 10사이의 정수의 합계를 구하는 예제는 while()문을 이용하여 아래와 같이 할 수 있습니다.

```
> x <- 1:10
> idx <- 1              # 벡터의 첫번째 구성요소를 지정하기 위한 초기값 지정
> while(idx <= 10){     # 만약, 지시자의 값이 10보다 작다면 중괄호의 표현식을 반복
+ print(x[idx])         # 지시자에 해당하는 x의 값을 출력함
+ idx <- idx + 1        # 다음 구성요소를 지시하기 위해서 현재 지시자의 값을 하나 업데이트 함.
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

이렇게 얻어진 각각의 구성요소들의 값에 대한 총계를 구하고자 한다면 아래와 같이 할 수 있습니다.

```
> x <- 1:10
> idx <- 1
> val <- 0
> while(idx <= length(x)){
+ val <- val + x[idx]
+ print(val)
+ idx <- idx + 1
+ }
[1] 1
[1] 3
```

```
[1] 6
[1] 10
[1] 15
[1] 21
[1] 28
[1] 36
[1] 45
[1] 55
```

이를 함수로 만든다면 아래와 같이 할 수 있습니다.

```
> mySum2 <- function(x){
+ idx <- 1
+ val <- 0
+ while(idx <= length(x)){
+ val <- val + x[idx]
+ idx <- idx + 1
+ }
+ return(val)
+ }
> x <- 1:10
> mySum2(x)
[1] 55
```

next와 break 활용 위에서 사용한 mySum()이라는 함수를 조금 더 활용해 보도록 합니다.

만약, 사용자가 1부터 10 사이의 정수들의 총계가 아닌, 1부터 7까지만의 합계를 알고 싶다고 가정합니다. 이런 경우에 x의 구성요소의 개수는 10개이지만, 첫번째부터 7번째 구성요소까지만 더하기를 수행하고 그 반복을 중단하면 될 것입니다. 이러한 기능을 위하여 R은 반복문을 사용할 때 break 라는 기능을 제공하고 있습니다. 이는 분기문이 실행될 때 해당 반복문을 완전 중단하고자 하는 경우에 사용됩니다. 아래의 프로그램을 살펴보시길 바랍니다.

```
> x <- 1:10
> for(idx in x){
+ if(x[idx] == 8) break  # 만약, 지시자가 8번째라면 반복문을 중단함.
+ print(x[idx])
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
```

```
[1] 7
>
```

이제 x가 가지는 값들 중에서 홀수만을 골라서 합을 구하는 프로그램을 짜봅니다. 먼저, 홀수만 더해진다면 아래와 같은 결과값을 가져야 할 것입니다.

```
> x[1] + x[3] + x[5] + x[7] + x[9]
[1] 25
```

이를 해결하는 방법은 x 라는 구성요소를 for()라는 반복문을 이용하여 순차적으로 그 값을 얻어 합계를 내고자 할때, 그 값이 짝수라면 더하는 과정을 실행하지 않고 홀수라면 더하는 과정을 실행하면 될 것입니다. R은 이러한 컨트롤을 위하여 next 라는 예약어를 제공하고 있습니다. 즉, 어떤 조건에 대하여 분기를 실행할 때 next가 이용된다면 현재의 반복과정이 실행되지 않고 바로 다음의 반복과정으로 넘어가게 됩니다. 따라서, 아래와 같이 프로그램을 작성할 수 있습니다.

```
> x <- 1:10
> val <- numeric(1)
> for(idx in x){
+ if(x[idx] %% 2 == 0) next
+ val <- val + x[idx]
+ }
> val
[1] 25
```

참고: '%%'는 나머지를 구하는 연산자임을 기억하시기 바랍니다.

벡터라이제이션: 실제로 이렇게 1부터 10사이의 홀수들의 합계를 구하는 것은 반복문을 사용하지 않고 1줄로 표현할 수 있습니다. 이렇게 벡터라이제이션의 기능을 활용하여 주어진 연산을 수행하는 것은 프로그램을 짧고 그 가독성을 높일 수 있습니다. 가독성이란 프로그램이 얼마나 읽기가 쉬운가를 말하는 것입니다.

```
> sum(x[x %% 2 != 0])
[1] 25
```

그러나, 이렇게 벡터라이제이션을 하는 것은 프로그래밍에 대한 경험이 쌓여야 합니다. 여기에서는 기초 프로그래밍을 위한 개념을 이해하기 위하여 for()과 while()을 설명했으나, 이 이후로의 모든 내용은 for()와 while()을 이용한 반복문은 가급적이면 피할 것입니다. 바로 다음에 다루게 될 데이터 조작 및 수치해석에 관련된 챕터들에서는 R의 특징을 최대한 살린 벡터라이제이션 중심의 연산방식으로 표현함으로서 독자에게 도움이 되고자 하였습니다.

반복문에서 repeat 라는 기능이 있습니다. (이것에 대한 설명은 문서를 읽던 독자의 요청이 있으면 기재하도록 합니다).

1.5 프로그램의 가독성

많은 초보 프로그래머들은 R 프로그램을 작성할 때 반드시 따라야 할 코딩 스타일을 따르고자 합니다. 코딩스타일을 이야기 하는 것은 프로그램의 가독성 때문입니다. 가독성이란 내가 작성한 프로그램을 다른 사람이 쉽게 이해할 수 있으며, 다른 사람이 작성한 프로그램을 내가 얼마나 이해할 수 있는가를 의미합니다. 이러한 가독성 때문에 프로그램을 작성할 때 어떤 작성 표준에 맞추어 짜야 한다는 것입니다. 그리고, 이러한 가독성은 결국 공동작업을 통한 협업을 할 때 그 힘을 발휘하게 됩니다.

사실상 그러한 코딩스타일에 대한 표준은 사실상 존재하지 않지만, 가독성과 공동작업을 위하여 일종의 코딩컨벤션 (coding convention)을 사용합니다.

- GNU C 프로그래밍 코딩 스탠다드 (`http://www.gnu.org/prep/standards/standards.html`)이 도움이 될 것입니다.
- 구글에 포스팅되어 있는 구글의 R 스타일 가이드 (`http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html`)이 도움이 될 것입니다.
- 그리고 Github에 게시된 Hadley의 스타일 가이드 (`https://github.com/hadley/devtools/wiki/Style`) 등이 여러분들의 R 프로그래밍 스타일을 갖추는데 도움이 될 것입니다.

프로그램의 가독성을 높이는 또 하나의 방법은 주석문을 많이 활용하는 것입니다. 예를들어, 위에서 우리가 작업한 plus() 함수는 만약 다른 사람이 보면 잘 이해하지 못할 것입니다. 따라서 아래와 같이 저장해 두는 것이 좋습니다.

```
#
# Authors: (프로그램 작성자 이름)
# Created on (작성날짜)
# Modified on (최종수정일)
#
# Description: (간단한 설명)
# 이 프로그램은 나의 첫번째 R 프로그램인 plus() 함수를 작성한 것입니다.
#

# 두 개의 입력인자를 가진는 plus()함수의 정의

plus <- function(x1, x2){

        # 입력인자가 2개임을 확인합니다.
        if(missing(x1) | missing(x2)) stop("Two arguments are needed")

        # 입력인자의 값은 반드시 양수이어야 합니다.
        if( (x1 > 0) & (x2 > 0) ){
                y <- x1 + x2
                return(y)
        }
```

```
        else print("Both x1 and x2 must be positive")
}
```

프로그래밍을 도와주는 여러가지 유틸리티들: 본 섹션에서는 간단한 예제를 통하여 R에서 제공하는 프로그래밍 요소인 조건문, 반복문, 그리고 함수에 대해서 알아보았습니다. 문서의 처음에 언급했던바와 같이 분석은 일련의 프로세스를 수행하는 것이고, 이러한 수행을 자동화 시켜주는 것에 있어서 프로그래밍은 매우 효과적인 도구입니다. 따라서, 어떤 일을 수행하도록 하는 명령어들이 복잡해지고 그 양이 많아진다면 외부파일에 스크립트를 작성후 배치처리를 하는 것이 좋습니다. 또한, 자신의 시스템 사용환경에 익숙하다면 프로그램을 작성하는데 더욱 도움이 됩니다. 이러한 내용은 이 문서의 "환경설정과 유틸리티" 라는 챕터에 기록해 두었습니다.

본 챕터는 사용에 익숙해지기 위한 아주 기초적인 내용만을 다루기 때문에 아래와 같은 내용은 "패키지 작성" 이라는 챕터에서 다루도록 하겠습니다.

- 렉시컬 스코핑
- environemnt
- 제네릭 함수
- 메소드와 클래스
- S3와 S4 프로그래밍 그리고 R5
- 에러 핸들링 - try(), tryCatch()
- 운영체제와의 소통 - 파일과 디렉토리 관리에 필요한 유틸리티들

TODO:

- 수치해석 부분에서 연산의 정확도와 라운딩에러에 대해서 설명해 주는게 좋을 것 같음 .
- 지금 현재 이 섹션에서 ifelse() 사용이 완전히 빠져 있음
- all, any, identical 과 같은 함수를 알려주어야 함.