3. R프로그래밍

3.2 흐름제어(조건문과 반복문)

이 절에서는 프로그래밍의 필수 요소가 되는 조건문과 반복문(if, for, while, repeat)에 대해서 알아본다 . 문법은 대부분의 언어와 유사하다.

3.2.1 if

먼저 다수의 TRUE, FALSE 데이터를 한 번에 처리한다면 ifelse() 함수를 고려해라.

ifelse : 주어진 test 값에 따라 yes or no 값을 반환한다.

```
ifelse(
    test, #참, 거짓을 저장한 객체
    yes, #test가 참일 때 선택할 값
    no #text가 거짓일 때 선택할 값
)

#test에 다수의 TRUE, FALSE가 저장되어 있을 때 TRUE에 대해서는 yes,
#                                              FALSE에 대해서는 no를 반환한다.
```

```
> if(TRUE){
+     print("TRUE")
+     print("hello")
+ }else {
+     print("FALSE")
+     print("world")
+
+ }
[1] "TRUE"
[1] "hello"
```

iselse()를 사용하면 if 문을 다수의 데이터에 한 번에 적용하는 연산이 가능하다. 다음은 1,2,3,4,5 에서 값이 짝수일 경우 "even", 홀수일 경우 "odd"를 출력하는 예이다.

```
> x <- c(1,2,3,4,5)
> ifelse(x %% 2== 0, "even", "odd")
[1] "odd"  "even" "odd"  "even" "odd"
```

3.2.2 반복문

R의 반복문에는 for, while, repeat문이 있다.

문법	의미

for(i in data){
i를 사용한 문장
}
	data에 들어 있는 각각의 값을 변수 i에 할당하면서 각각에 대해 블록 안의 문장을 수행한다.

while(cond){
조건이 참일 때 수행할 문장
}
	조건이 cond이 참일 때 블록안의 문장을 수행한다.

repeat{
반복해서 수행할 문장
}
	블록 안의 문장을 반복해서 수행한다. repeat은 다른 언어와 do-while에 해당한다.

반복문 내 블록에서는 break, next 문을 사용해 반복의 수행을 조정할 수 있다.

- break : 반복문을 종료한다.
- next : 현재 수행 중인 반복문 블록의 수행을 중단하고 다음 반복을 시작한다.

다음 코드에서 for 문은 변수 i가 주어진 벡터에 있는 $1, 2, 3, \cdots, 10$을 차례로 출력한다.

```
> for(i in 1:10){
+     print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

다음 while문은 1부터 10까지의 숫자를 출력하기 위해 변수 i의 값을 10과 비교하면서 print() 하는 예다.

```
> i<- 1
> while(i<=10){
+     print(i)
+     i=i+1
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

while문에서 next를 사용해 짝수만 출력해보자. 값이 짝수인지 확인하는 데는 나머지 연산자 %%를 사용했다.

```
> i<- 0
> while(i<=9){
+     i<-i+1
+     if(i%%2!=0){
+         next #print()를 실행하지 않고
+              #while문의 처음으로 돌아감
+     }
+     print(i)
+ }
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
```

다음은 repeat를 사용해서 1부터 10까지 값을 출력한 예다.

```
> i<-1
> repeat{
+     print(i)
+     if(i>=10){
+         break
+     }
+     i<-i+1
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

3.3 연산

이 절에서는 연산자와 벡터 연산, NA가 포함된 데이터에서의 연산에 대해 알아본다. 연산자에서 특히 벡터 연산과 NA가 포함된 데이터를 다루는 방법은 다른 언어와 차이가 있는 부분이니, 유의해서 보기 바란다.

3.3.1 수치 연산

지금까지 암시적으로 사용한 사칙 연산(+,-,*,/)을 포함한 수치 연산에 대해 알아보자. 사칙 연산은 다른 언어와 크게 다르지 않지만 약간의 문법적 차이가 있다.

다른 언어와의 차이점

연산자와 함수	의미
n%/%m	n을 m으로 나눈 몫
log(x, base=exp(1))	$log_{base}(x)$, 만약 base가 지정되지 않으면 $log_e(x)$ 를 계산
sin(x), cos(x), tan(x)	삼각함수

R은 다음과 같은 벡터 연산도 가능하다.

```
> 1:5 * 2+1
[1]  3  5  7  9 11
```

연산에 있어 또 다른 특별한 객체는 행렬이다.

3.3.2 벡터 연산

```
> x<-c(1,2,3,4,5)
> x +x
[1]  2  4  6  8 10
> x == x
[1] TRUE TRUE TRUE TRUE TRUE
> x ==c(1,2,3,4,55)
[1]  TRUE  TRUE  TRUE  TRUE FALSE
> c(T,T,F) & c(F,T,T)
[1] FALSE  TRUE FALSE
```

```
x<-c(1,2,3,4,5)
> ifelse(x%%2 == 0 , "even", "odd")
[1] "odd"  "even" "odd"  "even" "odd"
```

벡터 연산을 사용하면 데이터 프레임에 저장된 데이터 중 원하는 정보를 쉽게 얻을 수 있다. 기본 원리는 데이터 프레임에 진리값을 지정해 특정 행을 얻어올 수 있다는 점을 이용한 것이다. 다음은 1행, 3행, 5행에 TRUE를 지정해 해당 행들만 데이터 프레임에서 가져오는 예이다.

```
> (d<-data.frame(x=c(1,2,3,4,5),
+                y=c("a","b","c","d","e")))
  x y
1 1 a
2 2 b
3 3 c
4 4 d
5 5 e
> d[c(TRUE,FALSE,TRUE,FALSE,TRUE)]
Error in `[.data.frame`(d, c(TRUE, FALSE, TRUE, FALSE, TRUE)) :
  undefined columns selected
> d[c(TRUE,FALSE,TRUE,FALSE,TRUE),]
  x y
1 1 a
3 3 c
5 5 e
```

따라서 행의 선택 기준이 되는 TRUE, FALSE를 벡터 연산으로 만들어 주면 특정 행을 선택할 수 있다.

다음은 x값이 짝수인 행만 선택한 예다.

```
> d[d$x %% 2 == 0,]
  x y
2 2 b
4 4 d
```

3.3.3 NA

데이터에 NA가 포함되어 있을 경우 연산 결과가 다음과 같이 NA로 바뀌어 버리므로 주의가 필요하다.

```
> NA & TRUE
[1] NA
> NA + 1
[1] NA
```

이러한 문제점을 해결하기 위해 많은 R함수에서 na.rm을 함수 인자로 받는다. na.rm은 NA 값이 있을 때 해당 값을 연산해서 제외할 것인지를 지정하는 데 사용한다. 다음 예를 보자.

```
 sum(c(1,2,3,NA))
[1] NA
> sum(c(1,2,3,NA), na.rm = TRUE)
[1] 6
```

이처럼 NA 값에 따라 처리를 다르게 하려면 na.fail, na.omit, na.exclude, na.pass 함수를 사용한다. 또 다른예로, 잘 알려진 기계 학습 패키지 중 하나인 caret(Classification and Regression Training)은 NA 처리 방법을 결정한다.

함수	의미
na.fail(object, ...)	object에 NA가 포함되어 있으면 실패한다.
na.omit(object, ...)	object에 NA가 포함되어 있으면 이를 제외한다.
na.exclude(object, ...)	object에 NA가 포함되어 있으면 이를 제외한다는 점에서 na.omit과 동일하다. 그러나 naresid, napredict를 사용하는 함수에서 NA로 제외한 행을 결과에 다시 추가한다는 점이 다르다.
na.pass(object, ...)	object에 NA가 포함되어 있더라도 통과시킨다.

다음 예는 이 함수들의 차이를 보여준다.

```
> (x<-data.frame(a=c(1,2,3), b=c("a",NA,"c"),
+                c=c("a", "b",NA)))
  a    b    c
1 1    a    a
2 2 <NA>    b
3 3    c <NA>
> na.fail(x)
Error in na.fail.default(x) : 객체안에 결측값들이 있습니다
> na.omit(x)
```

```
  a b c
1 1 a a
> na.exclude(x)
  a b c
1 1 a a
> na.pass(x)
  a    b    c
1 1    a    a
2 2 <NA>    b
3 3    c <NA>
```

따라서 NA를 어떻게 처리할지를 na.action이라는 함수 인자로 받았다면 'na.action(데이터 프레임)'을 실행해 현재 처리 중인 데이터를 사용자가 원하는 대로 정제할 수 잇다.

na.omit과 na.exclude의 차이

다음과 같이 NA가 포함된 데이터 프레임을 가정해보자.

```
> df<-data.frame(x=1:5, y=seq(2,10,2))
> df[3,2]=NA
> df
  x  y
1 1  2
2 2  4
3 3 NA
4 4  8
5 5 10
```

이 데이터에 y = ax + b 형태의 선형 모델을 가정해보자. 이 모델은 선형 회귀 함수 lm()으로 만들 수 있다. 만들어진 모델에 resid() 함수를 적용하면 선형 모델로 예측한 값과 실제 y값 간의 차이인 잔차 (residual)를 구할 수 있다(선형 회귀 및 lm, resid 함수 등에 대한 내용은 다시 다룬다).

lm() 함수는 인자로 na.action을 받으며, 이 값에 따라 NA가 포함된 행을 다루는 방법이 달라진다. 예를 들어 na.omit을 지정하면 다음과 같이 NA가 포함된 3행을 제외하고 모델을 작성한다. 따라서 df에 총 5개 행이 있지만 resid()의 결과는 총 4개 값이다.

```
> resid(lm(y~x, data = df, na.action = na.omit))
           1            2            4
-7.423409e-16  9.696420e-16  6.043761e-17
           5
-2.877387e-16
```

반면 na.action에 na.exclude를 지정하면 NA를 제외하고 모델을 만들지만, 잔차(residual)를 구할 때 NA가 포함된 행은 잔차를 NA로 해서 추가한다. 따라서 resid()의 걀과의 길이와 원래 데이터의 길이가 같다.

```
> resid(lm(y~x, data = df, na.action = na.exclude))
           1            2            3
-7.423409e-16  9.696420e-16           NA
           4            5
 6.043761e-17 -2.877387e-16
```

3.4 함수의 정의

코드의 반복을 줄이거나 코드의 가독성을 높이려면 함수를 작성해 코드를 추상화해야한다. 또, 함수를 작성하면 유닛 테스트를 사용해 해당 함수의 동작을 검증할 수 있다는 장점이 있다. 이 절에서는 사용자 정의 함수를 작성하는 방법에 대해 알아본다.

3.4.1 기본 정의

```
function_name <- function(인자, 인자, ...)(
    함수 본문
    return(반환 값) #반환 값이 없다면 생략
)
```

예를 들어, 다음은 피보나치 함수를 구현한 예다.

```
> fibo<- function(n){
+     if (n==1 || n==2){
+         return(1)
+     }
+     return(fibo(n-1)+fibo(n-2))
+ }
> fibo(1)
[1] 1
> fibo(5)
[1] 5
```

Warning! R에서 함수를 정의하는 방법은 이처럼 다른 언어의 함수 정의와 유사하지만 몇 가지 차이점이 있다. 첫째는 값 반환시 'return 반환 값' 형태가 아니라 함수 호출 하듯이 'return (반환 값)'형태로 작성해야 한다는 점이다. 둘째는 return()을 생략하면 함수에서 마지막 문장의 반환 값이 함수의 반환 값이 된다는 점이다. 이 점을 이용하면 fibo() 함수를 다음과 같이 고쳐 쓸 수 있다.

```
fibo<- function(n){
    if (n==1 || n==2){
        1
    } else {
    return(fibo(n-1)+fibo(n-2))
    }
}
```

그러나 보통은 return()을 적어주어 코드의 의도를 명확히 한다.

함수를 호출할 때는 인자의 위치를 맞춰서 값을 넘겨주는 방식, 인자의 이름을 지정해서 넘겨주는 방식 두 가지 모두 가능하다. 실습해보자.

```
> f <- function(x,y){
+     print(x)
+     print(y)
+ }
> f(1,2)      #인자의 위치에 맞춘 전달
[1] 1
[1] 2
> f(y=1,x=2) #인자 이름 지정 방식으로 전달
[1] 2
[1] 1
```

3.4.2 가변 길이 인자

R에서 함수들의 도움말을 살펴보면 '...'을 인자 목록에 적은 경우를 종종 볼 수 있다. ...은 개수를 알 수 없는 임의의 인자를 표현하는데 사용하기도 하고, 내부에서 호출하는 다른 함수를 넘겨줄 인자를 표시하는 데도 사용한다.

다음은 가변 인자로 ...을 사용한 예다. 함수 f()에서 ...을 인자로 지정한 뒤 이름 하나씩 화면에 출력했다.

```
> f<-function(...){
+     args <- list(...)
+     for (a in args){
+         print(a)
+     }
+ }
> f('3','4')
[1] "3"
[1] "4"
```

다음은 함수 g()가 인자 z와 ...을 인자로 받아서 인자 z는 자신이 처리하고 나머지 인자들은 함수 f로 넘겨 처리하는 예이다. 코드에서 f()를 호출할 때 마치 명시적인 인자를 넘겨주듯이 ...을 인자로 지정했다.

```
> f <- function(x,y){
+     print(x)
+     print(y)
+ }
> g<- function(z, ...){
+     print(z)
+     f(...)
+ }
> g(1,2,3)
[1] 1
[1] 2
[1] 3
```

3.4.3 중첩 함수

함수 안에 또 다른 함수를 정의하여사용할 수 있다. 이를 중첩 함수라고 부른다. 다음 코드는 함수 f() 안에 g()를 정의하고, 함수 f() 안에서 이를 호출하여 사용하는 예를 보여준다.

```
> f<-function(x,y){
+     print(x)
+     g<-function(y){
+         print(y)
+     }
+     g(y)
+ }
> f(1,2)
[1] 1
[1] 2
```

중첩 함수를 사용하면 함수 안에서 반복되는 동작을 한 함수로 만들고 이를 호출하여 코들ㄹ 간결하게 표현할 수 있다는 장점이 있다. 또한, 내부 함수가 외부 함수에 정의된 변수를 접근할 수 있어 클로저로 사용할 수 있다.

```
f<-function(x1){
    return(function(x2){
        return(x1+x2)
    })
}
g<-f(1)
g(2)
[1] 3     #x1 = 1, x2 = 2
g2<-f(2)
g2(5)
[1] 7    #x1 = 2, x2 = 7
```

3.5 스코프

코드에 기술한 이름(예를 들면, 변수명)이 어디에서 사용 가능한지를 정하는 규칙을 스코프 라고 한다. R에서는 대부분의 현대적인 프로그래밍 언어가 그러하듯, 문법적 스코프(정적 스코프)를 사용하며, 문법적 스코프는 변수가 정의된 블록 내부에서만 변수를 접근할 수 있는 규칙을 말한다.

R의 스코프 규칙을 코드를 통해 알아보자. 예를 들어, 콘솔에서 변수를 선언하면 모든 곳에서 사용 가능한 전역 변수가 된다. 이 변수는 현재 실행 중인 R 세션 동안 유효하다. 따라서 코드를 여러 파일에 나눠놓고 source()를 사용해 실행할 경우 다른 파일에서도 해당 변수를 사용할 수 있다.

다음은 콘솔에서 변수 n을 선언한 다음, 이 변수를 함수 내부에서 사용한 예다.

```
n<-1
f<-function(){
    print(n)
}
f()
n<-2
f()
```

만약 함수 내부에서 전역 변수와 같은 이름의 지역 변수를 사용하면, 함수 내부의 지역 변수가 우선한다.

```
n<-100
f<-function(){
    n<-1
    print(n)
}
f()
```

만약 함수 내부에서도, 전역 변수로도 선언되지 않은 이름을 사용하면 에러다. 이를 살펴보기에 앞서, R 객체를 메모리에서 삭제하거나 객체를 나열하는 함수들에 대해 살펴보자.

메모리상의 객체 관련 함수

rm : 지정한 환경에서 객체를 삭제한다.

```
rm(
    ..., #삭제할 객체의 목록
    list = character(), #삭제할 객체를 나열한 벡터
    envir = as.environment(pos) #객체를 삭제할 환경
)
```

ls : 객체를 나열한다.

```
ls(
    name, #객체를 나열할 환경의 이름
    envir #name 대신 직접 환경을 지정할 경우 사용
)
#반환 값은 객체 이름의 문자열 벡터다.
```

따라서 rm(list = ls())는 메모리에 있는 모든 객체를 삭제하는 명령이 된다. 다음은 rm(list=ls())를 사용해 메모리에 존재하는 객체들을 삭제하여 앞서 예제에서 선언한 n을 없앤 다음, 함수 f()에서 n을 출력해본 예이다.

```
> rm(list=ls())
> f<- function(){
+     print(n)
+ }
> f()
Error in print(n) : object 'n' not found
```

변수는 내부 블록에서만 접근할 수 있으므로 함수 내부에 정의한 이름은 함수 바깥에서 접근할 수 없다.

```
> rm(list = ls())
> f<-function(){
+     n<- 1
+ }
> f()
> n
Error: object 'n' not found
```

함수 내에서 이름은 함수 안의 변수들로부터 먼저 찾는다. 같은 이유로 함수 인자의 변수명 역시 전역 변수보다 우선한다.

```
> n<-100
> f<-function(n){
+     print(n)
+ }
> f(1)
[1] 1
```

중첩 함수에도 같은 규칙이 적용된다. 다음 코드에서 함수 g() 안에는 변수 a가 선언되어 있지 않다. 따라서 R언어는 g() 함수를 감싼 f() 로부터 변수 a를 찾는다.

```
> f<-function(x){
+     a<-2
+     g<-function(y){
+         print(y+a)
+     }
+     g(x)
+ }
> f(1)
[1] 3
```

그러나 만약 함수 f()마저도 변수 a를 포함하고 있지 않다면 전역 변수 a를 사용한다.

```
> a<-100
> f<-function(x){
+     g<-function(y){
+         print(y+a)
+     }
+     g(x)
+ }
> f(1)
[1] 101
```

내부 블록에서 외부 블록에 선언된 값을 수정하고자 할 때는 주의가 필요하다. 내부 블록이 외부 블록보다 우선하므로 <-를 사용한 값의 할당 시, 값이 할당되는 대상이 내부 블록으로 간주되기 때문이다. 예를 들어, 함수 f() 안에 변수 a가 있고 중첩된 함수 g()에서 함수 f() 안의 변수 a에 값을 할당하려고 하는 다음 코드를 살펴보자.

```
> f<-function(){
+     a<-1
+     g<-function(){
+         a<-2
+         print(a)
+     }
+     g()
+     print(a)
+ }
> f()
[1] 2
[1] 1
```

이 코드에서는 함수 g()에서 함수 f()에 선언된 변수 a에 2를 할당하려 했지만 <- 는 함수 g() 내부에 새로운 변수 a를 만들고 해당 변수에 2를 할당한다. 따라서 f()에서 print(a)가 수행될 때 출력은 2가 아닌 1이 된다. 만약 함수 g에서 함수f() 안의 변수 또는 전역 변수에 값을 할당하려면 <<- 를 사용해야 한다.

```
b<-0
> f<-function(){
+     a<- 1
+     g<-function(){
+         a<<-2
+         b<<-2
+         print(a)
+         print(b)
+     }
+     g()
+     print(a)
+     print(b)
+ }
> f()
[1] 2
[1] 2
[1] 2
[1] 2
```

실행 결과 g() 안에서 <<-를 사용한 값의 할당은 함수 f에 선언된 a와 전역 변수인 b를 대상으로 이뤄졌음을 볼 수 있다.

3.6 값에 의한 전달

R에서는 모든 것이 객체다. 또, 객체는 함수 호출 시 일반적으로 값으로 전달된다. 이를 변수에 대한 참조에 의한 방식에 대비해 값에 의한 전달 이라고 한다. 값으로 전달된다는 말은 객체가 복사되어 함수로 전달된다는 의미다.

여기에는 예외(환경, 심볼, 스페셜, 빌트인)등도 있다. 그러나 이들에 대한 내용은 다루지 않는다.

값으로 데이터가 넘어가는 예를 살펴보자. 다음과 같이 데이터 프레임을 함수에 인자로 주었을 때 함수 내부에서 수행한 변경은 원래 객체에 반영되지 않는다. 왜냐면 함수 f() 안에서 df2 란 f()를 호출한 쪽에서 넘긴 객체를 가리키는 참조가 아닌 넘겨받은 df를 복사한 새로운 데이터 프레임이기 때문이다.

```
> f<-function(df2){
+     df2$a<-c(1,2,3)
+ }
> df<-data.frame(a=c(4,5,6))
> f(df)
> df
  a
1 4
2 5
3 6
```

만약 인자로 받은 df를 수정하고 이를 함수를 호출한 쪽에 반영하려면 함수 f에서 수정된 값을 반환하고 함수를 호출한 쪽에서 원래 변수에 할당해야 한다.

```
> f<-function(df){
+     df$a=c(1,2,3)
+     return(df)
+ }
> df<-data.frame(a=c(4,5,6))
> df1<-f(df)
> df1
  a
1 1
2 2
3 3
```

결과적으로 특별한 객체(예를 들면, 네트워크 접속이나 파일 입출력 등)를 제외하고는 객체의 상태가 함수에 의해 직접 수정되지 않는다. 이런 이유로 어떤 함수를 호출하더라도 인자로 넘긴 객체가 수정되지 않음을 보장받는다.

3.7 객체의 불변성

R의 객체는 (거의 대부분의 경우에) 불변이다. 객체 지향 프로그래밍에 친숙하지 않은 독자라면, 객체라는 용어는 R에서 메모리에 할당된 데이터 구조들을 뜻한다고 생각하면 된다. 예를 들어 벡터를 하나 메모리에 만들었으면 그것이 객체고, 리스트를 하나 만들었어도 객체다.

프로그래밍에서 값이 불변이란 말은 수정하는 것이 불가능하다는 뜻이다. 따라서 a라는 리스트 객체를 수정하는 것처럼 보이는 다음 코드는 실제로는 a객체를 변경하는 것이 아니다.

```
a<-list()
a$b<-c(1,2,3)
```

코드를 보면 리스트 a에 새로 필드 b를 만들고 거기에 c(1,2,3)을 할당하는 수정처럼 보인다. 그러나 실제로 일어나는 일은 a를 복사한 새로운 객체 a`을 만들고, 이 a` 에 필드 b를 추가하고 해당 필드에 c(1,2,3)을 채워넣은 다음, 변수명 a가 a`을 가리키도록 하는 것이다.

최초의 변수 a는 어떤 객체에 붙여진 이름이었다. 그러나 그 객체의 값을 바꾸면 이전 객체는 버려지며 새로이 a` = c(1,2,3)을 담고 있는 새로운 객체가 만들어지고, a는 이 새로운 객체의 이름이 된다.

이를 직접 코드로 확인해보자. 메모리를 추적하기 위해 먼저 다음 두 가지 함수를 알아둘 필요가 있다.

객체의 복사 추적 관련 함수 실습

다음은 tracemem()을 사용하여 리스트 a를 추적하게 한 다음, 리스트 a의 값을 수정하자 a가 복사됨을 보여준다.

```
> a<-list()
> tracemem(a)
[1] "<000002748C2C2F30>"
> a$b<-c(1,2,3)
tracemem[0x000002748c2c2f30 -> 0x000002748c476ea0]:
> untracemem(a)
```

'2,4 벡터' 절에서 벡터 연산을 설명하면서 벡터 기반 연산을 사용하는 것이 for 등의 반복문을 사용하는 것보다 효율적이라고 한 바 있다. 그 이유 중 하나가 바로 객체가 불변이라는 점이다. 예를 들어 for 문 안에서 벡터의 인자를 하나씩 바꾸는 다음 코드는 v[i]값을 1씩 증가시킬 때마다 i번째 값이 수정된 벡터를 매번 새로 만들어 v에 할당한다. 따라서 새로운 객체 1,000개를 생성하는 비효율이 발생한다.

```
> v<-1:1000
> for(i in 1:1000){
+     v[i]<-v[i]+1 #i번째 값을 바꿀 때마다 새로운 벡터가 생성된다!!
+ }
```

반면 같은 일을 하는 다음 코드는 v 안에 있는 전체 값을 1만큼 증가시킨 객체를 한 개 만든 다음, 이를 v에 할당한다.

```
v<-1:1000
v<-v+1
```

이처럼 벡터 연산이 더 바른 이유는 이러한 메모리 사용 최적화 문제와도 관련이 있다. 메모리 문제를 확인하는 한 가지 방법은 다음 코드를 실행해보는 것이다.

```
rm(list=ls())
gc()
v<-1:99999999
for(i in 1:99999999){
    for(j in 1:99999999){
        v[j] <- v[j]+1
    }
}
```

위 코드를 실행하고 for 문이 수행되는 동안 작업 관리자를 살펴보면 메모리 사용량이 서서히 증가함을 알 수 있다.

3.8 모듈 패턴

이 절에서는 모듈 패턴에 대해 살펴본다. 모듈이란 외부에서 접근할 수 없는 데이터와 그 데이터를 제어하기 위한 함수로 구성된 구조물을 말한다. 패턴이란 정형화된 코딩 기법을 뜻한다. 모듈 패턴의 장점은 다음과 같다.

첫째, 데이터를 외부에서 직접 접근할 수 없게 되어 내부 구현이 숨겨진다. 다시 말해, 모듈의 사용자가 모듈 내부에서 데이터가 어떻게 저장되는지 신경 쓸 필요가 없다는 의미다.

둘째, 사전에 정의된 함수로만 데이터를 다룰 수 있게 되어 데이터의 내부 구조를 잘 모르는 사용자가 데이터를 잘못 건드려서 손상시키는 일을 막을 수 있다.

마지막으로, 모듈의 사용자는 내부 구조는 건드릴 수 없고 외부로 노출된 함수만 불러 쓰고 있는 상태가 되므로, 모듈의 제작자는 해당함수가 이전과 같은 결과를 내놓기만한다면 함수의 내부 구조나 데이터 구조를 마음대로 바꿀 수 있다.

이 절에서는 잘 알려진 자료 구조인 큐를 모듈로 작성해본다.

3.8.1 큐

큐는 먼저 들어온 데이터를 먼저 처리(FIFO, (First In First Out))하는데 사용하는 자료 구조다. 큐는 사람들이 차례로 줄을 서 있고, 줄의 제일 앞에 서 있는 사람부터 자기 일을 처리하는 모습을 연상하면 된다. 새로 도착한 사람은 줄의 맨 뒤에 서서 기다린다.

큐는 다음 세 가지 함수로 구현한다.

- Enqueue : 줄의 맨 뒤에 데이터를 추가한다.
- Dequeue : 줄의 맨 앞에 있는 데이터를 가져온다. 가져온 데이터는 줄에서 빠진다.
- Size : 줄의 길이, 즉 자료 구조 내에 저장된 데이터의 수를 반환한다.

이 세가지 함수를 지원하는 큐를 작성해보자.

```
> q<-c()
> q_size<-0
> enqueue<-function(data){
+     q<<-c(q,data)
+     q_size<<-q_size+1
+ }
> dequeue<-function(){
+     first <- q[1]
+     q<<-q[-1]
+     q_size<<-q_size-1
+     return(first)
+ }
> size<-function(){
+     return(q_size)
+ }
```

위 코드에서 큐에 저장될 데이터는 벡터 q를 사용해 저장하고 있다. q_size는 큐에 저장된 데이터 수를 기록하는 목적으로 사용한다. 함수 enqueue()는 q에 이미 저장되어 있는 데이터에 인자로 받은 데이터를 추가하여 다시 변수 q에 할당한다. 이때 <<-를 사용해 전역 변수에 있는 q를 직접 접근하게 했다. 마지막으로 q_size의 값을 1 증가시킨다.

함수 dequeue()는 q에 저장된 데이터 중 첫 번째 요소를 first에 저장하고, q에는 이 데이터를 제외한 데이터를 저장한다음 first를 반환한다. 이때 q_size가 1 감소한다.

함수 size()는 q의 길이인 q_size를 반환한다.

위 코드는 다음과 같이 사용할 수 있다.

```
> enqueue(1)
> enqueue(1)  #1이 줄의 끝에 선다.
> enqueue(3)  #3이 줄의 끝에 선다.
> enqueue(5)  #5이 줄의 끝에 선다.
> print(size())
[1] 4
> print(dequeue())
[1] 1
> print(dequeue())
[1] 1
> print(dequeue())
[1] 3
> print(dequeue())
[1] 5
> print(size())
[1] 0
```

3.8.2 큐 모델 작성하기

앞 절에서 작성한 큐 코드의 작성자는 큐의 맨 뒤에 데이터를 저장하고 큐의 맨 앞에서 데이터를 가져오도록 설계했다. 하지만 q라는 변수가 전역으로 선언되어 있으므로 이 함수를 거치지 않고 외부에서 데이터를 조작해버릴 수 있다. 그리고 이때 데이터의 무결성이 다음과 같이 깨질 수 있다.

```
> enqueue(1)
> q<-c(q,5)
> print(size())
[1] 1
> dequeue()
[1] 1
> dequeue()
[1] 5
> size()
[1] -1
```

보다시피 q의 외부에서 q에 직접 값을 할당해버린 탓에 실제로는 q내부에 데이터가 2개 저장되 있지만 size()를 호출했을 때 크기가 1로 나타나는 문제가 있다. 이러한 문제를 막기 위한 방법은 큐 코드 전체의 관련된 변수를 한 함수 안으로 감추는 것이다.

```
> queue<-function(){
+     q<-c()
+     q_size<-0
+
+ enqueue <- function(data){
+     q<<-c(q,data)
+     q.size<<-c(q,data)
+ }
+ dequeue<-function(){
+     first<-q[1]
+     q<<-q[-1]
+     q_size<<-q_size-1
+     return(first)
+ }
+ size<-function(){
```

```
+     return(q_size)
+ }
+
+ return(list(enqueue=enqueue, dequeue=dequeue, size=size))
+ }
```

이 코드가 앞서와 다른 점이라면 q와 q_size가 이제 queue() 함수 안에 있는 지역 변수라는 점이다.

따라서 이 변수들은 외부에서 접근 이 불가하다. 또 함수 enqueue(), dequeue(), size()는 함수 queue()의 반환 값이 되었고 이 값은 리스트로 반환되고 있다.

이렇게 만든 queue()는 다음과 같이 사용될 수 있다.

```
> q<-queue()
> q$enqueue(1)
> q$enqueue(3)
> q$size()
[1] 2
> q$dequeue()
[1] 1
> q$dequeue()
[1] 3
> q$size()
[1] 0
```

queue() 함수 호출 시 만들어지는 queue() 함수 내부의 지역 변수 q와 q_size가 생성되는 공간은 queue() 함수를 호출할 때마다 매번 새로 생성된다. 즉, queue()를 다음과 같이 여러 개 만들어서 사용해도 데이터가 서로 섞이지 않게 된다.

```
> q<-queue()
> r<-queue()
> q$enqueue(1)
> r$size()
[1] 0
> r$enqueue(3)
> q$dequeue()
[1] 1
> r$dequeue()
[1] 3
> q$size()
[1] 0
> r$size()
[1] 0
```

이러한 코딩 기법을 모듈 패턴이라고 한다.