R 프로그래밍

환경생태데이터사이언스 실습 September 17, 2019

R의 특징

R 언어 프로그래밍 특징

1. **벡터** 처리에 최적화
    - 내장 함수: `sum()`, `mean()`, `lapply()`, `sapply()` 등
2. 결측치 (NA) 고려
    - `na.rm`, `na.omit`, `na.exclude`, `na.pass` 등
3. 객체의 불변성
    - 객체가 수정되지 않고 각 원소가 변경될 때마다 새로운 객체를 생성하여 저장함
    - 메모리, 속도 문제
4. 많은 자료를 다루면서 순환문이 불가피할 때
    - `Rcpp` 사용 (C++ 코드를 R에서 구동)

-> 할 수 있다면 **내장 함수**와 **벡터 연산** 권장

예시

```r
TestVector <- c(1:100000)
# Calculate the elapsed time for a code run: system.time()
system.time(TestCumSum <- cumsum(as.numeric(TestVector)))
```

```
##    user  system elapsed
##       0       0       0
```

```r
system.time( { TestSum <- numeric(length(TestVector))
               for(i in 1:length(TestVector)){
               if(i==1){
                    TestSum[1] <- TestVector[1]
               }else{
                    TestSum[i] <- TestSum[i-1] + TestVector[i]
               } } } )
```

```
##    user  system elapsed
##   0.062   0.000   0.048
```

오늘의 학습 내용 (조건문, 반복문, 함수)

조건문과 반복문 그리고 함수

출처: Chris_moden, Car_conveyor via wikimedia commons

조건문과 반복문 그리고 함수

출처: TE connectivity

흐름 제어 (조건문과 반복문)

조건문

1. 조건에 따라 코드의 수행 여부를 결정.
    - 예) 입장객의 나이가 7살 미만이면 오른쪽으로 나이가 7살 이상이면 왼쪽으로 나가게 하세요.
2. R에서는 주로 `if` 혹은 `ifelse` 함수를 이용하여 조건문 처리
    - 복잡한 혹은 복합적인 조건문
        - `if(condition){...TRUE...}else{...FALSE...}`
    - 한 줄 정도로 표현되는 단순한 조건문
        - `ifelse(condition, ...TRUE..., ...FALSE...)`

조건문 사용법

```
Attendance <- sample(30, 20); Attendance
```

```
##  [1] 24 20  6  8 18 21 30  1 13 19  3 23 28  7 27 10  9 15
## [19] 12 11
```

```
Direction  <- character(20); Direction
```

```
##  [1] "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""
## [19] "" ""
```

```
# if(condition){...TRUE...}else{...FALSE...}
for(i in 1:length(Attendance)){
    if(Attendance[i] < 7){
        Direction[i] <- "Right"
    }else{
        Direction[i] <- "Left"
    }
}
```

조건문 사용법

```
Direction
```

```
##  [1] "Left"  "Left"  "Right" "Left"  "Left"  "Left"  "Left"
##  [8] "Right" "Left"  "Left"  "Right" "Left"  "Left"  "Left"
## [15] "Left"  "Left"  "Left"  "Left"  "Left"  "Left"
```

```
# ifelse(condition, ...TRUE..., ...FALSE...)
for( i in 1:length(Attendance) ){
    Direction[i] <- ifelse(Attendance[i] < 7, "Right", "Left")
}
Direction
```

```
##  [1] "Left"  "Left"  "Right" "Left"  "Left"  "Left"  "Left"
##  [8] "Right" "Left"  "Left"  "Right" "Left"  "Left"  "Left"
## [15] "Left"  "Left"  "Left"  "Left"  "Left"  "Left"
```

다른 방법

```
# We can achieve our goal using the characteristics of vector
Direction[Attendance < 7] <- "Right"
Direction[Attendance >= 7] <- "Left"
Direction
```

```
##  [1] "Left"  "Left"  "Right" "Left"  "Left"  "Left"  "Left"
##  [8] "Right" "Left"  "Left"  "Right" "Left"  "Left"  "Left"
## [15] "Left"  "Left"  "Left"  "Left"  "Left"  "Left"
```

```
# We can also achieve the goal using data frame
TestDf <- data.frame(Attendance, Direction)
TestDf$Direction <- "Left"
TestDf$Direction[TestDf$Attendance < 7] <- "Right"
TestDf$Direction
```

```
##  [1] "Left"  "Left"  "Right" "Left"  "Left"  "Left"  "Left"
##  [8] "Right" "Left"  "Left"  "Right" "Left"  "Left"  "Left"
## [15] "Left"  "Left"  "Left"  "Left"  "Left"  "Left"
```

반복문

1. 주어진 조건동안 블록 안의 명령을 반복적으로 수행
    - `for`, `while`, `repreat`
2. 가장 많이 이용되는 반복문은 `for` 이다.
    - 'for(i in data){codes that are represented by i}
3. 쓰기 전에 내장 함수 혹은 벡터 연산으로 처리 가능한지 확인.
    - 많은 양의 데이터를 다룰 때 속도가 굉장히 느려짐.
4. `break`: 반복문 종료
5. `next` : 현재 명령을 수행하지 않고 다음 반복 시작.

반복문 사용법

```
# For-loop
for( i in data)
{
    Commands represented by i
}
# while-loop
while(condition)
{
    Commands
}
# repreat-loop
repeat{
    Commands
}
```

반복문 예시 (for)

```r
# Print integer from 1 to 10
for( i in 1:3){
    print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

```r
# Calculate accumulated sum from 1 to 100.
CumSum <- 0 # Value initialization
for( i in 1:100)
{
    CumSum <- CumSum + i
}
CumSum
```

```
## [1] 5050
```

반복문 예시 (for)

```
# Print words in the character vector
TestSentence <- c("Team Team Class is Fun")
TestChar <- strsplit(TestSentence, " ")[[1]]
TestChar
```

```
## [1] "Team"  "Team"  "Class" "is"    "Fun"
```

```
for( i in TestChar){
    print(i)
}
```

```
## [1] "Team"
## [1] "Team"
## [1] "Class"
## [1] "is"
## [1] "Fun"
```

반복문 예시 (while)

초등학교 입학 전까지 R 공부를 하지 않아도 된다고 말하는 코드를 작성해 봅시다.

```
age <- 0 # Value initialization
while(age < 7)
{
    print(paste("You are only ", age ,
                " years old. You don't need to study R.", sep=""))
    age <- age + 1
}
```

```
## [1] "You are only 0 years old. You don't need to study R."
## [1] "You are only 1 years old. You don't need to study R."
## [1] "You are only 2 years old. You don't need to study R."
## [1] "You are only 3 years old. You don't need to study R."
## [1] "You are only 4 years old. You don't need to study R."
## [1] "You are only 5 years old. You don't need to study R."
## [1] "You are only 6 years old. You don't need to study R."
```

반복문 예시 (repeat)

팀팀 클래스는 재밌다는 말을 다섯번 해봅시다.

```
count <- 0 # value initialization
repeat{
    count <- count + 1
    if(count > 5){break}
    print("Team-Team class is fun")
}
```

```
## [1] "Team-Team class is fun"
## [1] "Team-Team class is fun"
## [1] "Team-Team class is fun"
## [1] "Team-Team class is fun"
## [1] "Team-Team class is fun"
```

어떻게 하면 코드 순서만 바꿔서 6개의 문장을 만들 수 있을까요?

연산

수치 연산

▼ **표 3-4** 수치 연산자와 함수

연산자와 함수	의미
+, −, *, /	사칙 연산
n %% m	n을 m으로 나눈 나머지
n %/% m	n을 m으로 나눈 몫
n^m	n의 m승
exp(n)	e의 n승
log(x, base=exp(1))	logbase(x). 만약 base가 지정되지 않으면 loge(x)를 계산
log2(x), log10(x)	각각 log2(x), log10(x)를 계산
sin(x), cos(x), tan(x)	삼각 함수

출처: 서명구 (2014). R을 이용한 데이터 처리 & 분석 실무. 길벗

수치 연산 예시

```r
c(6+4, 6-4, 6/4, 4*2, 6%%4, 6%/%4)
```

```
## [1] 10.0  2.0  1.5  8.0  2.0  1.0
```

```r
c(4^6, exp(4), log(4), log(4, base=6), log2(4), log10(4))
```

```
## [1] 4096.0000000   54.5981500    1.3862944    0.7737056
## [5]    2.0000000    0.6020600
```

```r
c(sin(4), cos(4), tan(4), sin(pi/2), cos(pi/2), tan(pi/2))
```

```
## [1] -7.568025e-01 -6.536436e-01  1.157821e+00  1.000000e+00
## [5]  6.123234e-17  1.633124e+16
```

행렬 연산

▼ 표 2-16 행렬 연산자

연산자	의미
A + x	행렬 A의 모든 값에 스칼라 x를 더한다. 이외에도 -, *, / 연산자를 사용할 수 있다.
A + B	행렬 A와 행렬 B의 합을 구한다. 행렬 간의 차는 - 연산자를 사용한다.
A %*% B	행렬 A와 행렬 B의 곱을 구한다.

출처: 서명구 (2014). R을 이용한 데이터 처리 & 분석 실무. 길벗

행렬 및 벡터 연산 예시

```r
TestVector <- c(1:3)
TestVector * TestVector
```

```
## [1] 1 4 9
```

```r
TestVector %*% t(TestVector)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    2    4    6
## [3,]    3    6    9
```

```r
t(TestVector) %*% TestVector
```

```
##      [,1]
## [1,]   14
```

NA의 연산

R에서 NA는 결측치를 의미하며 일반적으로 다른 숫자와 연산할 때 NA를 출력한다.

```
c(NA+1, NA/0, NA/NA, NA * 0, NA - NA, 1/0)
```

```
## [1]  NA  NA  NA  NA  NA Inf
```

```
TestNA <- c(c(1:3), NA, c(3:6))
TestNA
```

```
## [1]  1  2  3 NA  3  4  5  6
```

```
c(sum(TestNA), sum(TestNA, na.rm=T),
  mean(TestNA), mean(TestNA, na.rm=T))
```

```
## [1]       NA 24.000000       NA  3.428571
```

NA의 처리

▼ 표 3-5 NA 처리 함수

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.

출처: 서명구 (2014). R을 이용한 데이터 처리 & 분석 실무. 길벗

NA의 처리 예시

```
TestDf <- data.frame(c(1, 2, NA), c(1, NA, NA), c(3, NA, 5))
# na.fail(TestDf)
na.exclude(TestDf)
```

```
##   c.1..2..NA. c.1..NA..NA. c.3..NA..5.
## 1           1            1           3
```

```
na.omit(TestDf)
```

```
##   c.1..2..NA. c.1..NA..NA. c.3..NA..5.
## 1           1            1           3
```

```
na.pass(TestDf)
```

```
##   c.1..2..NA. c.1..NA..NA. c.3..NA..5.
## 1           1            1           3
## 2           2           NA          NA
## 3          NA           NA           5
```

함수 (function)

함수의 정의

함수는 입력 인자들을 이용하여 새로운 결과 값을 계산하는 명령어들의 집합.

```
function_name <- function(parameter1, parameter2, ...){
     main process
    return(output)
}
```

```
Divider <- function(x, y){
    result <- x / y
    return(result)
}
c(Divider(3,5), Divider(y=5, x=3))
```

```
## [1] 0.6 0.6
```

중첩 함수

중첩함수: 다른 함수를 포함하고 있는 함수

```
NestedFn_1 <- function(x){
    return(function(y){ return( x + y )})
}
g <- NestedFn(1)
g(2)
```

```
## [1] 3
```

```
NestedFn_2 <- function(x,y){
    z <- function(y){return(y^3)}
    return(x/z(y))
}
NestedFn_2(3,5)
```

```
## [1] 0.024
```

함수 작성 예시 (큐 모듈 작성)

1. 큐 (Queue)는 먼저 들어온 데이터를 먼저 처리 (FIFO, First In First Out)하는 데 사용하는 자료 구조.
2. 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.

함수 작성 예시 (큐 모듈 작성)

```r
queue <- function(){
    q <- c(); qsize <- 0
    enqueue <- function(data){
        q <<- c(q, data)
        qsize <<- qsize + 1
        return(q)
    }
    dequeue <- function(){
        first <- q[1]
        q <<- q[-1]
        qsize <<- qsize - 1
        return(first)
    }
    size <- function(){return(qsize)}
    return(list(enqueue = enqueue, dequeue = dequeue, size = size))
}
```

함수 작성 예시 (큐 모듈 작성)

```
Testq <- queue()
Testq$enqueue(3)
```

```
## [1] 3
```

```
Testq$size()
```

```
## [1] 1
```

```
Testq$enqueue(5)
```

```
## [1] 3 5
```

```
Testq$dequeue()
```

```
## [1] 3
```

간단한 과제

큐 모듈을 적절히 변형하여 다음과 같은 생태계 지수를 계산할 수 있는 모형을 작성하세요.

1. 현재 생물 종과 각각의 종의 수를 입력하여 Shannon index 구하기.
2. 새로운 종이 들어왔을 때 각기 다른 생물 종의 숫자 (풍부도), Shannon index (다양성 지수)를 계산 및 출력.
3. 하나의 종이 사라졌을 때 생물 종의 숫자 (풍부도), Shannon index (다양성 지수)를 계산 및 출력.

- Shannon index (다양성): $S = -\sum_{i=1}^{R} p_i log(p_i)$
- Abundance (풍부도): R