응용 데이터 사이언스
제어와 함수

명지대학교 데이터 사이언스 연계 전공

2018. 10. 26(금)

1 Scripts

- *R Studio - File - New File - R Script*
- 순차적으로 실행하게 될 함수들을 모아 놓은 텍스트(재연 가능한 분석)

  - 스크립트 실행하기 : `source()`[1]

  - 재연 가능한 연구를 위한 환경 설정

    R Studio - Options - General - Restore .Rdata into workspace at startup [ ][2]

    ```
    rm(list=ls())
    ```

2 제어문 : 조건 또는 반복

- 일련의 함수들 중 일부를 **조건에 따라**, 혹은 **반복적으로** 실행해야 할 경우가 있다.
- 조건문 : `if/else, ifelse, switch`

```
if (s == "here") {
  print("Kookmin Univ.")
} else {
```

[1] 현재의 환경(변수, 함수)를 재구성하는 방법은 R 스크립트를 다시 실행하는 방법과 현재의 변수, 함수를 스크립트(`dump()`), 또는 데이터 화일(`save.image()`)로 저장하는 방법이 있다. 다시 불러들일 경우 `source()`, `load()`를 사용한다.

[2] *Options - General - Save workspace to .RData on exit [Never]* 설정을 사용할 수도 있다.

```
  print("Somewhere else than Kookmin Univ.")
}
```

Listing 1: `if/else` 구문

```
s2 <- ifelse(s=="here",
             "Kookmin Univ.", # if s=="here"
             "Somewhere else than Kookmin Univ") # if s!="here"
```

Listing 2: `ifelse` 함수

```
x <- "two"
switch(x,
  one = 1,
  two = 2,
  3)

x <- "two"; z <- 3
switch(x,
  one=1,
  two={
  y <- paste(x,"is entered.")
  z + 1},
  three=z^2)
```

Listing 3: `switch` 함수

- 반복문 : `for, while, repeat / next, break`

```
s=0
for (i in 1:10)
  s = s + i
print(s)
```

Listing 4: `for` 구문

```
s=0; i=1
while (i <= 10) {
  s=s+i;
```

```
4
5   i=i+1
6 }
7 print(s)
```

Listing 5: `while` 구문

```
1 s=0; i=1
2 repeat {
3   s=s+i; i=i+1
4   if (i>10) break
5 }
6 print(s)
```

Listing 6: `repeat` 구문

- 연습문제

  - 길이 1의 `numeric` 벡터를 받아 `1`이면 `"one"`, `2`이면 `"two"`, 그 밖에는 `"other than one or two"`을 출력하세요. 이때 `if`를 활용한 방법과 `switch`를 활용한 방법을 모두 구현하세요.

  - `1`부터 `100`까지 모든 짝수를 합하는 스크립트를 `for (i in 1:100)`, `while`, `repeat`를 활용하여 작성하세요.

  - `1`부터 시작하여 '`3, 7, 23`의 배수이며 `2`의 약수가 아닌 수'까지 모든 정수를 더하는 스크립트를 `for (i in 1:1000)`, `while`, `repeat`를 활용하여 작성하세요.

3 R 반복문 다시 보기

3.1 반복문의 기본적인 구조

1. 결과를 저장할 장소(변수)를 마련한다.
2. 특정한 값을 따라, 혹은 주어진 조건이 만족할 때까지 반복하도록 한다.
3. 반복되는 내용

```
s=0                 # 결과를 저장할 변수 s
for (i in 1:10)     # 변수 가i 부터1 까지10 반복된다.
  s = s + i         # 반복되어 수행되는 작업
print(s)
```

Listing 7: 반복 구문의 일반적인 구조

```
x = c(1, 3, 5, 9, 15)
s = rep(NA, 5)           # 결과를 저장할 변수 s
#for (i in 1:5)          # 변수 가i 벡터 의x 길이에 맞춰 변한다.
#for (i in 1:length(x))
for (i in seq_along(x))
  s[i] = x[i]^2          # 반복되어 수행되는 작업
print(s)
```

Listing 8: 벡터 x의 모든 원소에 대해 제곱한 값을 구하기

3.2 반복문의 대체

```
#01. Vectorized function
x <- c(1, 3, 5, 9, 15)
s <- sqrt(x)

#02. sapply
x <- c(1, 3, 5, 9, 15)
s <- sapply(x, sqrt)
```

Listing 9: 반복문을 대체하여 벡터 x의 모든 원소에 대해 제곱근을 구하기

```
#01. if the function is not vectorized
tonum = function(x) {
  switch(x,
    one = 1,
    two = 2,
    3)
  }
x <- c("one","three","two","four","two")
```

```
s <- tonum(x)
#Error in switch(x, one = 1, two = 2, 3) : EXPR must be a length 1
    vector

tonumV = Vectorize(tonum)
s <- tonumV(x)

#02. sapply with 'Vectorize'-d function
x <- c("one","three","two","four","two")
s <- sapply(x, tonumV)
```

Listing 10: `Vectorize` 함수

3.3 `for`의 몇 가지 변형

- `for (x in xs)`, `for (nm in names(xs))`
- 결과의 길이가 가변적일 때

```
x <- c(1,3,2,4)
result = vector("list", length(x))
#result = c()
for (i in seq_along(x)) {
  result[[i]] = rep(x,x)
  #result = c(result, rep(x,x))
  }
result <- unlist(result)
```

Listing 11: 결과가 가변적일 때 반복문의 예시

- 반복의 횟수가 가변적일 때 : `while`, `repeat/break`

4 반복문의 속도 비교

동일한 반복 작업을 수행하는 여러 가지 다른 방법의 속도를 비교해보자. 행렬의 경우, `system.time()`를 사용하여 측정한 속도의 순서는 다음과 같다.

R의 내장 함수 > `apply` > `for`

```r
mat <- matrix(1:1000, 1000, 1000, byrow=T)

# R 내장 함수
result <- colSums(mat)

# apply
result <- apply(mat, 2, sum)

# for
result <- rep(NA, 1000) # result <- NA, length(result) <- 1000
for (i in 1:ncol(mat)) {
  result[i] <- sum(mat[,i])
}
```

Listing 12: `colSums`, `apply`, `for`의 속도 비교

5 연습 문제

- 음이항 분포Negative Binomial Distribution는 베르누이 시행의 결과 Success/Fail의 한 결과가 일정 횟수가 나올 때까지 필요한 베르누이 시행의 수를 나타내는 분포이다.
- 동전을 던져서 앞면이 3번 나올 때까지 필요한 횟수를 세어보자.
- `sample(c("S","F"), 1, prob=c(.5, .5))`의 결과는 50%의 확률로 `'S'`, 50%의 확률로 `'F'`이다. 이 함수를 이용하여 `'S'`가 5번 나올 때까지 시행을 반복하는 반복문을 작성해 보자. (이를 1000번 반복해보자.)

6 함수

- 스크립트와 같이 반복해서 활용하게 될 코드를 모아 놓은 것.
- **매번 동일하게 실행되는 부분**과 **달라지는 부분**으로 구분해 볼 수 있다.

- 예) 다음은 1에서 5까지의 합, 1에서 10까지의 합, 1에서 20까지의 합을 구하고 있다.

```
s=0
for {i in 1:5} {
  s = s + i
  }
print(s)

s=0
for (i in 1:10) {
  s = s + i
  }
print(s)

s=0
for (i in 1:20) {
  s = s + i
  }
print(s)
```

Listing 13: 1부터 n까지 더하기

- 여기서 공통되는 부분을 함수로 나타내는 방법은 다음과 같이 진행할 수 있다.

1) 공통되는 부분을 확인한 후, 달라지는 부분을 변수로 치환한다.

2) 전체를 `{ }`로 감싼다.

3) 반환값을 명시한다: `return( )`

4) 함수의 이름과 인자를 명시한다: `= function(a, b, ...)`

```
s=0
for {i in 1:n} {
  s = s + i
  }
```

```
print(s)
```

Listing 14: 달라지는 부분을 변수로 나타낸다.

```
{
  s=0
  for {i in 1:n} {
    s = s + i
    }
  print(s)
}
```

Listing 15: 전체를 { }로 감싼다.

```
{
  s=0
  for {i in 1:n} {
    s = s + i
    }
  print(s)
  return(s)
}
```

Listing 16: 반환값을 명시한다.

```
sumToN = function(n) {
  s=0
  for {i in 1:n} {
    s = s + i
    }
  print(s)
  return(s)
}
```

Listing 17: 함수의 이름과 인자를 명시한다.

7 함수의 인자

- 인자가 여럿일 때 구분하는 방법은 1) 이름으로, 2) 위치로, 3) 부분 매칭(partial matching)이 있다.

```
sumAToB = function(startNumber, endNumber) {
  s=0
  for {i in startNumber:endNumber} {
    s = s + i
    }
  print(s)
  return(s)
}
```

Listing 18: `startNumber`에서 `endNumber`까지 더하는 함수

1) 이름으로 : `sumAToB(startNumber=1, endNumber=5)`, `sumAToB(endNumber=5, startNumber=1)`

2) 위치로 : `sumAToB(1, 5)`

3) 부분 매칭으로 : `sumAToB(startNum = 1, end=5)`

- 인자는 기본값(default value)을 설정해 줄 수 있고, 기본값이 주어진 인자는 함수를 부를 때 생략할 수 있다.

```
sumAToB = function(startNumber=1, endNumber=10) {
  s=0
  for {i in startNumber:endNumber} {
    s = s + i
    }
  print(s)
  return(s)
}
```

Listing 19: 기본값이 주어진 인자들

- 함수는 ...(dot-dot-dot)을 활용하여 불특정 다수의 인자를 받아들일 수 있다. 이때에는 부분 매칭이 불가능하다. 보통 **그 밖의** 인자들을 다른 함수로 넘길 때 사용된다.

```
sqPlot = function(x, y, ...) {
  plot(x^2, y, ...)
}
```

Listing 20: ... 활용의 예

- 만약 함수가 제대로 작동하기 위해 인자에 특별한 조건이 있다면 `stopifnot( )`을 사용할 수 있다.

```
boxcox = function(x, lambda=1)
{
  stopifnot(length(lambda)==1)
  if (lambda != 0) {
    return((x^lambda-1)/lambda)
  } else {
    log(x)
  }
}
```

Listing 21: `stopifnot` 활용의 예

8 함수와 인자의 클래스

- Generic function: 인자의 클래스(`class`)에 따라 함수의 행동이 달라지는 함수이다. `methods( )`를 통해 클래스에 따른 함수를 확인할 수 있다.

```
methods(print)
methods('print')
```

Listing 22: 제너릭 함수인 `print`에 대해 `methods( )`

9 그 밖의 몇 가지

- R은 기본적으로 Call by value이다. 함수의 인자는 인자의 주소가 아니라 값이 함수로 전달된다.
- 함수 내에서 사용된 변수는 함수가 끝나는 순간 사라진다.
- 함수 밖에서 할당된 변수는 참조할 수 있지만 '=' 또는 '<-'로 수정할 수 없다. 따라서 R의 함수 내 변수는 흔히 말하는 지역 변수(local variables)라고 할 수 있다.
- 만약 함수 밖에서 할당된 변수에 새로운 값을 할당하려면 연산자 <<-를 사용한다.

10 디버깅(Debugging)

R studio에는 함수 내부를 디버깅할 수 있는 도구가 마련되어 있다. 특히 break-point를 지정하는 방법을 숙지하자. 에러가 발생되었을 때, *Show Traceback*, *Rerun with Debug* 등을 사용할 수 있다.