

이 장에서는 복잡한 데이터 처리에 필요한 R의 중요 패키지와 코드 테스트 및 디버깅 방법을 알아본다. 이 장에서 설명할 패키지들에 대한 간략한 설명이다.

## 5.1 데이터 처리 및 가공 패키지

### 5장에서 설명할 주요 R패키지

패키지	용도
sqldf	SQL을 사용한 데이터 처리
plyr	데이터를 분할하고(split), 분할된 결과에 함수를 적용한 뒤(apply), 그 결과를 재조합(combine)
reshape2	데이터의 모양을 바꾸거나 요약
data.table	R의 데이터 프레임을 대신 더 빠르고 편리한 데이터 타입
foreach	apply 계열 함수들과 for 문을 대신할 수 있는 반복문 구조
doParallel	멀티코어를 사용한 프로그램의 병렬적 수행 기능
testthat	R 코드의 기능 테스트를 위한 유닛 테스트 프레임워크

## 5.2 SQL을 사용한 데이터 처리

R에는 많은 데이터 처리 함수가 있어 데이터를 편리하게 조작할 수 있다는 장점이 있다. 그러나 한편으로는 원하는 형태로 데이터를 만들기 위해 여러 가지 함수를 알아야 하는 점이 부담이 되기도 한다. sqldf는 이런 부담을 털어버리는 데 큰 도움이 되는 패키지로, SQL문을 사용할 줄 아는 사용자가 더욱 쉽게 데이터를 접근할 수 있게 해준다.

sqldf는 SQL 명령이 주어지면 자동으로 스키마를 생성하고 데이터를 테이블로 로드한 뒤 SQL 문을 실행한다. 그리고 SQL의 실행 결과는 다시 R로 로드된다. 이 작업은 자동으로 이뤄지기 때문에 사용자가 힘들여 DB를 설치하고 환경을 설정하는 작업이 필요 없다. 또한, 성능 최적화가 최대한 이뤄진 DB 기술을 활용하게 되어 데이터 처리 성능도 상당히 우수하다.

sqldf 패키지의 메인 함수는 sqldf()이다.

**sqldf::sqldf**: 데이터 프레임에 SQL SELECT 질의를 수행한다.

```
sqldf::sqldf(  
  x, #SQL SELECT 문  
  stringAsFactors = FALSE #문자열을 팩터로 반환할지 또는 문자열로 반환할지 여부  
)  
#반환 값을 데이터 프레임이다.
```

sqldf를 사용하려면 다음과 같이 패키지를 설치한다.

```
install.packages('sqldf')  
library(sqldf)
```

sqldf()를 사용해 아이리스 데이터를 살펴보자. iris에는 세 종류의 붓꽃 종류가 저장되어 있으며 붓꽃 종은 Species 컬럼에 저장되어 있다. 이를 확인해보자.

```
> sqldf('select distinct Species from iris ')
  Species
1   setosa
2 versicolor
3  virginica
```

이번에는 setosa에 속하는 데이터에서 Sepal.Length의 평균을 구해보자.

```
sqldf("select avg(Sepal_Length) from iris where Species = 'setosa'")
```

Note R과 달리 SQL에서 '.'은 테이블 컬럼명이 될 수 없으므로, Sepal.Length가 아니라 SepalLength로 컬럼명을 적어야 한다. 또한, SQL에서 대소문자 구별은 없으므로 SepalLengt 대신 sepal length로 적어도 된다.

만약 앞에 보인 작업을 R의 기본 함수들로 수행한다면 subset()으로 원하는 종의 데이터를 얻은 뒤 mean()을 적용하면 된다.

```
> mean(subset(iris, Species == 'setosa')$Sepal.Length)
[1] 5.006
```

종별 Sepal.Length의 평균은 SQL의 group by를 사용해 손쉽게 처리할 수 있다.

```
sqldf('select Species,avg(Sepal_Length) from iris group by Species')
```

비교를 위해 split(), sapply()를 사용해 같은 명령을 실행하는 다음 코드를 살펴보자.

```
> sapply(split(iris$Sepal.Length, iris$Species),mean )
      setosa versicolor  virginica 
      5.006      5.936      6.588
```

sqldf 패키지에서는 디스크를 저장소로 사용한다거나 매번 sqldf()를 수행할 때마다 DB에 데이터 프레임을 저장했다가 처리하고 삭제하는 대신 한 번 만들어둔 데이터를 재사용하는 등의 많은 최적화가 가능하다.

### 5.3 분할, 적용, 재조합을 통한 데이터 분석

plyr 패키지는 데이터를 분할하고(split), 분할된 데이터에 특정 함수를 적용한 뒤(apply), 그 결과를 재조합(combine) 하는 세 단계로 데이터를 처리하는 함수들을 제공한다. plyr의 입력은 배열, 데이터 프레임, 리스트가 될 수 있다. 출력 역시 배열, 데이터 프레임, 리스트가 될 수 있으며 아무런 결과도 출력하지 않을 수도 있다.

plyr은 데이터의 분할, 계산, 조합을 한 번에 처리해주어 여러 함수로 처리해야 할 일을 짧은 코드로 대신 해준다. 뿐만 아니라 입력과 출력에서 다양한 데이터 타입을 지원해주어 데이터 변환의 부담을 크게 줄여준다.

plyr의 데이터 처리 함수들은 {adl}{adl}ply 형태의 5글자 함수명을 사용한다.

첫 번째 글자는 입력 데이터 타입에 따라 각각 배열(a), 데이터 프레임(d), 리스트(l)로 정해진다.

두 번째 글자는 출력 데이터 타입으로 a,d,l 또는 \_로 정해지는데 이 중 \_는 아무런 출력도 내보내지 않음을 뜻한다.

예를 들어, `adply()`는 입력이 배열, 출력이 데이터 프레임이다. 한편 `lply()`는 입력과 출력이 리스트이다. 다음은 `plyr`의 형태와 의미를 정리한 것이다.

`plyr`에는 이외에도 데이터 프레임 또는 배열을 입력으로 받고 `a,d,p, _` 유형의 출력을 지원하는 `m{adp}_ply()` 형태의 특별한 함수들이 있다. 이는 `mapply()`와 유사하게 다수의 인자를 함수에 넘겨 처리하는 함수지만 출력을 좀 더 유연하게 지정할 수 있는 점이 다르다.

이 절에서는 `plyr`의 함수들 중 가장 유용한 `adply()`, `ddply()`, `mdply()`에 대해 설명한다. 또, 각 그룹의 계산을 도와주는 유틸리티 함수인 `transform()`, `mutate()`, `summarise()`, `subset()`에 대해서 알아본다.

먼저 `plyr` 패키지를 설치하고 로드한다.

```
install.packages('plyr')
library('plyr')
```

### 5.3.1 adply()

`adply()`는 배열(a)을 받아 데이터 프레임(d)를 반환하는 함수다. 그러나 입력이 반드시 배열일 필요는 없다. 그보다는 주어진 입력을 숫자 색인으로 읽을 수 있는가(즉, 행렬처럼 다룰 수 있는 형태의 데이터인가)하는 점이 중요하다. 이런 이유로 데이터 프레임도 숫자 색인으로 각 행이나 열을 접근할 수 있어 `adply()`를 적용할 수 있다.

**`plyr::adply`: 배열을 분할하고 함수를 적용한 뒤 결과를 데이터 프레임으로 반환한다.**

```
plyr::adply(
  .data, #행렬, 배열, 또는 데이터프레임
  #함수를 적용할 방향, 1(행방향), 2(컬럼 방향), c(1,2)는 행과 열 모두 적용
  .margins,
  .fun = NULL #.margin 방향으로 잘려진 데이터에 적용할 함수
)
#반환 값은 데이터 프레임이다.
```

`adply()`와 `apply()`는 비슷하다. 그러나 `apply()`는 행 방향으로 처리할 때 각 컬럼에 서로 다른 데이터 타입이 섞여 있다면 예상치 못한 타입 변환이 발생할 수 있다. 예를 들어, 다음 코드에서 볼 수 있듯, `apply()`에 숫자형 컬럼만 입력으로 주었을 경우에는 그 값이 제대로 숫자로 넘어오지만 문자열이 섞이면 데이터가 모두 문자열로 변환된다.

```
> apply(iris[,1:4],1,function(row){print(row)})
Sepal.Length Sepal.Width Petal.Length Petal.Width
      5.1         3.5         1.4         0.2
Sepal.Length Sepal.Width Petal.Length Petal.Width
      4.9         3.0         1.4         0.2

> apply(iris,1,function(row){print(row)})
Sepal.Length Sepal.Width Petal.Length Petal.Width
      "5.1"         "3.5"         "1.4"         "0.2"
Species
"setosa"
```

이러한 변환이 발생한 이유는 `apply()`가 한 가지 타입만 저장할 수 있는 '행렬'로 결과를 반환하기 때문이다. 본래 `apply()`는 벡터, 행렬, 리스트 중 한 가지 타입으로 결과를 반환할 수 있다. 그리고 결과가 한 행이면 벡터를 반환하고, 여러 행이면 행렬을 반환하며, 각 행마다 컬럼 개수가 다르다면 리스트를 반환한다. 위의 예에서는 각 행의 컬럼 개수가 5로 모두 일치하므로 리스트를 반환하지 않고 행렬로 결과가 반환된 것이다.

반면 `adply()`를 사용해 결과를 데이터 프레임으로 변환하면 결과 타입이 문자열로 모두 바뀌는 현상을 피할 수 있다.

다음은 `adply()`를 사용해 데이터 프레임의 각 행을 보면서 `Sepal.Length`가 5.0이상이고 `Species`가 `setosa`인지 여부를 확인한 다음 그 결과를 새로운 컬럼 `V1`에 기록하는 예이다.

```
> adply(iris,
+       1,
+       function(row){
+         row$Sepal.Length >= 5.0 &
+         row$Species == "setosa"
+       })
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	V1
1	5.1	3.5	1.4	0.2	setosa	TRUE
2	4.9	3.0	1.4	0.2	setosa	FALSE

위 예에서는 `adply()`에 인자로 넘긴 함수의 반환 값이 단순한 boolean 값이고, 그 결과가 임의의 컬럼명 `V1`에 저장되었다. 그러나 최종 반환 값이 데이터 프레임인 경우 함수의 반환 값을 데이터 프레임으로 하는 것이 안전하며, 데이터 프레임을 반환할 경우 계산 값을 저장한 컬럼명을 적절히 지정할 수 있다. 다음은 앞서와 같은 계산을 수행하지만 함수 내부에서 데이터 프레임을 반환하는 예다.

```
head(adply(iris,
+         1,
+         function(row) {
+           data.frame(
+             sepal_ge_5_setosa = c(row$Sepal.Length) >= 5.0 &
+                                   row$Species == 'setosa'
+           )
+         }))
```

### 5.3.2 ddply()

`ddply()`는 데이터 프레임(d)을 입력으로 받아 데이터 프레임(d)을 내보내는 함수다.

**`plyr::ddply()`: 데이터 프레임을 분할 하고 함수를 적용한 뒤 결과를 데이터 프레임으로 반환한다.**

```
plyr::ddply(
  .data,
  .variables, #데이터를 그룹 지을 변수명
  .fun = NULL
)
#반환 값은 데이터 프레임이다.
```

`adply()`와 `ddply()`의 가장 큰 차이점이라면 `adply()`는 행 또는 컬럼 단위로 함수를 적용하는 반면 `ddply()`는 `.variable`에 나열한 컬럼에 따라 데이터를 나눈 뒤 함수를 적용한다는 점이다.

다음은 `iris` 데이터에서 `sepal.Length`의 평균을 `species`별로 계산하는 예다. 두 번째 인자인 데이터를 그룹 짓는 변수는 `.`()안에 기록한다.

```
ddply(iris,
      .(Species),
      function(sub) {
        data.frame(Sepal.width.mean =
                    mean(sub$Sepal.width))
      }
    )
```

여러 변수로 그룹을 짓고자 한다면 .()안에 조건들 또는 컬럼명들을 콤마로 구분해서 나열한다. 다음은 붓꽃의 종과 Sepal.Length가 5.0보다 큰지 여부 두 가지 조건으로 데이터를 그룹 지은 뒤 각 그룹마다 Sepal.Width의 평균을 계산한 예다.

```
> ddply(iris,
+       .(Species, Sepal.Length>5.0),
+       function(sub) {
+         data.frame(Sepal.width.mean =
+                     mean(sub$Sepal.width))
+       }
+ )
```

	Species	Sepal.Length > 5	Sepal.width.mean
1	setosa	FALSE	3.203571
2	setosa	TRUE	3.713636
3	versicolor	FALSE	2.233333
4	versicolor	TRUE	2.804255
5	virginica	FALSE	2.500000
6	virginica	TRUE	2.983673

(데이터 분석을 위한 분할-적용-병합 전략)에 실린 미국 프로야구 선수 데이터가 저장된 baseball 데이터를 활용한 예를 살펴보자.

```
> head(baseball)
```

	id	year	stint	team	lg	g	ab	r	h	x2b	x3b
4	ansonca01	1871	1	RC1		25	120	29	39	11	3
44	forceda01	1871	1	WS3		32	162	45	45	9	4
68	mathebo01	1871	1	FW1		19	89	15	24	3	1
99	startjo01	1871	1	NY2		33	161	35	58	5	1
102	suttoez01	1871	1	CL1		29	128	35	45	3	7
106	whitede01	1871	1	CL1		29	146	40	47	6	5

	hr	rbi	sb	cs	bb	so	ibb	hbp	sh	sf	gidp
4	0	16	6	2	2	1	NA	NA	NA	NA	NA
44	0	29	8	0	4	0	NA	NA	NA	NA	NA
68	0	10	2	1	2	0	NA	NA	NA	NA	NA
99	1	34	4	2	3	0	NA	NA	NA	NA	NA
102	3	23	3	1	1	0	NA	NA	NA	NA	NA
106	1	21	2	2	4	1	NA	NA	NA	NA	NA

데이터의 각 행에서 선수가 해당 연도에 기록한 성적이 들어 있다.

다음은 선수 ansonca01의 기록을 살펴본 예다.

```
> head(subset(baseball, id=='ansonca01'))
```

	id	year	stint	team	lg	g	ab	r	h	x2b
4	ansonca01	1871	1	RC1		25	120	29	39	11
121	ansonca01	1872	1	PH1		46	217	60	90	10
276	ansonca01	1873	1	PH1		52	254	53	101	9

```

398 ansonca01 1874      1 PH1      55 259 51  87   8
525 ansonca01 1875      1 PH1      69 326 84 106  15
741 ansonca01 1876      1 CHN NL  66 309 63 110   9
      x3b hr rbi sb cs bb so  ibb hbp sh sf gidp
4      3  0  16  6  2  2  1  NA  NA NA NA  NA
121    7  0  50  6  6 16  3  NA  NA NA NA  NA
276    2  0  36  0  2  5  1  NA  NA NA NA  NA
398    3  0  37  6  0  4  1  NA  NA NA NA  NA
525    3  0  58 11  6  4  2  NA  NA NA NA  NA
741    7  2  59 NA  NA 12  8  NA  NA NA NA  NA

```

baseball 데이터에 ddp1y()를 사용해 각 선수가 출전한 게임 수(컬럼명 'g')의 평균을 구해보자. 각 선수별로 데이터를 그룹 짓기 위해 .(id)를 사용하고, 분할된 각 그룹마다 g의 평균을 계산하면 선수마다의 평균 게임 수가 된다.

```

> head(ddply(baseball, .(id), function(sub){mean(sub$g)}))
      id      v1
1 aaronha01 143.39130
2 abernte02  40.05882
3 adairje01  77.66667
4 adamsba01  25.36842
5 adamsbo03  85.40000
6 adcocjo01 115.23529

```

## 그룹마다 연산을 쉽게 수행하기

지금까지 살펴본 plyr의 예에서는 adply() 또는 ddp1y()에 임의의 사용자 정의 함수를 넘겨주어 분석을 수행했다. 그러나 공통적으로 자주 사용하는 유형의 계산은 transform(), mutate(), summarise(), subset()을 사용하면 더 간단히 표현할 수 있다. 다음 표에 이들 세 함수를 보았다.

## 데이터 변환 편의 함수

**base::transform** : 객체(예를 들면, 데이터 프레임)을 변환한다.

## transform()

base::transform()은 연산 결과를 데이터 프레임의 새로운 컬럼에 저장하는 함수다. 이를 사용해 baseball 데이터에 각 행이 선수의 몇 년차 통계인지를 뜻하는 cyear 컬럼을 추가해보자. 다음 코드는 데이터를 선수 id로 분할하여 그룹 지은 뒤, 각 그룹에서 year의 최솟값과 현재 행의 year값의 차이를 cyear에 저장한다.

```

> head(ddply(baseball, .(id), transform,cyear = year - min(year)+1))
      id year stint team lg   g  ab   r   h x2b
1 aaronha01 1954      1  ML1 NL 122 468  58 131  27
2 aaronha01 1955      1  ML1 NL 153 602 105 189  37
3 aaronha01 1956      1  ML1 NL 153 609 106 200  34
4 aaronha01 1957      1  ML1 NL 151 615 118 198  27
5 aaronha01 1958      1  ML1 NL 153 601 109 196  34
6 aaronha01 1959      1  ML1 NL 154 629 116 223  46
      x3b hr rbi sb cs bb so  ibb hbp sh sf gidp cyear
1    6 13  69  2  2 28 39  NA   3  6  4   13    1
2    9 27 106  3  1 49 61   5   3  7  4   20    2
3   14 26  92  2  4 37 54   6   2  5  7   21    3
4    6 44 132  1  1 57 58  15   0  0  3   13    4
5    4 30  95  4  1 59 49  16   1  0  3   21    5
6    7 39 123  8  0 51 54  17   4  0  9   19    6

```

## mutate()

plyr에는 `base::transform()`을 개선한 `plyr::mutate()` 함수가 있다. 이 함수는 여러 컬럼을 데이터 프레임에 추가할 때 바로 앞서 추가한 컬럼을 뒤에 추가하는 컬럼에서 참조할 수 있어 편리하다. 예를 들어, 아래 코드에서는 `mutate()`를 이용해 `transform()` 예에서처럼 `cyear`를 계산한 뒤 `log(year)`를 `log_cyear` 컬럼으로 추가한다. 만약 `mutate()`가 아닌 `transform()`을 사용하면 이 경우 에러가 발생한다.

```
> head(ddply(baseball, .(id), mutate, cyear = year-min(year)+1, log_cyear =
log(cyear)))
```

	id	year	stint	team	lg	g	ab	r	h	x2b	x3b	hr	rbi	sb	cs	bb	so	ibb	hbp	sh	sf	gidp	cyear
1	aaronha01	1954	1	ML1	NL	122	468	58	131	27													
2	aaronha01	1955	1	ML1	NL	153	602	105	189	37													
3	aaronha01	1956	1	ML1	NL	153	609	106	200	34													
4	aaronha01	1957	1	ML1	NL	151	615	118	198	27													
5	aaronha01	1958	1	ML1	NL	153	601	109	196	34													
6	aaronha01	1959	1	ML1	NL	154	629	116	223	46													

	id	year	stint	team	lg	g	ab	r	h	x2b	x3b	hr	rbi	sb	cs	bb	so	ibb	hbp	sh	sf	gidp	cyear
1											6	13	69	2	2	28	39	NA	3	6	4	13	1
2											9	27	106	3	1	49	61	5	3	7	4	20	2
3											14	26	92	2	4	37	54	6	2	5	7	21	3
4											6	44	132	1	1	57	58	15	0	0	3	13	4
5											4	30	95	4	1	59	49	16	1	0	3	21	5
6											7	39	123	8	0	51	54	17	4	0	9	19	6

	id	year	stint	team	lg	g	ab	r	h	x2b	x3b	hr	rbi	sb	cs	bb	so	ibb	hbp	sh	sf	gidp	cyear
1																							0.0000000
2																							0.6931472
3																							1.0986123
4																							1.3862944
5																							1.6094379
6																							1.7917595

## summarise()

`plyr::summarise()`는 데이터의 요약정보를 만드는 데 사용하는 함수다. `transform()`이 인자로 주어진 계산 결과를 새로운 컬럼에 추가한 데이터 프레임을 반환하는 반면, `summarise()`는 계산 결과만을 담은 새로운 데이터 프레임을 반환한다. `baseball` 데이터에서 각 선수의 최초 데이터가 몇 년도인지를 조사해보자. 아래 예에서는 데이터를 `id`마다 그룹 지은 뒤 그룹마다 `year`의 최솟값을 계산한 `minyear` 컬럼을 생성했다. 연산 시 사용할 함수로 `summarise()`를 지정했으므로 그룹을 짓는 변수인 `id`와 각 그룹의 요약 값 `minyear`만 저장된 데이터 프레임이 반환됐다.

```
> head(ddply(baseball, .(id), summarise, minyear = min(year)))
```

	id	minyear
1	aaronha01	1954
2	abernte02	1955
3	adairje01	1958
4	adamsba01	1906
5	adamsbo03	1946
6	adcocjo01	1950

만약 여러 요약 값을 구하고 싶다면 요약 값 계산을 계속 나열한다. 다음은 선수별 최초 데이터가 저장된 `minyear`와 `maxyear`를 출력하는 코드이다.

```
> head(ddply(baseball, .(id), summarise, minyear = min(year), maxyear =
max(year)))
```

	id	minyear	maxyear
1	aaronha01	1954	1976
2	abernte02	1955	1972
3	adairje01	1958	1970
4	adamsba01	1906	1926
5	adamsbo03	1946	1959
6	adcocjo01	1950	1966

## subset()

이름에서 쉽게 알 수 있듯이 subset()은 각 분할별로 데이터를 추출하는 데 사용한다. subset()에 조건을 지정하면 그룹별로 조건을 만족하는 행만 추출된다.

다음은 각 선수별로 가장 많은 수의 게임을 플레이한 해의 기록을 찾는 예이다. baseball을 id별로 그룹 지은 뒤 가장 많은 게임의 수를 max(g)로 구하고 현재 행의 게임 수가 max(g)와 같은 행만 subset()으로 선택했다.

```
> head(ddply(baseball, .(id), subset, g==max(g)))
```

	id	year	stint	team	lg	g	ab	r	h	x2b
1	aaronha01	1963	1	ML1	NL	161	631	121	201	29
2	abernte02	1965	1	CHN	NL	84	18	1	3	0
3	adairje01	1965	1	BAL	AL	157	582	51	151	26
4	adamsba01	1913	1	PIT	NL	43	114	13	33	6
5	adamsbo03	1952	1	CIN	NL	154	637	85	180	25
6	adcocjo01	1953	1	ML1	NL	157	590	71	168	33

	x3b	hr	rbi	sb	cs	bb	so	ibb	hbp	sh	sf	gidp
1	4	44	130	31	5	78	94	18	0	0	5	11
2	0	0	2	0	0	0	7	0	1	3	0	0
3	3	7	66	6	4	35	65	7	2	4	2	26
4	2	0	13	0	NA	1	16	NA	0	3	NA	NA
5	4	6	48	11	9	49	67	NA	0	8	NA	15
6	6	18	80	3	2	42	82	NA	2	6	NA	22

## mdply()

m{adl}\_ply(), 즉 maply(), mdply(), mldply(), m\_ply() 함수는 데이터 프레임 또는 배열을 인자로 받아 각 컬럼을 주어진 함수에 적용하고 그 실행 결과들을 조합한다. 여기서는 이들 중 mdply()에 대해 살펴본다.

예를 들어, 평균과 표준 편차를 저장한 데이터프레임을 가정해보자.

```
> (x<-data.frame(mean = 1:5, sd = 1:5))
```

	mean	sd
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5

mdply()를 사용하면 위 데이터 프레임의 각 행을 rnorm() 함수의 mean, sd에 대한 인자로 넘겨 주어 실행한 뒤 그 결과를 데이터 프레임으로 모을 수 있다. 다음 예에서는 각 mean, sd 마다 2개 씩 난수를 발생 시켰다.



```
> mdply(x, rnorm, n=2)
  mean sd      v1      v2
1    1  1  1.789871 0.7700061
2    2  2  0.362995 2.9994683
3    3  3  3.477577 4.6278793
4    4  4  3.373420 5.7551733
5    5  5 12.439353 5.3008255
```

## 5.4 데이터 구조의 변형과 요약

reshape2는 데이터의 모양을 바꾸거나 그룹별 요약 값을 계산하는 함수들을 담고 있는 패키지다. 변환된 데이터는 측정치를 variable과 value라는 두 컬럼으로 표현하므로 데이터의 통계치 계산이 편리해진다.

reshape2가 제공하는 변환은 크게 melt()와 cast() 두 함수로, 이 둘은 4장에서 다룬 stack(), unstack() 함수와 유사한 기능을 한다. 다음 표에 이 두 함수의 역할에 대해 정리했다.

### reshape2의 melt()와 cast() 함수

함수	의미
melt()	여러 컬럼으로 구성된 데이터를 데이터 식별자(id), 측정변수(variable), 측정값(value)이라는 3개 컬럼으로 변환한다. 만약 한 데이터에 대해 다수의 측정 변수와 측정값이 있다면 이들은 여러 행을 표현된다. 이렇게 변환된 결과는 variable 컬럼에 측정 대상이 기록되어 있으므로 각 variable 마다 value의 통계 값을 계산하는 것이 편리하다.
cast()	melt()된 데이터를 다시 여러 컬럼으로 변환된다. 데이터에 여러 측정 변수와 측정값이 존재한다면 이들은 모두 새로운 컬럼으로 변환된다. cast() 로 변환된 결과는 마치 스프레드시트에 입력한 데이터 모양과 유사하므로 분석자가 읽기 쉽다. 또한, cast() 시 melt()된 데이터의 여러 행이 한 셀에 대응하는 경우 데이터의 요약 값을 자동으로 계산해준다.

### melt()

melt() 함수는 식별자, 측정 변수, 측정치 형태로 데이터를 재구성하는 함수다.

french\_fries에 melt()를 적용해보자. french\_fries에서 데이터를 식별하는 식별자로 볼 수 있는 부분은 time, treatment, subject, rep의 1:4 컬럼을 id, 나머지 컬럼들을 측정치로 놓고 melt()를 적용해보자.

```
> m<-melt(french_fries, id.vars = 1:4)
> head(m)
  time treatment subject rep variable value
1    1          1      3    1  potato    2.9
2    1          1      3    2  potato   14.0
3    1          1     10    1  potato   11.0
4    1          1     10    2  potato    9.9
5    1          1     15    1  potato    1.2
6    1          1     15    2  potato    8.8
> head(french_fries)
  time treatment subject rep potato buttery grassy
61    1          1      3    1    2.9      0.0    0.0
25    1          1      3    2   14.0      0.0    0.0
62    1          1     10    1   11.0      6.4    0.0
26    1          1     10    2    9.9      5.9    2.9
63    1          1     15    1    1.2      0.1    0.0
27    1          1     15    2    8.8      3.0    3.6
  rancid painty
```

61	0.0	5.5
25	1.1	0.0
62	0.0	0.0
26	2.2	0.0
63	1.1	5.1
27	1.5	2.3

이 결과에서 볼 수 있듯이 여러 컬럼으로 나열된 측정치들을 variable, value의 두 개 컬럼을 사용해 여러 행으로 변환하는 것이 melt()의 역할이다.

이렇게 변환된 결과는 측정 변수가 variable이라는 컬럼에 값으로 저장되어 variable마다 그룹 지어 통계치를 계산하는 작업이 간단해진다. 예를 들어, 식용유 종류 마다 측정 변수의 평균을 구하는 코드는 다음과 같다.

```
> ddply(m,.(variable), summarise, mean = mean(value,na.rm = T))
  variable      mean
1  potato 6.9525180
2 buttery 1.8236994
3  grassy 0.6641727
4  rancid  3.8522302
5  painty 2.5217579
```

위 코드에서 na.rm = T는 NA값들을 제거하기 위해 사용된 것이다. NA를 포함한 행들을 확인하기 위해 complete.cases()를 사용해보자.

complete.cases() 함수는 해당 행의 모든 값이 NA가 아닌 경우 T, 해당 행의 값이 하나라도 NA를 포함하고 있는 경우 F를 반환한다. 이를 사용해 NA를 포함하는 행을 찾을 수 있다.

즉, NA는 F이다.

```
> french_fries[!complete.cases(french_fries),]
  time treatment subject rep potato buttery grassy
315   5         3      15   1    NA      NA      NA
455   7         2      79   1   7.3      NA    0.0
515   8         1      79   1  10.5      NA    0.0
520   8         2      16   1   4.5      NA    1.4
563   8         2      79   2   5.7       0    1.4

  rancid painty
315    NA     NA
455   0.7     0
515   0.5     0
520   6.7     0
563   2.3     NA
```

NA를 포함하는 측정치를 melt() 시 제외하려면 na.rm = T를 지정해야한다.

```
> m<-melt(id = 1:4, french_fries, na.rm = T)
> head(m)
  time treatment subject rep variable value
1    1         1       3    1  potato    2.9
2    1         1       3    2  potato   14.0
3    1         1      10    1  potato   11.0
4    1         1      10    2  potato    9.9
5    1         1      15    1  potato    1.2
6    1         1      15    2  potato    8.8
```

## cast()

cast()는 결과로 얻고자 하는 데이터 타입에 따라 dcast(), acast()로 구분하여 사용한다. dcast()는 결과로 데이터 프레임, acast()는 벡터, 행렬, 배열을 반환한다.

다음 코드는 french\_fries 데이터를 melt() 후, dcast()를 사용해 다시 원 데이터로 변환하는 예를 보여준다. 코드 마지막에 사용된 함수인 identical()은 두 데이터가 완전히 동일한 객체인지를 알려주는 함수다. rownames()에 NULL을 부여한 것은 french\_fries에 특이하게 부여된 행 번호 (61,25,62,... 형태로 행 번호가 부여돼 있음)를 무시하기 위함이다.

```
m<-melt(french_fries, id.vars = 1:4)
r<-dcast(m,time+treatment+subject+rep~...)
rownames(r)<-NULL
rownames(french_fries)<-NULL
identical(r,french_fries)
```

## 데이터 요약

cast()의 또 다른 유용성은 데이터의 요약 값을 계산하는 기능이 있다는 점이다. 그러나 melt()된 데이터에 plyr 패키지만 잘 적용해도 이 절에서 설명할 대부분의 요약치 계산은 쉽게 수행할 수 있다.

이에 cast()가 제공하는 요약 기능은 reshape2가 제공하는 하나의 보너스 정도로 생각해라.

reshape2에서 데이터 요약을 수행하려면 melt()에서 사용한 것 보다 적은 개수의 식별자를 dcast()의 formula에 지정하면 된다. 예를 들어, melt() 시 사용한 time, treatment, subject, rep 였고, 이를 dcast()에서 복구할 때는 이들을 모두 time+treatment+subject+rep ~ ... 형태로 지정했다. 그러나 이들 식별자 중 하나를 포물러 '~' 왼쪽에서 제외한다면 dcast() 시 여러 행이 하나의 셀로 모이게 될 것이다. 바로 이렇게 모인 값들에 요약치를 계산한다는 것이 핵심이다.

다음 예는 dcast() 시 time 만 포물러에서 ~ 왼쪽에 적고 측정 변수를 오른쪽에 적은 예를 보여준다. 이 경우 같은 time 값에 해당하는 여러 개의 데이터가 결과 데이터 프레임에 모이게 되고, dcast()는 자동으로 length()을 적용한다.

```
> m<-melt(french_fries, id.vars = 1:4)
> dcast(m,time~variable)
Aggregation function missing: defaulting to length
  time potato buttery grassy rancid painty
1     1      72      72     72     72     72
2     2      72      72     72     72     72
3     3      72      72     72     72     72
4     4      72      72     72     72     72
5     5      72      72     72     72     72
6     6      72      72     72     72     72
7     7      72      72     72     72     72
8     8      72      72     72     72     72
9     9      60      60     60     60     60
10    10      60      60     60     60     60
```

length 보다도 sum, mean 또는 임의의 함수를 적용해 구할 수 있다. 다음 데이터는 time에 따라 평균값이 어찌 달라지는지 보여준다.

```
dcast(m,time~variable,mean, na.rm=T)
  time  potato  buttery  grassy  rancid  painty
1     1  8.562500  2.236111  0.9416667  2.358333  1.645833
2     2  8.059722  2.722222  1.1819444  2.845833  1.444444
3     3  7.797222  2.102778  0.7500000  3.715278  1.311111
```

```

4      4 7.713889 1.801389 0.7416667 3.602778 1.372222
5      5 7.328169 1.642254 0.6352113 3.529577 2.015493
6      6 6.670833 1.752778 0.6736111 4.075000 2.341667
7      7 6.168056 1.369014 0.4208333 3.886111 2.683333
8      8 5.431944 1.182857 0.3805556 4.272222 3.938028
9      9 5.673333 1.586667 0.2766667 4.670000 3.873333
10     10 5.703333 1.765000 0.5566667 6.068333 5.291667

```

```

> dcast(m,time~variable,sum, na.rm=T)
   time potato buttery grassy rancid painty
1      1  616.5   161.0    67.8   169.8   118.5
2      2  580.3   196.0    85.1   204.9   104.0
3      3  561.4   151.4    54.0   267.5    94.4
4      4  555.4   129.7    53.4   259.4    98.8
5      5  520.3   116.6    45.1   250.6   143.1
6      6  480.3   126.2    48.5   293.4   168.6
7      7  444.1    97.2    30.3   279.8   193.2
8      8  391.1    82.8    27.4   307.6   279.6
9      9  340.4    95.2    16.6   280.2   232.4
10     10  342.2   105.9    33.4   364.1   317.5

```

dcast() 호출 시 포뮬러에서 ~ 우측은 측정 변수를 적는 곳으로, 이곳에 적은 변수는 결과에서 새로운 컬럼이 된다. 따라서 식별자를 ~ 오른쪽에 적으면 해당 식별자 값마다 variable의 요약치를 새로운 컬럼으로 한 결과를 얻을 수 있다. 다음은 각 time 마다 (treatment, variable) 순서쌍에 해당하는 value의 평균을 계산한 예다.

```

> head(dcast(m,time~treatment+variable,mean, na.rm=T))
   time 1_potato 1_buttery 1_grassy 1_rancid
1      1 7.925000 1.795833 0.9041667 2.758333
2      2 7.591667 2.525000 1.0041667 3.900000
3      3 7.770833 2.295833 0.8166667 4.650000
4      4 8.404167 1.979167 1.0250000 2.079167
5      5 7.741667 1.366667 0.7708333 4.279167
6      6 6.079167 1.825000 0.4666667 4.337500
   1_painty 2_potato 2_buttery 2_grassy 2_rancid
1 2.1500000 8.775000 2.491667 0.9958333 1.716667
2 1.9750000 8.537500 3.125000 0.9500000 2.141667
3 1.1166667 7.637500 2.079167 0.7250000 2.895833
4 0.4666667 8.204167 1.608333 0.6416667 3.512500
5 3.0083333 6.933333 1.858333 0.5833333 3.641667
6 2.5541667 7.016667 2.054167 0.9208333 3.841667
   2_painty 3_potato 3_buttery 3_grassy 3_rancid
1 0.8083333 8.987500 2.420833 0.9250000 2.600000
2 0.6625000 8.050000 2.516667 1.5916667 2.495833
3 1.5625000 7.983333 1.933333 0.7083333 3.600000
4 1.8583333 6.533333 1.816667 0.5583333 5.216667
5 0.7375000 7.308696 1.704348 0.5478261 2.630435
6 2.7500000 6.916667 1.379167 0.6333333 4.045833
   3_painty
1 1.979167
2 1.695833
3 1.254167
4 1.791667
5 2.313043
6 1.720833

```

위 결과에서 1\_potato는 treatment가 1 일때, potato값을, 2\_potato는 treatment가 2 일 때를 의미한다.

```
> head(ddply(m,.(time,treatment,variable),
+         function(rows){
+             mean(rows$value,na.rm = T)
+         })
+ ))
  time treatment variable      v1
1    1          1  potato 7.925000
2    1          1 buttery 1.795833
3    1          1  grassy 0.904167
4    1          1  rancid 2.758333
5    1          1  painty 2.150000
6    1          2  potato 8.775000
```

## 5.5 데이터 테이블 : 더 빠르고 편리한 데이터 프레임

데이터 테이블은 R의 기본 데이터 타입인 데이터 프레임을 대신하여 사용할 수 있는 더 빠르고 편리한 데이터 타입이다. 데이터 테이블의 첫 번째 장점은 속도다. 조건을 만족하는 데이터를 색인을 사용하여 빠르게 찾을 수 있으며 참조를 통한 데이터 갱신을 지원하여 데이터 복사에 따른 비용을 줄인다. 두 번째 장점은 연산의 편의성이다. 데이터를 조건에 따라 그룹 지은 뒤 통계치를 계산하는 작업을 편리하고 빠르게 수행할 수 있다.

먼저 데이터 테이블을 설치해보자.

### 데이터 테이블 생성

데이터 테이블은 데이터 프레임을 만드는 것과 동일한 문법으로 생성한다. 또는 `as.data.frame()` / `as.data.table()`을 사용해 데이터 프레임과 데이터 테이블 간의 상호작용이 가능하다.

다음은 아이리스 데이터에 대한 예시이다. 더 많은 데이터를 출력하려면 `iris_table[1:n,]`처럼 행 번호를 직접 지정하거나 `print(dt, nrow = Inf)` 명령을 사용한다.

```
> (iris_table <- as.data.table(iris))
   Sepal.Length Sepal.Width Petal.Length
1:           5.1          3.5          1.4
2:           4.9          3.0          1.4
3:           4.7          3.2          1.3
4:           4.6          3.1          1.5
5:           5.0          3.6          1.4
---
146:          6.7          3.0          5.2
147:          6.3          2.5          5.0
148:          6.5          3.0          5.2
149:          6.2          3.4          5.4
150:          5.9          3.0          5.1
   Petal.Width Species
1:          0.2  setosa
2:          0.2  setosa
3:          0.2  setosa
4:          0.2  setosa
5:          0.2  setosa
---
146:          2.3 virginica
147:          1.9 virginica
```

```
148:      2.0 virginica
149:      2.3 virginica
150:      1.8 virginica
```

다음은 데이터 테이블을 직접 생성하는 예다.

```
> (x<-data.table(x=c(1,2,3),y=c('a','b','c')))  
   x y  
1: 1 a  
2: 2 b  
3: 3 c
```

이 절의 시작 부분에서 언급한 바와 같이 데이터 테이블은 데이터 프레임과 동일하게 취급한다. 다음 코드는 데이터 테이블의 클래스가 data.frame을 포함하고 있음을 보여준다. 클래스가 data.frame을 포함하므로 summary, print, plot 등의 데이터 프레임을 처리하는 함수들이 데이터 테이블에도 동일하게 동작한다. 다시 말해, data.frame을 인자로 기대하는 함수에 데이터 테이블을 넘겨도 많은 경우 별 문제없이 동작한다.

```
> class(data.table())  
[1] "data.table" "data.frame"
```

이렇게 만든 데이터 테이블들의 목록은 tables()로 열람할 수 있다.

```
> iris_table<-as.data.table(iris)  
> x<-data.table(x=c(1,2,3),y=c('a','b','c'))  
> tables()  
      NAME NROW NCOL MB  
1: iris_table  150    5  0  
2:           x     3    2  0  
  
      COLS  
1: Sepal.Length, Sepal.Width, Petal.Length, Petal.Width, Species  
2:                                     x, y  
  
      KEY  
1:  
2:  
Total: 0MB
```

## 데이터 접근과 그룹 연산

데이터 테이블은 데이터 프레임이 사용되는 대부분의 위치에서 문제없이 동작한다. 하지만 데이터를 접근할 때 사용하는 []에는 몇 가지 주의를 기울여야 한다.

데이터 테이블의 데이터는 [행, 표현식, 옵션] 형태로 접근한다. 행은 '행 번호' 또는 '행을 선택할지 나타내는 진릿값'으로 지정한다. 이는 데이터 프레임과 동일하다. 다음은 행 번호와 진릿값을 사용하는 두 경우의 예를 보여준다.

```
> DT <- as.data.table(iris)  
> DT[1,]  
   Sepal.Length Sepal.Width Petal.Length Petal.Width  
1:           5.1           3.5           1.4           0.2  
   species  
1:  setosa  
> head(DT[DT$Species == 'setosa',])  
   Sepal.Length Sepal.Width Petal.Length Petal.Width  
1:           5.1           3.5           1.4           0.2
```

```

2:      4.9      3.0      1.4      0.2
3:      4.7      3.2      1.3      0.2
4:      4.6      3.1      1.5      0.2
5:      5.0      3.6      1.4      0.2
6:      5.4      3.9      1.7      0.4
  Species
1: setosa
2: setosa
3: setosa
4: setosa
5: setosa
6: setosa

```

다음은 1행의 Sepal.Length 컬럼을 접근하는 예다. 컬럼명을 'Sepal.Length'가 아닌 Sepal.Length 로 따옴표 없이 지정했음을 유의하기 바란다. 따옴표로 묶는 형식은 데이터 프레임에서 사용하는 문법이다.

```

> DT[1,Sepal.Length]
[1] 5.1

```

다음은 list() 안에 컬럼명을 나열하여 여러 컬럼을 접근한 예다.

```

> DT[1,list(Sepal.Length,Species)]
  Sepal.Length Species
1:      5.1 setosa
> DT[,mean(Sepal.Length)]
[1] 5.843333
> DT[,mean(Sepal.Length-Sepal.Width)]
[1] 2.786

```

지금까지는 컬럼명을 그대로 코드에 작성했다. 만약 데이터 테이블에서 []의 두 번째 인자로 컬럼명을 담은 문자열 또는 컬럼 번호를 지정하고자 한다면 with = FALSE 옵션을 줘야 한다.

```

> DT<-as.data.table(iris)
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.1          3.5          1.4          0.2
2          4.9          3.0          1.4          0.2
3          4.7          3.2          1.3          0.2
4          4.6          3.1          1.5          0.2
5          5.0          3.6          1.4          0.2
6          5.4          3.9          1.7          0.4
  Species
1 setosa
2 setosa
3 setosa
4 setosa
5 setosa
6 setosa
> iris[1,1]
[1] 5.1
> DT[1,1]
  Sepal.Length
1:      5.1
> DT[1,1,with=F]
  Sepal.Length

```

```

1:      5.1
> iris[1,1]
[1] 5.1
> DT[1,1]
      Sepal.Length
1:      5.1
> iris[1,c('Sepal.Length')]
[1] 5.1
> DT[1,c('Sepal.Length')]
      Sepal.Length
1:      5.1

```

데이터 테이블의 세 번째 인자에는 데이터를 그룹 지을 변수를 지정한다. 예를 들어, Sepal.Length의 평균값을 Species 별로 구하는 계산은 세 번째 인자로 by = 'Species'를 지정해 수행할 수 있다.

```

> DT[,mean(Sepal.Length),Species]
      Species      v1
1:    setosa 5.006
2: versicolor 5.936
3:  virginica 6.588

```

만약 그룹을 지을 때 사용할 변수가 여러 개라면 by에 컬럼명을 계속 나열하면 된다. 다음은 값을 저장한 x, 데이터의 분류를 저장한 y,z 를 가진 데이터 테이블에서 x의 평균을 y,z 두 값에 따라 계산한 예이다.

```

> DT<-data.table(x=c(1,2,3,4,5),
+               y=c('a','a','a','a','a'),
+               z=c('c','c','d','d','d'))
> DT
      x y z
1:  1 a c
2:  2 a c
3:  3 a d
4:  4 a d
5:  5 a d
> DT[,mean(x),by="y,z"]
      y z      v1
1:  a c 1.5
2:  a d 4.0

```

## Key를 사용한 빠른 데이터 접근

데이터 프레임에서 특정 컬럼에 특정 값이 들어 있는 행을 찾는 작업은 모든 행의 값을 하나하나 검토하는 방식으로 이뤄진다. 이에 데이터양이 많고 데이터 검색 작업의 횟수가 많다면 긴 수행 시간이 걸린다. 데이터 테이블을 사용한다면 이런 경우 값에 대한 색인을 이진 탐색 트리로 미리 생성해둘 수 있다. 이에 데이터 검색 작업의 횟수가 많을 경우 데이터 테이블이 데이터 프레임에 비해 빠른 속도로 동작한다. 데이터 테이블의 색인 생성을 위한 함수는 다음과 같다.

키를 만든 후 X[J(...), 표현식] 문법을 사용하면 ...에 지정한 값을 키로 가지는 행들을 찾는다. '표현식'에 아무것도 적지 않으면 데이터 테이블의 컬럼들이 그대로 반환되고, 표현식에 연산을 적으면 그 결과가 반환된다.

데이터 프레임에서 조건에 따른 행의 검색과 데이터 테이블에서 색인을 사용한 검색 속도 비교를 해보자.



다음은 x에 연속 균등 분포를 따르는 난수 값을, y에 각 알파벳 10,000 회씩 반복한 값을 저장한 데이터 프레임에서 y값이 C인 행들을 선택하는 예이다.

```
> DF<-data.frame(x=runif(26000),y=rep(LETTERS,each=10000))
> str(DF)
'data.frame': 260000 obs. of 2 variables:
 $ x: num 0.219 0.153 0.863 0.661 0.359 ...
 $ y: Factor w/ 26 levels "A","B","C","D",...: 1 1 1 1 1 1 1 1 1 ...
> tail(DF)
           x y
259995 0.1367344 Z
259996 0.2485927 Z
259997 0.9872642 Z
259998 0.2334237 Z
259999 0.1387530 Z
260000 0.5200952 Z
> system.time(x<-DF[DF$y=='C',])
사용자   시스템 elapsed
      0       0       0

> DT<-as.data.table(DF)
> setkey(DT,y)
> system.time(x<-DT[J('C'),])
사용자   시스템 elapsed
      0       0       0
```

### key를 사용한 데이터 테이블 병합

key는 여러 데이터 테이블의 병합에도 사용할 수 있다. 예를 들어, DT1, DT2 두 데이터 테이블이 있을 때, DT1[DT2, 표현식]은 DT1에서 DT2에 대응하는 데이터를 찾는 방식으로 데이터를 병합한다. 다음 표를 보라.

다음 두 데이터 테이블을 가정해보자.

```
DT1<-data.table(x=runif(26000),
                y=rep(LETTERS,each=10000))
DT2<-data.table(y=c('A', 'B', 'C'),z=c('a', 'b', 'c'))
```

DT1[DT2,]는 DT1으로부터 y값이 A,B,C인 행을 찾아 병합한다.

```
> setkey(DT1, y)
> DT1[DT2,]
           x y z
1: 0.59487861 A a
2: 0.40375250 A a
3: 0.99956869 A a
4: 0.87794716 A a
5: 0.89406507 A a
---
29996: 0.37917209 C c
29997: 0.19118266 C c
29998: 0.01675704 C c
29999: 0.43845137 C c
30000: 0.91195260 C c
```

반면 DT2[DT1,]는 260,000개 행을 DT2로부터 검색하되 DT2에 대응하는 행이 없는 경우, z에 NA가 포함된 행이 반환된다. 이에 그 결과는 총 260,000개 이다.

```
> setkey(DT2, y)
> DT2[DT1,]
      y      z      x
1: A      a 0.7851983
2: A      a 0.6771699
3: A      a 0.4090483
4: A      a 0.4088542
5: A      a 0.2065352
---
259996: Z <NA> 0.5483318
259997: Z <NA> 0.5053930
259998: Z <NA> 0.8475649
259999: Z <NA> 0.5842968
260000: Z <NA> 0.9616494
```

데이터 테이블 간의 병합은 색인을 활용하므로 그 속도가 빠르다. 또 데이터 테이블 병합시 DT1[DT2,표현식]과 같이 얻고자 하는 결과의 표현식을 바로 지정할 수 있으므로, 데이터 병합과 동시에 계산을 수행할 수 있는 장점이 있다.

편리해 보이긴 하지만 데이터 프레임과 merge()를 사용해도 같은 결과를 얻을 수 있지 않을까?

물론 merge()를 사용해도 같은 결과를 얻을 수 있다. 그러나 데이터 테이블이 제공하는 테이블 병합 방식의 장점은 속도에 있다. 데이터 테이블이 훨씬 빠르다.

### 참조를 사용한 데이터 수정

3.7 객체의 불변성 절에서, R객체는 수정되지 않으며, 값을 수정하는 것처럼 보이는 경우에도 실제로는 새로운 객체를 매번 만든다고 설명했다. 이에 for문 안에서의 데이터 수정은 매우 긴 시간이 소요되고 이런 이유로 벡터화된 연산을 사용해야 한다.

데이터 테이블은 데이터 프레임이 지원하는 일반적인 데이터 수정 연산자 외에도 := 연산자를 사용한 데이터 수정 기능을 제공한다.

### 참조를 사용한 할당 문법

문법	의미
DT1[i, LHS:=RHS]	LHS에 RHS를 참조로 할당한다.
DT1[i,c('LHS1','LHS2')] := list(RHS1,RHS2)]	LHS1, LHS2에 RHS1,RHS2를 참조로 할당한다.

다음은 데이터 프레임과 데이터 테이블을 각각 사용하여 for문에서 첫 번째 컬럼에 1부터 1000까지의 값을 지정하는 예다. 다음 예를 보자.

```
> system.time({
+   for (i in 1:1000){
+     DF[i,1] <- i
+   }
+ })
사용자   시스템 elapsed
  0.03    0.00    0.03
> system.time({
+   for (i in 1:1000){
+     DT[i,v1 := i]
```

```
+ }
+ })
  사용자   시스템 elapsed
    0.44    0.05    0.53
```

## 리스트를 데이터 프레임으로 변환하기

lapply()를 비롯한 많은 R 함수는 결과를 리스트로 반환한다. 리스트를 반환 값으로 채택한 이유 중 하나는 아마도 결과 값의 데이터 타입을 다양하게 설정할 수 있다는 점 때문일 것이다.

그러나 많은 데이터 모델링 또는 시각화 함수에서 인자로 데이터 프레임을 받는다. 또, 아무래도 데이터 프레임이 보기 편하다. 이런 이유로 리스트를 데이터 프레임으로 변환할 필요가 종종 생긴다.

데이터 프레임을 함수 실행 거로가로 얻는 가장 간단한 방법은 ply의 ldply() 등과 같이 결과를 데이터 프레임으로 출력하는 함수를 사용하는 것이다. 하지만 안타깝게도 ldply() 함수의 속도는 lply()와 같이 리스트를 결과로 출력하는 경우에 비해 좋지 않다. 다음은 ply의 ldply()와 lply()의 성능을 비교한 예다.

```
Browse[1]> system.time( x <- ldply(1:10000, function(x){
+   data.frame(val = x,
+               val2 = 2 * x,
+               val3 = 2 / x,
+               val4 = 4 * x,
+               val5 = 4 / x
+   )
+ })))
  user   system elapsed
  3.55    0.01    3.58
Browse[1]> system.time( x <- llply(1:10000, function(x){
+   data.frame(val = x,
+               val2 = 2 * x,
+               val3 = 2 / x,
+               val4 = 4 * x,
+               val5 = 4 / x
+   )
+ })))
  user   system elapsed
  2.89    0.00    2.93
```

결과를 보면 같은 연산을 수행하는 코드임에도 불구하고 ldply()의 실행 소요 시간이 lply()의 1.22 배에 가까운 것을 알 수 있다. 이는 ldply()는 lply()를 사용해 리스트로 구성된 결과를 얻은 다음 이를 다시 데이터 프레임으로 변환하기 때문이다. 그리고 리스트를 데이터 프레임으로 만드는데는 생각보다 긴 시간이 소요된다.

이를 좀 더 직접적으로 알아보기 위해 리스트를 데이터 프레임을 변환하는 다음 예를 살펴보자.

```
> x<- lapply(1:10000, function(x){
+   data.frame(val = x,
+               val2 = 2 * x,
+               val3 = 2 / x,
+               val4 = 4 * x,
+               val5 = 4 / x
+   )
+ })
> system.time(y<-do.call(rbind,x))
  사용자   시스템 elapsed
    0.94    0.00    0.95
```

위 코드는 `do.call(rbind, 데이터 프레임)`을 사용해 리스트 안에 저장된 데이터 프레임을 하나로 합치는데 0.95초가 소요됐다. 긴 시간은 아니지만 컬럼 수가 많아지거나 데이터가 많아지면 이 시간은 급격히 증가하며 심지어 데이터 처리의 대부분의 시간이 여기에만 소요되기도 한다.

이러한 문제를 해결해주는 `data.table` 패키지의 함수가 `rbindlist()` 다. 다음 표에 `rbindlist()`에 대해 정리했다.

`rbindlist`는 인자로 데이터 테이블 또는 데이터 프레임의 리스트를 받아 하나의 데이터 테이블로 합쳐준다. 따라서 `lply`를 사용해 일단 데이터 테이블의 리스트로 결과를 만든 다음 이를 다시 하나의 데이터 테이블로 병합한다면 `ldply()`를 사용하는 것보다 속도가 매우 빠를 것이다.

```
> system.time(x <- ldply(1:10000, function(x){
+   data.frame( val = x,
+               val1 = x*2,
+               val2 = 2/x,
+               val3 = 4 * x,
+               val4 = 4/x)
+ })))
```

사용자	시스템	elapsed
3.58	0.00	3.61

```
> system.time(x<-lply(1:10000, function(x){
+   data.frame(val = x,
+               val1 = x*2,
+               val2 = 2/x,
+               val3 = 4 * x,
+               val4 = 4/x)
+ })))
```

사용자	시스템	elapsed
2.95	0.00	3.02

```
> system.time(x <- rbindlist(x))
```

사용자	시스템	elapsed
0.03	0.00	0.03

보다시피 데이터 프레임을 대신 만들어주는 `ldply()`는 3.61초가 걸린 반면, `lply`를 하고 `rbindlist`를 하면 3.02초가 걸린다.

## 5.6 더 나은 반복문

`foreach`는 `apply()` 계열 함수, `for()` 문 등을 대체할 수 있는 루프문을 위한 함수다. `for` 문과의 가장 큰 차이는 반환 값이 있고, `{}`가 아닌 `%do%` 문을 사용해 블록을 지정한다는 점이다.

`foreach` 패키지를 설치한다.

기본적인 예로 1~5 까지의 숫자를 루프를 돌면서 `%do%` 안에서 반환하는 예를 살펴보자. `..combine`이 지정되지 않았으므로 결과가 리스트로 반환된다.

```
> foreach(i= 1:5) %do%{
+   return(i)
+ }
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
```

```
[1] 3

[[4]]
[1] 4

[[5]]
[1] 5
```

.combine = c 를 지정하면 결과를 벡터로 받는다.

```
> foreach(i = 1:5, .combine = c) %do%{
+   return(i)
+ }
[1] 1 2 3 4 5
```

.combine에 rbind를 지정하면 결과를 행 방향으로 합친 데이터 프레임을 반환하며, cbind를 지정하면 컬럼 방향을 합친다.

```
> foreach( i = 1:5, .combine = rbind) %do%{
+   data.frame(val= i)
+ }
val
1  1
2  2
3  3
4  4
5  5
> foreach( i = 1:5, .combine = cbind) %do%{
+   data.frame(val= i)
+ }
val val val val val
1  1  2  3  4  5
```

그러나 cbind와 rbind는 속도가 느릴 수 있다. 따라서 일단 리스트로 결과를 받고 rbindlist()를 하는 방법에 대해서도 고려하기 바란다.

.combine에는 또한 연산자를 지정할 수 있다. 예를 들어, +를 지정하면 모든 결과를 합한 결과를 반환한다. 다음은 1부터 5까지의 합을 계산하는 예이다.

```
> foreach( i = 1:5, .combine = "+") %do%{
+   return(i)
+ }
[1] 15
```

## 5.7 병렬처리

doMC와 doParallel 패키지는 멀티코어를 활용해 프로그램을 병렬적으로 수행할 수 있게 해준다. 멀티코어를 활용한다는 뜻은 여러 프로세스를 실행하고 각 프로세스가 개별 CPU 코어에서 돌아가게 한다는 의미다. 이 두 패키지는 plyr 또는 foreach와 결합해 편리하게 사용할 수 있다.

윈도우 사용자는 doParallel 패키지를 사용해야 한다.

### 5.7.1 프로세스의 수 설정

병렬화를 사용하기 위해 가장 먼저 할 일은 몇 개의 프로세스를 사용할 것인지 선택해 registerDoParallel()에 지정하는 것이다.

```
> registerDoParallel(cores = 4)
> registerDoParallel(cores = 3)
```

R for Windows front-end	0%	36.8MB	0MB/s	0Mbps
R for Windows front-end	0%	36.8MB	0MB/s	0Mbps
R for Windows front-end	0%	36.8MB	0MB/s	0Mbps
R for Windows front-end	0%	36.8MB	0MB/s	0Mbps
R for Windows front-end	0%	36.8MB	0MB/s	0Mbps
R for Windows front-end	0%	36.8MB	0MB/s	0Mbps
R for Windows front-end	0%	36.8MB	0MB/s	0Mbps

registerDoParallel() 호출 후, 작업 관리자를 살펴보면 위 그림처럼 R for Windows front-end 프로세스가 시작된 것을 볼 수 있다. 이 프로세스들이 병렬 작업을 처리한다.

### **Note cores 개수의 결정**

#### **cores의 지정에는 크게 두 가지 측면을 고려할 수 있다.**

**첫째로**, 어느 정도의 입출력 작업이 있는가 하는 점이다. 디스크, DB, 네트워크 입출력이 발생하면 기기(디스크나 네트워크 인터페이스 카드)에서 응답이 올 때까지 해당 프로세스는 CPU를 사용하지 않는다. 따라서 입출력 작업이 많다면 코어보다 프로세스의 수를 더 많게 정해 입출력을 대기 중인 프로세스가 CPU를 사용하지 않는 동안 다른 프로세스들이 해당 CPU를 사용할 수 있게 할 수 있다. 그러나 입출력 작업이 거의 없는 상태에서 다수의 프로세스를 실행하면, 하나의 CPU 코어가 여러 프로세스를 실행해야 하고, 이 경우 하나의 CPU 자원을 위해 여러 프로세스가 경쟁하게 되어 전체 성능은 오히려 떨어질 수 있다.

**둘째로**, 여러 프로세스를 실행하면 각각이 메모리를 소비하게 된다는 점이다. 특히 사용 중인 알고리즘이나 모델링 패키지에 따라서 이 용량은 매우 커질 수 있다. 예를 들어, 어떤 데이터를 분석하는 데 통상 100MB 메모리가 소요된다고 하자, 만약 프로세스가 4개라면, 각 프로세스가 25MB씩 소모하여 총 100MB 메모리를 차지할 것 같지만 실제로는 이들 프로세스를 관리하는 데 따라 발생하는 오버헤드 때문에 그보다 큰 메모리를 사용할 수 있다. 따라서 사용 중인 컴퓨터의 메모리를 잘 관찰해 메모리 여분이 충분히 남는지, 그리고 불필요한 페이지링이나 스와핑이 발생하지 않는지 등 메모리 사용량을 유심히 지켜볼 필요가 있다. 만약 페이지링이나 스와핑이 발생한다면 디스크 입출력 작업으로 성능이 오히려 더 떨어질 수 있다.

모든 경우에 맞는 프로세스의 수 결정 공식이란 존재하지 않으므로 프로세스의 수는 상황에 따라 적절히 조절해나가야 한다. 만약 적절한 입출력이 있는 경우라면 프로세스의 수를 코어 개수 x2로부터 시작할 것을 추천한다. 하지만 CPU 작업이 대부분이라면 코어 개수만큼 프로세스의 수를 설정하거나 또는 registerDoParallel()의 기본값처럼 코어 개수의 절반을 선택할 수도 있다.


### **5.7.2 plyr의 병렬화**


plyr의 함수들 중 {adl}{adl}\_ply 형태 함수들의 도움말을 살펴보면 .parallel 옵션이 있다. 이 옵션 값이 TRUE로 설정되면 registerDoParallel()을 사용해 설정한 만큼의 프로세스가 동시에 실행되어 데이터를 병렬적으로 처리한다. 그리고 그 결과들은 자동으로 하나의 결과로 병합되어 사용자에게 반환된다. 그러나 .parallel의 기본값은 FALSE 이므로 이 옵션을 명시적으로 설정하지 않으면 registerDoParallel()을 호출했다 하더라도 병렬화가 이뤄지지 않는다.


그렇다면 설정을 어찌하는지 알아보자.


```
big_data <- data.frame(
  value = runif(NROW(LETTERS) * 2000000),
  group = rep(LETTERS, 2000000)
)


dplyr::big_data, .(group), function(x){
  mean(x$value)
}, .parallel = T)
```

 R for Windows front-end

 R for Windows front-end

 R for Windows front-end

 R for Windows front-end

 RStudio R Session

0%	675.3MB
0%	703.4MB
0%	657.8MB
0%	667.6MB
0.1%	1,075.7MB

```
> tail(big_data)
      value group
51999995 0.13832832 U
51999996 0.53320050 V
51999997 0.50858554 W
51999998 0.31705654 X
51999999 0.86496742 Y
52000000 0.03236805 Z

> dplyr::big_data, .(group), function(x){
+   mean(x$value)
+ }, .parallel = T)
$A
[1] 0.5001537

$B
[1] 0.4998935

$C
[1] 0.5001037

$D
[1] 0.4996509

$E
[1] 0.5000236
```

위 코드를 실행한 다음 프로세스를 보면 다수의 R 프로세스가 실행되어 있고 이들이 대량의 메모리를 소모함을 알 수 있다. 이처럼 대량의 데이터를 병렬로 처리할 때는 프로세스의 실행을 관찰하면서 시스템 전체의 가용 메모리가 얼마나 남아 있는지를 확인해야 한다. 메모리가 부족하면 부족한 메모리를 보충하기 위해 디스크를 메모리처럼 사용하게 되고, 디스크 입출력 작업 때문에 데이터 처리의 속도가 느려지는 스래싱(thrashing) 현상이 발생할 수 있기 때문이다.

만약 스래싱이 발생한다면 다음 증상들을 볼 수 있다.

- 시스템의 가용 메모리가 거의 남아 있지 않다.

- 디스크 입출력이 지속적으로 많이 발생한다.
- R프로세스의 CPU 사용량이 낮다.

### 5.7.3 foreach의 병렬화

앞에서 foreach() 함수에는 %do%를 사용해 실행할 명령의 블록을 지정해야 한다고 설명했다. 만약 foreach()에서 블록 내 명령을 병렬로 수행하고자 한다면 registerDoParallel()을 한 뒤, foreach()에 %do% 대신 %dopar%를 지정하면 된다.

예를 들어, 다음은 i값이 1부터 800,000까지 변할 때 이 값을 big\_data\$value에 더한 다음 그 평균을 매 i마다 계산한 예다.

```
big_data <- data.frame(
  value = runif(NROW(LETTERS) * 2000000),
  group = rep(LETTERS, 2000000)
)

foreach(i=1:800000) %dopar% {
  mean(big_data$value + i)
}
```

이 예에서는 설명을 위해 다소 작위적인 코드를 작성했다. 좀 더 실제적인 적용 사례로는 다양한 파라미터에 따라 모델을 만들어 보면서 가장 적절한 파라미터를 찾는 예를 들 수 있다. 예를 들어, 다수의 의사 결정 나무를 가진 랜덤 포레스트 모델을 만들 때 적절한 나무 개수를 설정하기 위해 다양한 나무의 수를 사용해 보는 경우를 들 수 있다. 다음은 이를 의사 코드로 표현한 것이다.

```
foreach(ntree = c(10,20,30,100,1000)) %dopar% {
  build_model(big_data, ntree = ntree)
}
```

## 5.8 유닛 테스트와 디버깅

유닛 테스트는 코드 단위(유닛)가 정확히 작동하는지 코드를 사용해 검증하는 방법을 말한다. 유닛 테스트 코드는 코드의 일반적인 사용 상황과 특이한 사용 상황(코너 케이스)을 테스트 하기 마련이므로 코드의 사용 방법을 설명하는 문서의 역할까지도 담당한다. 또 유닛 테스트를 작성하다 보면 함수의 사용자가 함수로부터 어떤 기능을 기대할지를 미리 생각할 수 있게 해주므로 테스트 주도 개발(TDD, Test Driven Development)이 가능해진다.

그러나 모든 코드를 항상 유닛 테스트로 완벽히 테스트할 수 있는 것은 아니다. 어떤 버그는 코드를 한 행씩 따라가야만 찾을 수 있다. 이런 이유로 디버깅 역시 또 하나의 중요한 개발 도구다.

### 5.8.1 testthat

R에는 유닛 테스트를 위한 패키지로 RUnit, testthat 등이 있으며, 여기서는 그중 좀 더 사용하기 편리한 testthat에 대해 설명하겠다. 먼저 testthat 패키지를 설치해라.

#### expect 함수들

유닛 테스트는 작성한 코드에 어떤 입력을 주었을 때 예상되는 출력이 실제로 반환되는지 확인하는 것을 기본으로 한다. 기댓값과 실제 반환 값을 비교하기 위해 사용하는 함수의 일부를 다음 표에 정리했다.



함수	의미
<code>expect_true(x)</code>	x가 참인가
<code>expect_false(x)</code>	x가 거짓인가
<code>expect_equal(object, expected)</code>	object가 기대되는 값 expected와 동일한가. 값의 비교에는 <code>all.equal()</code> 이 사용되며 <code>all.equal()</code> 은 두 객체가 (거의) 같은지를 비교한다. 따라서 부동소수의 경우 <code>epsilon</code> (매우 작은 값) 이하로 차이나는 두 값은 같은 값으로 취급한다.
<code>expect_equivalent(x,y)</code>	x가 y와 동등한가. <code>expect_equal()</code> 과 달리 속성(예를 들면, <code>rownames</code> )을 제거한 뒤 값만 비교하면 된다.

`expect_equal()`와 `expect_equivalent()`는 비슷하면서도 다른 점이 있어 다음의 예를 가져왔다.

`expect_equal()`은 완전히 같은 값의 경우에만 테스트가 통과하고 `expect_equivalent()`는 객체의 속성을 제외한 값만 비교한다.

```
> a<-1:3
> b<-1:3
> expect_equal(a,b)
> expect_equivalent(a,b)
> names(a) <- c('a', 'b', 'c')
> expect_equal(a,b)
Error: `a` not equal to `b`.
names for target but not for current
> expect_equivalent(a,b)
```

`expect_equal`는 완전히 같아야 하며, `expect_equivalent`는 속성만 같으면 된다.

`expect_equal()`을 사용한 테스트 방법을 피보나치 수열을 구하는 함수를 예로 들어 살펴보자. 피보나치 수열은 1,1,2,3,5,... 값을 반환해야 하지만 다음 코드에는 일부러 버그를 만들어 두었다. `fib(0) = 1`, `fib(1) = 1` 이어야 하지만 `fib(1) = 1` 부분이 빠져 있다.

```
fib <- function(n) {
  if(n==0){
    return(1)
  }
  if (n>0){
    return(fib(n-1)+fib(n-2))
  }
}
```

`expect_equal()`을 사용해 `fib(0)`이 1인지 확인해보자.

```
expect_equal(fib(0),1)
```

피보나치 수열의 첫 번째 값인 `fib(0)`을 구하면 정답인 1을 제대로 반환한다. 따라서 `expect_equal()`에 인자로 준 기댓값 1과 `fib(0)`의 값은 같다. 따라서 아무런 경고 메시지를 출력하지 않는다.

그러나 수열의 두 번째 값인 `fib(1)`을 확인해보면 버그 조건에 해당해 `fib(1)`의 값이 1과 같지 않다는 에러 메시지가 출력된다.

```
> expect_equal(fib(1),1)
Error: fib(1) not equal to 1.
Lengths differ: 0 is not 1
```

이처럼 잘못된 값이 반환된 것을 찾았다면 이에 따라 fib() 함수를 올바르게 수정해나가면 된다.

### 5.8.2 test\_that를 사용한 테스트 그룹화

서로 관련된 내용을 테스트하는 expect 문장들은 test\_that() 함수를 사용해 그룹으로 묶을 수 있다. 이렇게 하면 연관된 테스트가 하나의 블록으로 묶이므로 테스트 코드를 보기 좋게 구조화할 수 있다.

예를 들어, 피보나치 수열의 처음 두 개 값은 각각 1로 정해지는데, 이를 'base test'라 명명해 다음과 같이 묶을 수 있다.

```
> test_that('base case', {
+   expect_equal(1, fib(0))
+   expect_equal(1, fib(1))
+ })
Error: Test failed: 'base case'
* 1 not equal to fib(1).
Lengths differ: 1 is not 0
```

fib(2)이상은 재귀적으로 실행되므로 이를 'recursion test'라고 명명해 따로 묶을 수 있다. 다음은 버그가 없도록 수정한 fibo() 함수와 이를 base test와 recursion test로 나눠 테스트하는 예를 보여준다.

```
> fib <- function(n){
+   if(n==0 || n==1){
+     return(1)
+   }
+   if(n>=2){
+     return(fib(n-2)+fib(n-1))
+   }
+ }
> test_that('base case', {
+   expect_equal(1, fib(0))
+   expect_equal(1, fib(1))
+ })
> test_that('recursion case', {
+   expect_equal(2, fib(2))
+   expect_equal(3, fib(3))
+   expect_equal(5, fib(4))
+ })
```

테스트 중 아무런 오류도 발견되지 않았으므로 위 코드의 수행 시 특별히 출력되는 내용은 없다.

### 5.8.3 테스트 파일 구조

이 절에서는 코드와 테스트를 어떤 파일에 어떻게 기술해야 하는지를 살펴본다. 가장 일반적인 파일 구성은 비즈니스 로직이 되는 fibo()를 한 소스파일에 넣고, 테스트 코드는 별도의 소스 파일에 넣는 것이다. 한 가지 파일 구성의 예는 다음과 같다.

- **fibo.R** : 비즈니스 로직인 fibo() 함수를 작성해 저장
- **run\_tests.R** : 테스트들을 실행하기 위한 명령을 저장한 파일
- **tests/test\_fibo.R** : test 디렉터리에 fibo 함수에 대한 테스트 코드를 작성해 test\_fibo.R로 저장

fibo.R 함수에는 다음과 같이 fibo() 함수 코드만 작성해 넣으면 된다

```
fib <- function(n){
  if(n==0 || n==1){
    return(1)
  }
  if(n>=2){
    return(fib(n-2)+fib(n-1))
  }
}
```

run\_tests.R 은 다음과 같이 작성한다. 먼저 fibo.R을 source()하여 해당 파일에 있는 명령을 실행한다. 그러면 그 결과로 fibo() 함수가 정의될 것이다. 테스트 수행은 test\_dir() 함수를 사용하는데, 아래 예에서는 tests 디렉터리에 있는 모든 테스트 파일을 실행하고 그 결과의 요약을 얻게 했다.

```
require(testthat)
source('fibo.R')

test_dir('tests', reporter = 'summary')
```

tests 디렉터리의 test\_fibo.R의 첫 행에는 해당 파일 내 테스트들이 무엇에 대한 테스트인지를 명시하기 위해 contexts('fibonacci series')를 적는다. 이 예에서는 테스트 파일이 하나뿐이지만 여러 테스트 파일을 작성하면 context()에 기록한 정보가 유용하게 사용된다. 그 뒤 test\_that()들을 기술하는데, test\_test()은 앞서 설명한 대로 base test와 recursion test 두 가지 경우로 나눈다.

```
context('fibonacci series')

test_that('base case', {
  expect_equal(1, fib(0))
  expect_equal(1, fib(1))
})

test_that('recursion case', {
  expect_equal(2, fib(2))
  expect_equal(3, fib(3))
  expect_equal(5, fib(4))
})
```

테스트를 위한 준비가 모두 끝났다. run\_tests.R 파일에 있는 디렉터리에서 R을 실행하고 source('run\_test.R')을 수행하면 테스트가 실행되는 화면을 볼 수 있다. 테스트 실행 시에는 context()로 지정한 'fibonacci series' 라는 문자열이 출력되므로 어떤 테스트들이 수행 중인지 좀 더 쉽게 알 수 있다.

```
source("run_tests.R")
fibonacci series: .....

=== DONE =====
```

에러가 발생하는 경우의 출력을 살펴보기 위해 일부러 expect\_equal(0,fibo(5))를 recursion test 에 추가해 다시 run\_test.R를 실행해보자.

## 5.8.4 디버깅

코드가 원하는 대로 동작하지 않을 때 그 이유를 확인하는 방법에는 크게 print(), sprintf(), cat()을 사용해 메시지나 객체의 내용을 출력해보는 방법과 browser()를 사용한 코드 디버깅 방법이 있다. 다음 표에 이 절에서 설명할 함수들을 보았다.

## print()

print()는 주어진 객체를 화면에 출력한다. 따라서 print()문은 코드 중간 중간에 삽입해 변수에 할당된 값을 출력하거나 현재 어느 부분의 코드가 실행중인지를 쉽게 확인할 수 있다. 이때 객체를 문자열로 편리하게 변환해주는 paste() 함수가 유용하게 사용된다.

## paste()

paste()는 인자로 주어진 값들을 하나의 문자열로 합쳐 출력한다. 다음 예를 보자.

```
> paste('a',1,2,'b','c')
[1] "a 1 2 b c"
```

문자열 'a'(문자열 'a'를 저장한 길이 1인 벡터로 취급)와 1~6까지의 숫자를 저장한 길이 6인 벡터를 인자로 주면 'a'를 반복하여 사용하면서 두 벡터를 붙인다.

```
> paste('a',1:6)
[1] "a 1" "a 2" "a 3" "a 4" "a 5" "a 6"
```

sep는 여러 인자를 문자열로 붙일 때 사용하는 구분자다. 따라서 paste()에 sep = ""를 지정하면 "a"와 1:6 간의 공백은 사라진다. sep를 지정하는 대신 paste0()을 호출해도 마찬가지로 결과를 얻는다.

```
> paste('a',1:6, sep='')
[1] "a1" "a2" "a3" "a4" "a5" "a6"
> paste0('a',1:6)
[1] "a1" "a2" "a3" "a4" "a5" "a6"
```

collapse는 결과를 하나의 문자열로 만들 때 사용하는 구분자다. 다음은 collapse = ","를 지정해 위의 결과를 , 로 연결한 하나의 문자열로 만든 예다.

```
> paste('a',1:6, sep='', collapse = ',')
[1] "a1,a2,a3,a4,a5,a6"
```

다음 코드는 피보나치 수열을 구하는 함수가 잘 동작하는지 확인해보기 위해 코드 사이에 print()를 넣은 예다.

```
> fibo<-function(n){
+   if(n==1||n==2){
+     print('base case')
+     return(1)
+   }
+   print(paste0("fibo(",n-1,")+fibo(",n-2,")"))
+   return(fibo(n-1)+fibo(n-2))
+ }
>
> fibo(1)
[1] "base case"
[1] 1
> fibo(2)
[1] "base case"
[1] 1
> fibo(3)
[1] "fibo(2)+fibo(1)"
[1] "base case"
```

```
[1] "base case"
[1] 2
```

## sprintf()

sprintf()는 print()와 유사하지만 주어진 인자들을 특정한 규칙에 맞게 문자열을 변환해 출력해주는 차이점이 있다. 호출 형식은 sprintf(fmt, ...)과 같은 방식이다. 이때 포매팅 문자열 fmt에는 주어진 인자를 어떤 방식으로 포매팅할 것인지를 결정하는 특수한 문자열을 적을 수 있는데, 가장 중요한 포매팅 문자열 %d, %s, %f에 대해 알아볼 필요가 있다.

sprintf()의 가장 기본적인 사용 예를 살펴보자. 아래 코드에서 첫 번째 예는 주어진 인자를 그대로 숫자로 출력하며, 두 번째 예는 "Number:" 뒤에 정수를 출력한다. 세 번째 예에서는 "String:" 뒤에 문자열을 출력한다.

```
> sprintf("%d",123)
[1] "123"
> sprintf("Number :%d",123)
[1] "Number :123"
> sprintf("Number :%d, String :%s",123,"TT")
[1] "Number :123, String :TT"
```

%d 나 %nd 또는 %.mf 형태로 적을 수 있는데, 이때 n은 1의 자리 이상의 수를 몇 자리 까지 출력할 것인지를, m은 소수점 이하 자리의 수를 몇 자리 까지 출력할 것인지를 지정하는 데 사용한다.

다음은 주어진 실수를 소수점 첫째 자리까지만 출력하는 예이다.

```
> sprintf("%.1f",123.43)
[1] "123.4"
```

다음은 1의 자리 이상의 수를 5자리로 고정하는 예다. 소수점 이하를 지정한 경우와 달리 1의 자리 이상의 숫자가 5자리로 잘리지는 않으며, 다만 1의 자리 이상의 숫자를 5자리 이상으로 맞춰주는 역할만 한다.

```
> sprintf("%6d",1443)
[1] " 1443"
> sprintf("%5d",123)
[1] " 123"
> sprintf("%6d",1443234)
[1] "1443234"
```

이처럼 1의 자리 이상의 숫자 자릿수를 맞춰서 출력해주면 숫자가 주어진 자릿수에 맞춰 우측 정렬되므로 좀 더 보기 좋은 출력물을 얻을 수 있다.

## cat()

print()나 sprintf()는 결과를 출력한 뒤 개행이 일어난다. 반면 cat()은 주어진 입력을 출력하고 행을 바꾸지 않는다는 특징이 있다. 또한 cat()에는 여러 인자를 나열해 쓰면 해당 인자들이 계속 연결되어 출력된다는 특징이 있다. 따라서 프로그램 진행 상황을 보기 좋게 출력하는 데 종종 사용한다.

먼저 cat()이 행을 바꾸지 않는다는 점을 다음 코드를 통해 살펴보자.

```
> print("rr")
[1] "rr"
> cat("rrsdd12")
rrsdd12> cat(1,2,3,4)
1 2 3 4
```

보다시피 print는 개행이 됐고, cat은 그렇지 않았다. 첫번째 > cat("rrsdd12") 명령이후,

rrsdd12> cat(1,2,3,4) 이 바로 개행되지 않고 실행됐다. cat에서 개행하고 싶다면 "\n" 을 말미에 붙여야 한다.

```
cat(1,2,3,4,"\n")
1 2 3 4
>
```

cat()이 개행을 하지 않는다는 점을 이용하면 데이터 처리가 어떻게 수행 중인지를 다음과 같이 좀 더 보기 좋게 출력해줄 수 있다.

```
> sum_to_ten<- function(){
+   sum<- 0
+   cat("Adding...")
+   for(i in 1:10){
+     sum <- sum + i
+     cat(i,"...")
+   }
+   cat("Done!","\n")
+   return(sum)
+ }
> sum_to_ten()
Adding...1 ...2 ...3 ...4 ...5 ...6 ...7 ...8 ...9 ...10 ...Done!
[1] 55
```

## browser()

browser()가 호출되면 명령의 수행이 중지되고, 해당 시점부터 디버깅이 시작된다. 디버깅 모드가 되면 browser()가 호출된 행에서 접근 가능한 변수 및 함수의 내용을 볼 수 있으며, 코드를 한 행씩 실행하거나 다시 browser()가 호출될 때까지 명령의 실행을 재개할 수 있다.

```
> sum_to_ten<- function(){
+   sum<- 0
+   for(i in 1:10){
+     sum <- sum + i
+     if(i>=5){
+       browser()
+     }
+   }
+   return(sum)
+ }
```

이 코드를 실행하면 다음과 같이 "Browse"라는 새로운 프롬프트가 출력된다.

```
> sum_to_ten()
called from: sum_to_ten()
Browse[1]>
```

이 프롬프트는 i가 5인 상황에서 정지된 그 환경을 그대로 가지고 있다. 따라서 i나 sum의 값을 출력해보거나 일반적인 R 명령 프롬프트에서 하는 모든 작업을 해볼 수 있다. 다음은 browser()가 호출된 상태에서 i와 sum 값을 출력하고 i와 sum의 합을 구하는 예이다.

```

Browse[1]> i
[1] 5
Browse[1]> sum
[1] 15
Browse[1]> sum + i
[1] 20

```

## Browse 프롬프트 명령

명령	의미
c	continue, 다음 browser() 명령을 만날 때까지 코드의 수행을 재개
n	next, 현재 수행한 명령의 다음 명령을 수행
Q	Quit, browser()를 종료

현재 예에서는 i가 5이상일 때 매번 browser()가 호출되게 했으므로, i가 5일 때, 정지된 상태에서 c를 입력하면 i가 6일 때 다시 browser()로 진입한다.

```

Browse[1]> i
[1] 5
Browse[1]> sum
[1] 15
Browse[1]> sum + i
[1] 20
Browse[1]> c
Called from: sum_to_ten()
Browse[1]> i
[1] 6
Browse[1]> n
debug at #4: sum <- sum + i

```

n은 정지된 상태의 다음 문장을 차례로 수행하는 명령이다. n을 입력할 때마다 현재 수행 중인 문장을 출력하여 보여주므로 어느 코드가 수행되었는지를 알 수 있다.

```

Browse[2]> i
[1] 7
Browse[2]> n
debug at #5: if (i >= 5) {
  browser()
}
Browse[2]> i
[1] 7
Browse[2]> n
debug at #6: browser()
Browse[2]> i
[1] 7
Browse[2]> i
[1] 7
Browse[2]> n
Browse[2]> n
debug at #4: sum <- sum + i
Browse[2]> n
debug at #5: if (i >= 5) {

```

```
browser()
}
```

browser()의 수행을 마치려면 Q를 입력한다.

```
Browse[2]> Q
>
```

browser()는 함수의 사용법을 알아보는 데도 유용하다. 예를 들어, ply의 ddply()가 주어진 함수에 어떤 값들을 넘기고 있는지 확인하는 데도 쓸 수 있다. 다음은 ddply를 사용해 아이리스 데이터를 붓꽃의 종 별로 묶었을 때 지정한 함수에 어떤 행들이 넘어오는지를 확인하는 목적으로 browser()를 사용한 예다.

```
> library(plyr)
> ddply(iris, .(Species)
+       , function(rows){browser()})
Called from: .fun(piece, ...)
Browse[1]> head(rows)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1           3.5           1.4           0.2  setosa
2           4.9           3.0           1.4           0.2  setosa
3           4.7           3.2           1.3           0.2  setosa
4           4.6           3.1           1.5           0.2  setosa
5           5.0           3.6           1.4           0.2  setosa
6           5.4           3.9           1.7           0.4  setosa
```

지금까지 디버깅을 위한 여러 가지 방법을 살펴봤다. 그러나 코드 완결성을 위해서는 디버깅에 의존하기 보다는 testthat과 같은 유닛 테스트를 자주 또는 먼저 작성하여 좀 더 다양한 테스트를 수행하길 권한다. 테스트를 작성하면 사전에 예상하지 못한 다양한 입력에 대해 코드를 검증해볼 수 있으며, 코드를 수정할 때 새로운 버그가 생기지 않았음을 좀 더 쉽게 확인할 수 있기 때문이다.

## 5.9 코드 수행 시간 측정

코드가 생각보다 느리게 실행된다면 속도 향상에 앞서 어떤 부분에 시간이 오래 걸리는지를 조사할 필요가 있다. 이런 목적으로 코드 성능을 평가하는 방법에는 system.time()을 사용해 간단히 함수의 수행 시간을 출력해보는 방법과 Rprof() 함수를 사용해 좀 더 본격적인 보고서를 출력해보는 방법이 있다.

### 5.9.1 명령문 실행 시간 측정

system.time()은 인자로 주어진 명령이 수행하는 데 걸린 시간을 측정한다.

다음은 1부터 N까지 더하는 함수인 sum\_to\_n()의 수행 시간을 N=1,000,000, N = 10,000,000, N=100,000,000 인 경우에 대해 각각 측정해본 예다.

```
> sum_to_n <- function(n){
+   sum<- 0
+   for(i in 1:n){
+     sum <- sum + i
+   }
+   return(sum)
+ }

> system.time(sum_to_n(1,000,000))
사용자   시스템 elapsed
 0.01    0.00    0.01

> system.time(sum_to_n(10,000,000))
```



```

사용자  시스템 elapsed
0.24    0.00    0.23
> system.time(sum_to_n(100,000,000))
사용자  시스템 elapsed
2.50    0.03    2.55

```

system.time()이 출력하는 수행 시간은 user.time, system time, elapsed time 으로 구분된다. 이 중 elapsed time 이 가장 알기 쉬운 개념이다. 이 시간은 코드의 총 소요 시간으로, 코드를 시작한 직후부터 코드 수행이 끝날 때까지의 시간을 초시계로 잰 때 얼마나 걸렸는지를 나타낸다. 통상적으로 이야기하는 코드 수행시간이 이에 해당한다.

user time, system time은 elapsed time을 구성하는 요소로 각각 프로그램 코드 자체를 수행하는 데 걸린 시간과 프로그램이 운영체제의 명령을 호출했다면 그때 운영체제가 명령을 수행하는데 걸린 시간을 의미한다.

프로그램이 운영체제의 명령을 호출하는 대표적인 예로는 파일 입출력이 있다.

```

> system.time(save(x, file = "x.RData"))
사용자  시스템 elapsed
53.02    0.33    53.36
> system.time(load(file="x.RData"))
사용자  시스템 elapsed
2.09    0.14    2.25

```

위 결과를 보면 1부터 N까지의 숫자를 합하는 함수의 경우와 달리 system time이 크게 나타난 것을 볼 수 있다. 이는 파일 입출력을 위해 R 환경이 운영체제의 기능을 많이 호출해서 사용하기 때문에 발생한 현상이다.

system.time은 코드를 수행하는 머신의 메모리가 부족해 운영체제가 디스크의 일부를 메모리로 사용하는 페이징이 빈번히 발생할 경우에도 증가할 수 있다. 따라서 system time이 크다면 메모리가 혹시 부족하지는 않는지 운영체제의 작업 관리자를 통해 잘 살펴봐야 한다.

이처럼 코드의 수행 시간을 측정할 수 있으면 다양한 구현 방법 중 어떤 방법이 더 나은 성능을 보이는지 평가할 수 있다. 예를 들어 앞 절에서 rbind()보다 rbindlist()가 성능이 더 나음을 system.time() 함수를 사용해 보인 바 있다.

## 코드 프로파일링

Rprof()는 좀 더 본격적인 코드 수행 성능 평가를 위한 함수다. prof는 Profiling을 의미하는데 소프트웨어 공학에서는 프로그램의 동적인 성능, 즉 메모리나 CPU 사용량을 평가하는 작업을 흔히 코드 프로파일링이라 부른다.

코드 프로파일링은 Rprof() 함수 호출로 시작한다. Rprof() 함수 호출 이후에는 특정 시간 간격마다 현재 어떤 함수가 수행 중인지 샘플이 추출되어 파일에 저장된다. 파일에 기록된 결과는 이후 summaryRprof() 함수로 분석할 수 있다. 다음 표에 이들 함수에 대해 정리했다.

다음은 주어진 벡터에 1을 더하는 비효율적인 함수에 대해 코드 프로파일링을 수행하는 예를 보여준다. 코드에서 seq\_along()은 인자로 주어진 벡터의 길이만큼 1,2,3,...,N으로 구성된 숫자 벡터를 생성하는 함수다.

```

Rprof("add_one.out")
x<-add_one_to_vec(1:1000000)
head(x)
Rprof(NULL)

```

코드를 실행하면 add\_one.out이라는 파일이 생성된다. 결과를 분석하려면 Rprof()가 생성한 파일을 summaryRprof()에 넘겨주면 된다.

```
> summaryRprof("add_one.out")
$by.self
      self.time self.pct total.time total.pct
"add_one"      0.26   28.26      0.26   28.26
"add_one_to_vec" 0.24   26.09      0.52   56.52

$by.total
      total.time total.pct self.time self.pct
"add_one_to_vec" 0.52   56.52      0.24   26.09
"add_one"        0.26   28.26      0.26   28.26

$sample.interval
[1] 0.02

$sampling.time
[1] 0.92
```

summaryRprof()의 분석 결과는 크게 by.self와 by.total 두 섹션으로 나뉘는데, 이 두 섹션에 들어 있는 내용은 동일하다. 다만 by.self는 self.time으로 정렬된 표이고, by.total은 total.time으로 정렬된 표다. 따라서 설명의 편의를 위해서는 by.self를 기준으로 살펴보기로 한다.

by.self에서 self.time은 각 함수가 수행하는 데 걸린 시간이다. 코드 전체 수행 시간 대비 각 함수의 self.time 시간의 비율은 self.pct를 통해 알 수 있다.

total.time은 각 함수 내 코드를 수행하는 데 걸린 시간과 해당 함수가 호출한 함수를 수행하는 데 걸린 시간의 합이다. 예를 들어, "add\_one\_to\_vec"가 걸린 시간을 비교해보자. add\_one\_to\_vec의 total.time은 0.52로, 이는 add\_one이 total.time(0.26) + self.time(0.26)을 더한 것에 대한 결과이다.

이처럼 summaryRprof()의 결과는 어떤 함수가 수행하는데 얼마나 시간이 소모되는지를 조금 더 쉽고 분석적으로 나열해주므로, 어떤 함수가 가장 긴시간이 소모되는지를 판단할 수 있다. 만약 R코드의 성능이 만족스럽지 못하다면 Rprof()을 사용해 소모 시간과 그 비율을 판단하고 가장 긴 시간이 걸리는 항목부터 개선해나감으로써 더 과학적인 성능 개선을 이룰 수 있다.

Rprof()에는 여기에서 설명하지 않은 메모리 프로파일링 등의 다양한 옵션이 있으므로 코드 성능 평가 시 help(Rprof)를 살펴보기 바란다.