Brief Introduction to R

どんぐり研究所　孫 建強

Brief Introduction to R

Contens

データ型

- ベクトル
- 行列
- データフレーム
- リスト
- 文字列

データ型

- ベクトル
- 行列
- データフレーム
- リスト
- 文字列

変数

変数は、値を保存するための箱のようなものである。変数に値を保持させるためには、その変数に値を付与する必要がある。この付与作業を代入という。R では、変数（オブジェクト）に値を代入するとき、代入演算子 "<-" を使う。代入演算子 "<-" は、右辺の値を、左辺の変数（オブジェクト）に代入する機能を持つ。

右のコードは、1 という値を、a という名前の変数（オブジェクト）に代入することを表している。このコードを実行することによって、プログラムが終了するまで、a は 1 を保持している状態になる。

```
a <- 1
```

※数値（整数、小数）以外に、複素数や文字、文字列を代入することもできる。

変数

代入演算子 "<-" の右辺が計算式の場合は、計算式ではなく、その計算結果が左辺に代入される。また、すでに値を保持している変数に、他の値を代入すると、既存の値が上書きされる。

```
a <- 1
b <- 1

c <- a + b
c
# [1] 2

d <- c - 2
d
# [1] 0

e <- a + 2
e
# [1] 3

a <- d + e
a
# [1] 3
```

標準出力

オブジェクトに代入された値は、コンピューターのメモリ上に保存される。出力命令を出さない限り、ディスプレイ（標準出力）に出力されない。出力命令をだすとき、print 関数を使用する。

※この資料では、これ以降 print 関数を省略する。

```
a <- 1
b <- 1

c <- a + b
print(c)
# [1] 2

d <- c - 2
print(d)
# [1] 0

e <- a + 2
print(e)
# [1] 3

a <- d + e
print(a)
# [1] 3
```

算術演算子

R で四則演算を行うのに次のような演算子を使用する。

割り算に関して、余と商を求める演算子もある。

演算子	意味	例
+	加	7 + 4 = 11
-	減	7 - 4 = 3
*	乗	7 * 4 = 28
/	除	7 / 4 = 1.75
%%	余	7 %% 4 = 3
%/%	商	7 %/% 4 = 1
^	累乗	2 ^ 10 = 1024

※小数に対しても余と商を求めることができる。a%/%b は、a を b で割った後に整数部分を返す。また、a%%b は、a – a%/%b*b で計算された結果が返される。

```
a <- 11
b <- 3

a + b
# [1] 14

a - b
# [1] 8

a * b
# [1] 33

a / b
# [1] 3.666667

a %% b
# [1] 2

a %/% b
# [1] 3
```

ベクトル

複数の値をオブジェクトに代入する場合は、複数の値を c 関数で束ねてから代入する。

```
a <- c(1, 2, 3)
```

```
a <- c(2, 4, 6)
b <- c(1, 3, 5)
c <- c(1, 1, 2, 3)
```

ベクトル

ベクトル同士の計算も数値と同様に行える。ただし、計算対象のベクトルの長さが同じでないときも、エラーなく計算できるため、十分に注意すること。

```
a <- c(2, 4, 6)
b <- c(1, 3, 5)
c <- c(1, 1, 2, 3)

x <- a + b
x
# [1]  3  7 11

y <- a - 4
y
# [1] -2  0  2

z <- y + 2
z
# [1] 0 2 4

w <- b + c
w
# [1] 2 4 7 4
```

計算可能だが警告文が出る。 ▶

Warning message: In b + c : longer object length is not a multiple of shorter object length

ベクトル

c 関数の他に、等差数列や全要素が同じ値であるような特殊なベクトルを作成するには、次のように、seq 関数や rep 関数などを使用する。

```
a <- 2:8
a
# [1] 2 3 4 5 6 7 8

a <- 8:2
a
# [1] 8 7 6 5 4 3 2

b <- seq(2, 8)
b
# [1] 2 3 4 5 6 7 8

e <- seq(2, 9, 3)
e
# [1] 2 5 8
```

```
f <- rep(1, 6)
f
# [1] 1 1 1 1 1 1

g <- c(1, 2, 3)
h <- rep(g, times = 2)
h
# [1] 1 2 3 1 2 3

i <- c(1, 2, 3)
j <- rep(i, each = 2)
j
# [1] 1 1 2 2 3 3
```

統計

R は統計解析向けに開発されたプログラミング言語ないし実行環境である。そのため、R には、統計計算に関連した関数が多く実装されている。

関数	機能
mean	平均値
sd	不偏標準偏差
var	不偏分散
sum	総和
range	範囲
quantile	分位数
max	最大値
min	最小値
scale	標準化（平均0 分散1）
table	頻度

```
x <- c(1, 6, 5, 3, 2, 6, 2, 3, 8, 9, 7)

mean(x)

sd(x)

var(x)

sum(x)

range(x)

quantile(x)

max(x)

min(x)

scale(x)

table(x)
```

集合演算

R のベクトルに対して、重複要素を取り除いたり、あるいは 2 つのベクトルを比較して共通要素を取り出したりするような集合演算が行える。

演算子	動作
unique(a)	集合 a から重複要素を除去
union(a, b)	集合 a と集合 b の和集合
intersect(a, b)	集合 a と集合 b の積集合
setdiff(a, b)	集合 a と集合 b の差集合
a %in% b	集合 a の各要素が集合 b に含まれているかどうかを判定

```
a <- c(4, 8, 12, 16, 20, 24, 28, 32)
b <- c(3, 6,  9, 12, 15, 18, 21, 24)


x <- intersect(a, b)
x
# [1] 12 24


x <- setdiff(a, b)
x
# [4, 8, 16, 20, 28, 32]


x <- (a %in% b)
x
# [F, F, T, F, F, T, F, F]
```

問題 01-1

3 min

ベクトル a の最大値と最小値の差を計算せよ。

```
a <- c(6, 7, 8, 5, 2, 3, 9)
```

問題 01-2

3 min

ベクトル a とベクトル b の共通要素の和を求めよ。

```
a <- c(1, 3, 5, 7, 9, 11, 13, 15, 17)
b <- c(1, 1, 2, 3, 5, 8, 13, 21, 34)
```

ベクトル要素参照

ベクトルは基本的に全要素を束ねて使うことが多いが、一部の要素だけを取り出したり、変更したりすることができる。その際、ベクトルの先頭から数えて何番目の要素を取り出したいのかを、整数で指定する必要がある。ただし、R では 1 番から数え始めることに注意。

```
w <- c(12, 10, 11, 13, 11)

w[1]

w[2]

w[6]
```

ベクトル要素参照

ベクトルは基本的に全要素を束ねて使うことが多いが、一部の要素だけを取り出したり、変更したりすることができる。その際、ベクトルの先頭から数えて何番目の要素を取り出したいのかを、整数で指定する必要がある。ただし、R では 1 番から数え始めることに注意。

```
w <- c(12, 10, 11, 13, 11)


w[1]
# [1] 12

w[2]
# [1] 10

w[6]
# [1] NA
```

該当の要素がないため NA となる。

ベクトル要素参照

ベクトルは基本的に全要素を束ねて使うことが多いが、一部の要素だけを取り出したり、変更したりすることができる。その際、ベクトルの先頭から数えて何番目の要素を取り出したいのかを、整数で指定する必要がある。ただし、R では 1 番から数え始めることに注意。

```
w <- c(12, 10, 11, 13, 11)


w[1]
# [1] 12

w[2]
# [1] 10

w[6]
# [1] NA


w[2] <- 9

w
```

ベクトル要素参照

ベクトルは基本的に全要素を束ねて使うことが多いが、一部の要素だけを取り出したり、変更したりすることができる。その際、ベクトルの先頭から数えて何番目の要素を取り出したいのかを、整数で指定する必要がある。ただし、R では 1 番から数え始めることに注意。

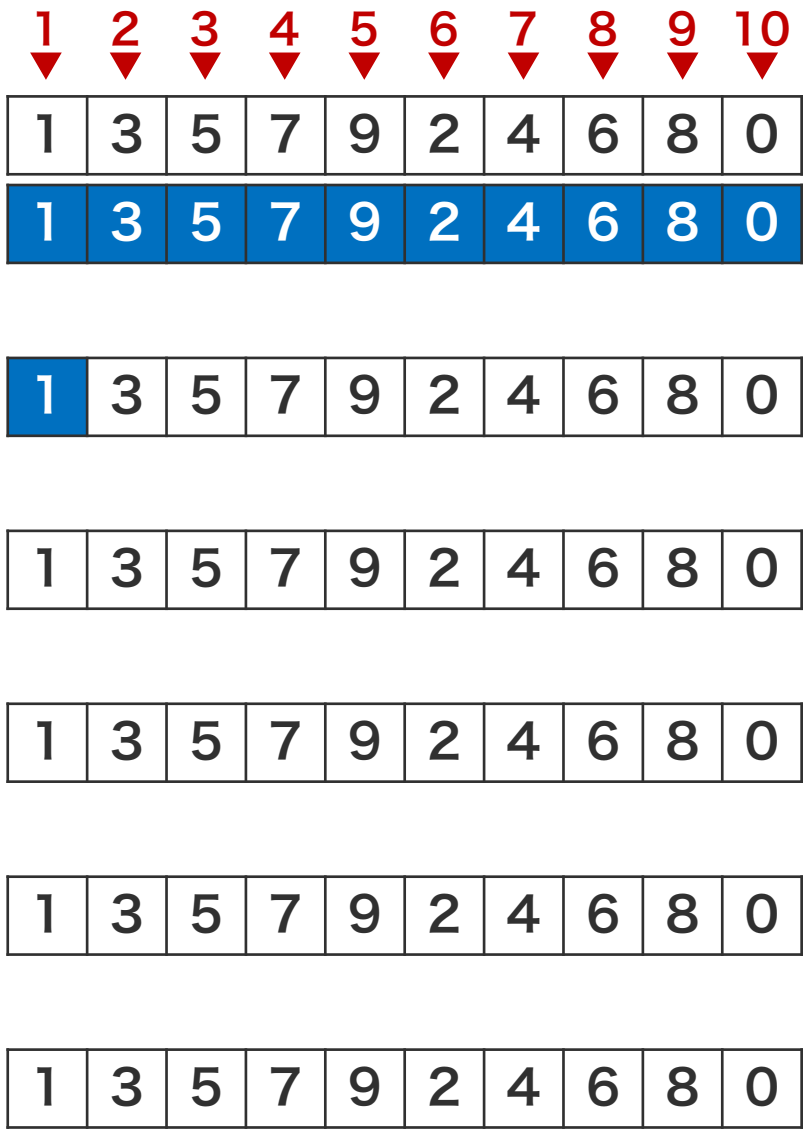
```
w <- c(12, 10, 11, 13, 11)



w[1]
# [1] 12

w[2]
# [1] 10

w[6]
# [1] NA


w[2] <- 9

w
# [1] 12  9 11 13 11
```

ベクトル要素参照

ベクトルから要素を取り出す際に、複数の位置番号を指定することで、複数の値を一括に取り出すことができる。

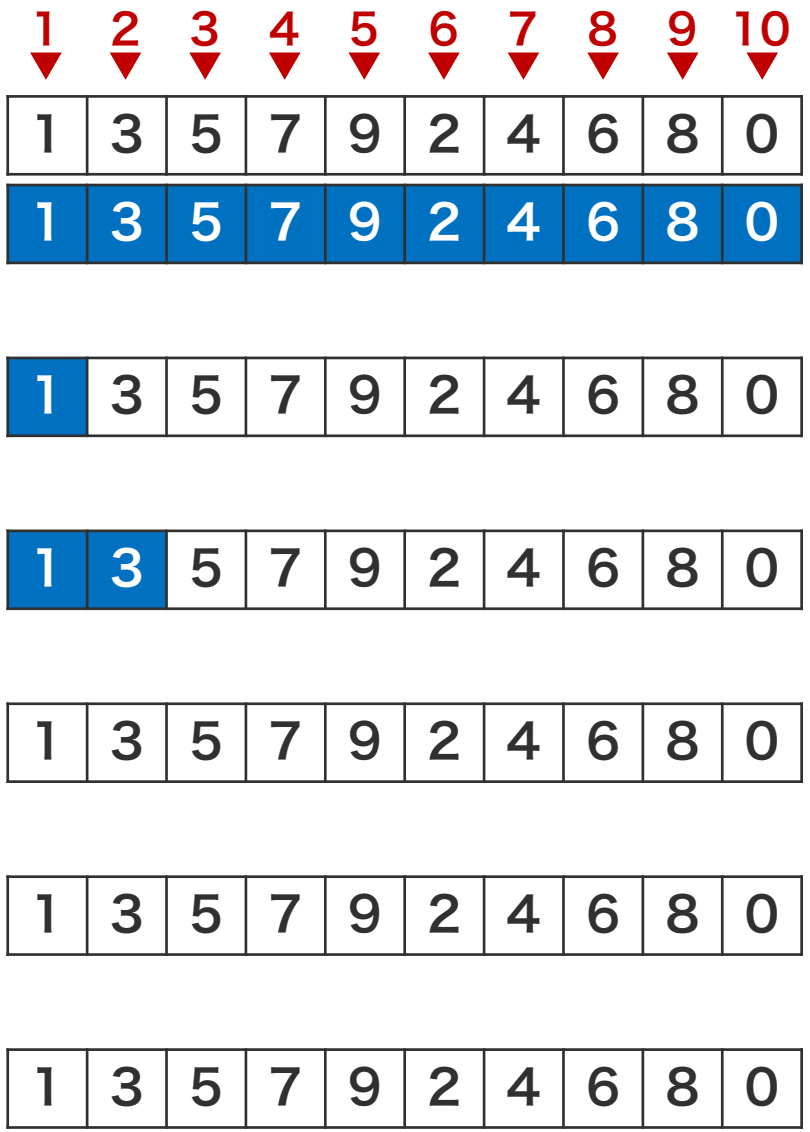
```
a <- c(1, 3, 5, 7, 9, 2, 4, 6, 8, 0)


a
# [1] 1 3 5 7 9 2 4 6 8 0

a[1]


a[1:2]


a[2:6]


a[1:4]


a[5:length(a)]
```

ベクトル要素参照



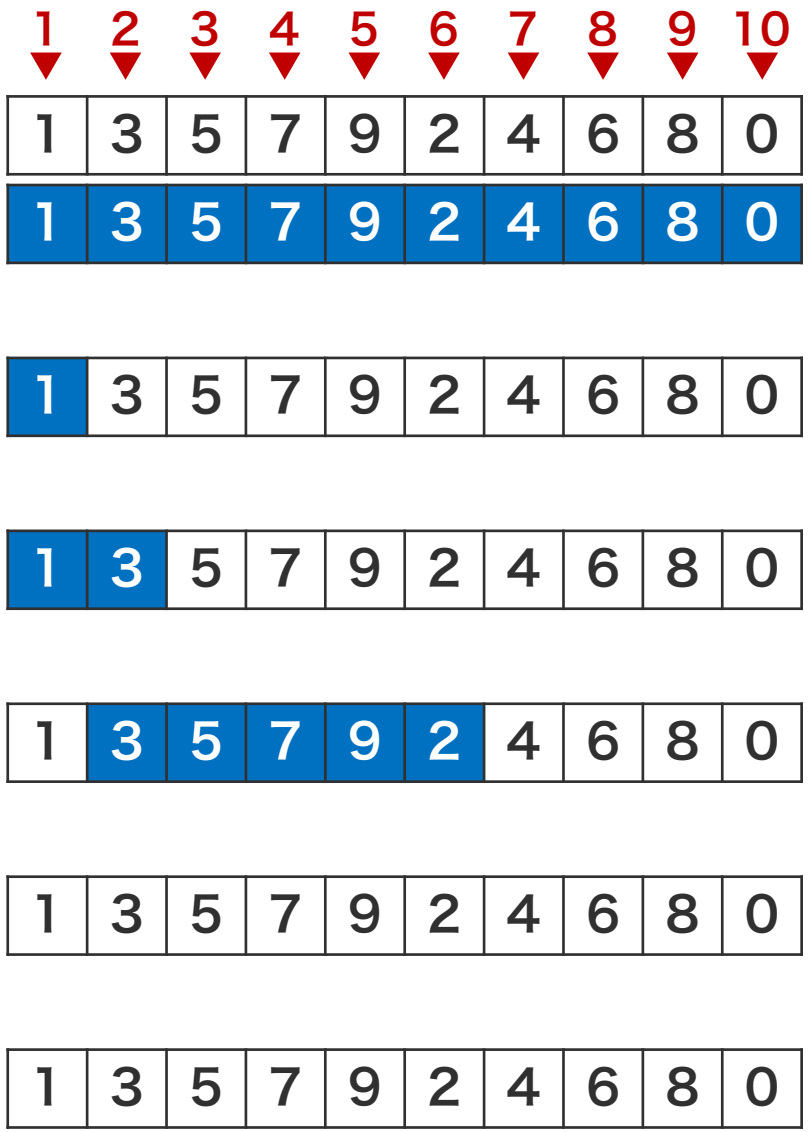

```
a <- c(1, 3, 5, 7, 9, 2, 4, 6, 8, 0)


a
# [1] 1 3 5 7 9 2 4 6 8 0

a[1]
# [1] 1

a[1:2]


a[2:6]


a[1:4]


a[5:length(a)]
```

ベクトル要素参照



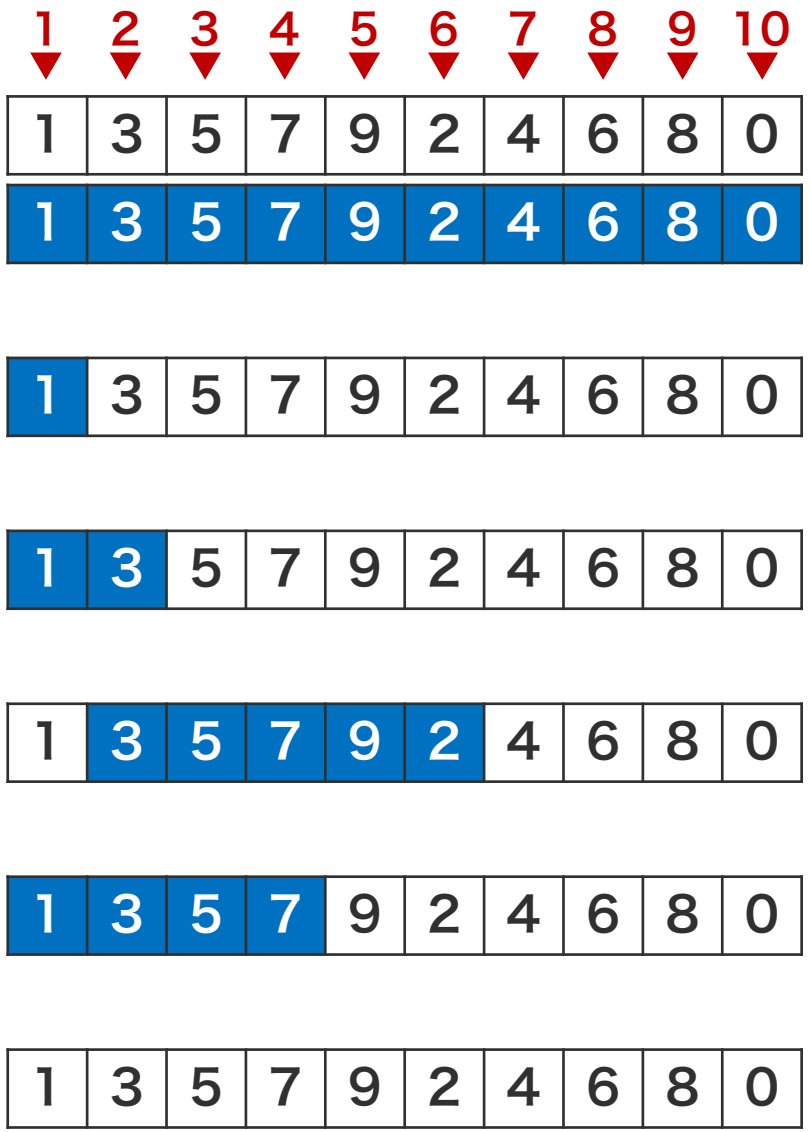

```
a <- c(1, 3, 5, 7, 9, 2, 4, 6, 8, 0)


a
# [1] 1 3 5 7 9 2 4 6 8 0

a[1]
# [1] 1

a[1:2]
# [1] 1 3

a[2:6]


a[1:4]


a[5:length(a)]
```

ベクトル要素参照



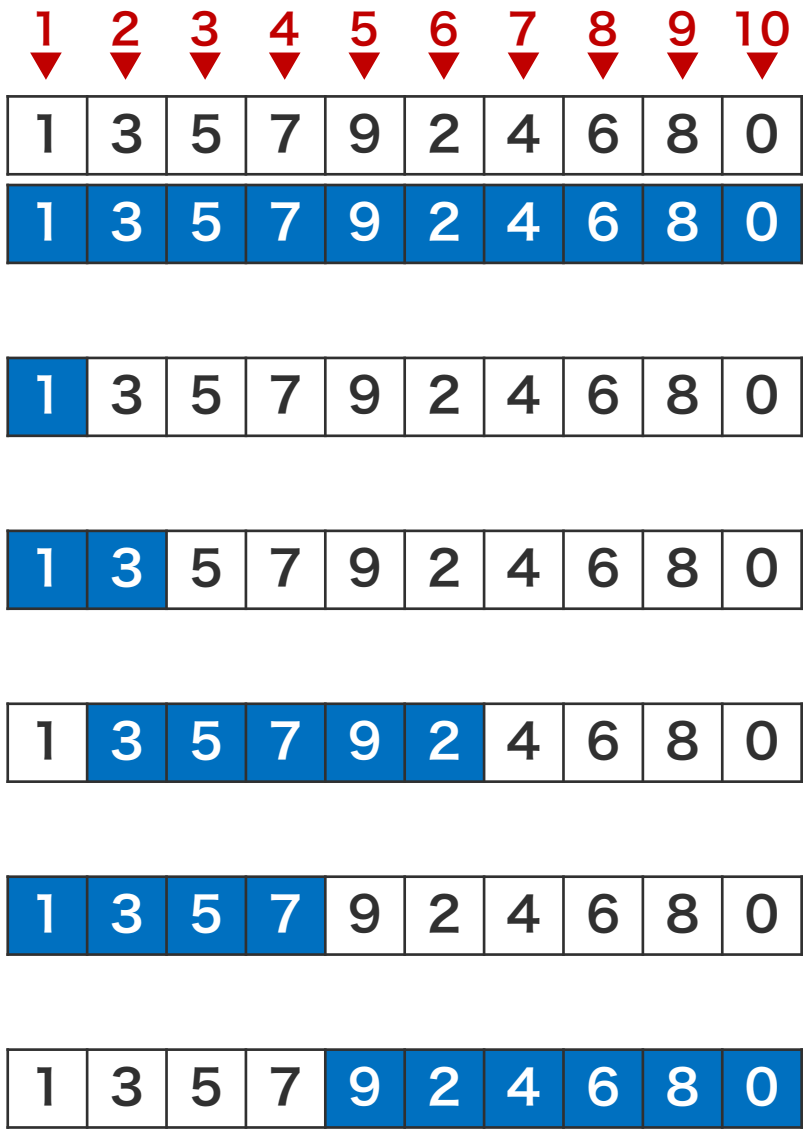

```
a <- c(1, 3, 5, 7, 9, 2, 4, 6, 8, 0)


a
# [1] 1 3 5 7 9 2 4 6 8 0

a[1]
# [1] 1

a[1:2]
# [1] 1 3

a[2:6]
# [1] 3 5 7 9 2

a[1:4]


a[5:length(a)]
```

ベクトル要素参照




```
a <- c(1, 3, 5, 7, 9, 2, 4, 6, 8, 0)


a
# [1] 1 3 5 7 9 2 4 6 8 0

a[1]
# [1] 1

a[1:2]
# [1] 1 3

a[2:6]
# [1] 3 5 7 9 2

a[1:4]
# [1] 1 3 5 7

a[5:length(a)]
```

ベクトル要素参照




```
a <- c(1, 3, 5, 7, 9, 2, 4, 6, 8, 0)


a
# [1] 1 3 5 7 9 2 4 6 8 0

a[1]
# [1] 1

a[1:2]
# [1] 1 3

a[2:6]
# [1] 3 5 7 9 2

a[1:4]
# [1] 1 3 5 7

a[5:length(a)]
# [1] 9 2 4 6 8 0
```

ベクトル要素参照

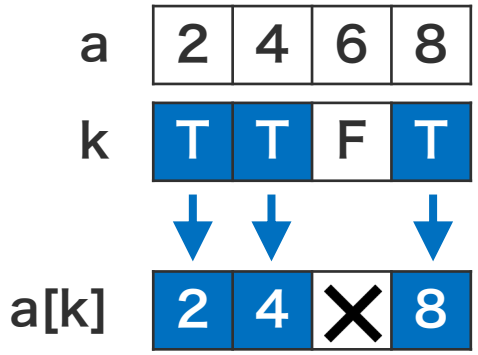
ベクトルから要素を取り出すときに位置番号を指定する方法の他に、TRUE または FALSE からなるベクトルで指定することもできる。このとき、TRUE/FALSE からなるベクトルの長さは、操作対象となるベクトルの長さと同じでなければならない。

```
a <- c(   2,     4,      6,     8)
k <- c(TRUE, TRUE, FALSE, TRUE)

a[k]
```

```
a <- c(   2,     4,     6,     8)
k <- c(FALSE, TRUE, TRUE, TRUE)

a[k]
```

ベクトル要素参照

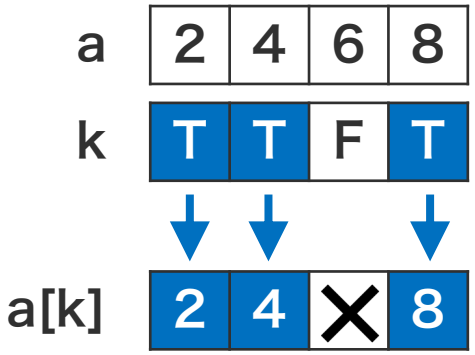
フィルター k を用いて、ベクトル a の要素をフィルタリングしているイメージ。



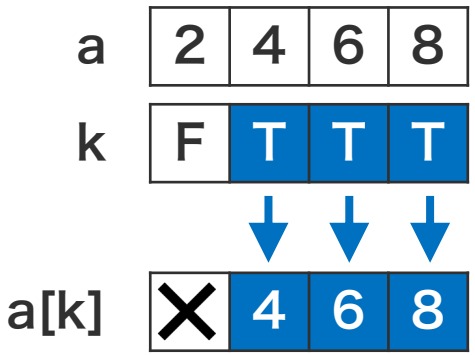

```
a <- c(   2,     4,      6,     8)
k <- c(TRUE, TRUE, FALSE, TRUE)

a[k]
# [1] 2 4 8
```

```
a <- c(   2,     4,     6,     8)
k <- c(FALSE, TRUE, TRUE, TRUE)

a[k]
```

ベクトル要素参照




```
a <- c(   2,     4,      6,     8)
k <- c(TRUE, TRUE, FALSE, TRUE)

a[k]
# [1] 2 4 8
```




```
a <- c(   2,     4,     6,     8)
k <- c(FALSE, TRUE, TRUE, TRUE)

a[k]
# [1] 4 6 8
```

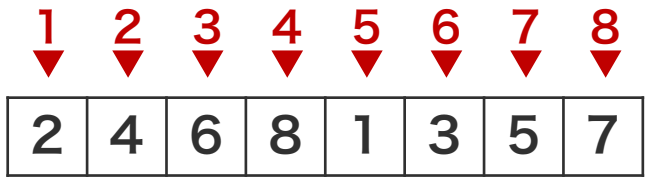
フィルター k を用いて、ベクトル a の要素をフィルタリングしているイメージ。

ベクトル要素参照

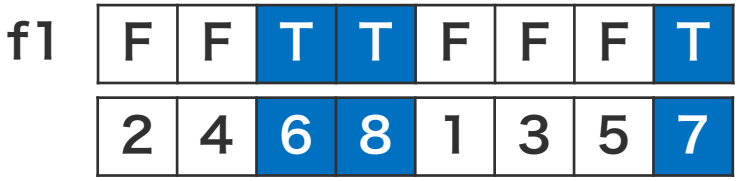
フィルターは、TRUE または FALSE を組み合わせて直打ちで作成できるが、ある条件を制定して、その条件に基づいて作成することが一般的である。例えば、5 よりも大きい値を取り出すフィルターや奇数の値を取り出すフィルターは、右のように作成する。

```
a <- c (2, 4, 6, 8, 1, 3, 5, 7)



f1 <- (a > 5)
f1
# [1] F F T T F F F T


f2 <- (a %% 2 == 1)
f2
# [1] F F F F T T T T
```

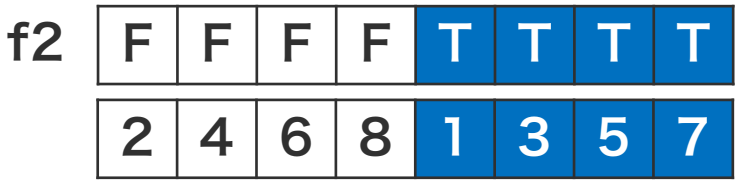
ベクトル要素参照




```
a <- c (2, 4, 6, 8, 1, 3, 5, 7)
```

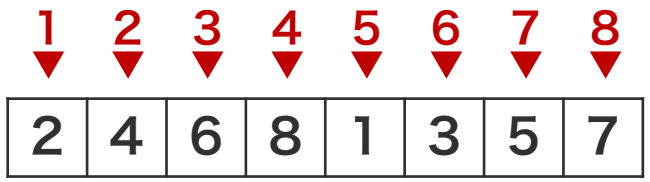



```
f1 <- (a > 5)
a[f1]
# [1] 6 8 7
```

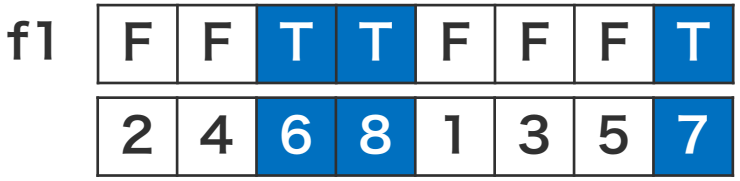



```
f2 <- (a %% 2 == 1)
a[f2]
# [1] 1 3 5 7
```

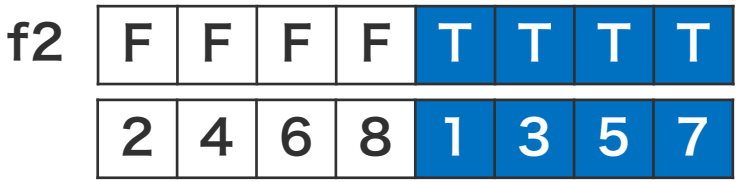
ベクトル要素参照



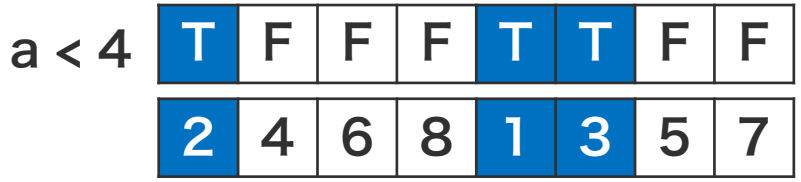

```
a <- c (2, 4, 6, 8, 1, 3, 5, 7)
```




```
f1 <- (a > 5)
a[f1]
# [1] 6 8 7
```




```
f2 <- (a %% 2 == 1)
a[f2]
# [1] 1 3 5 7
```




```
a[(a < 4)]
# [1] 2 1 3
```

フィルターを一次変数に保存せずに、直接使用することもできる。

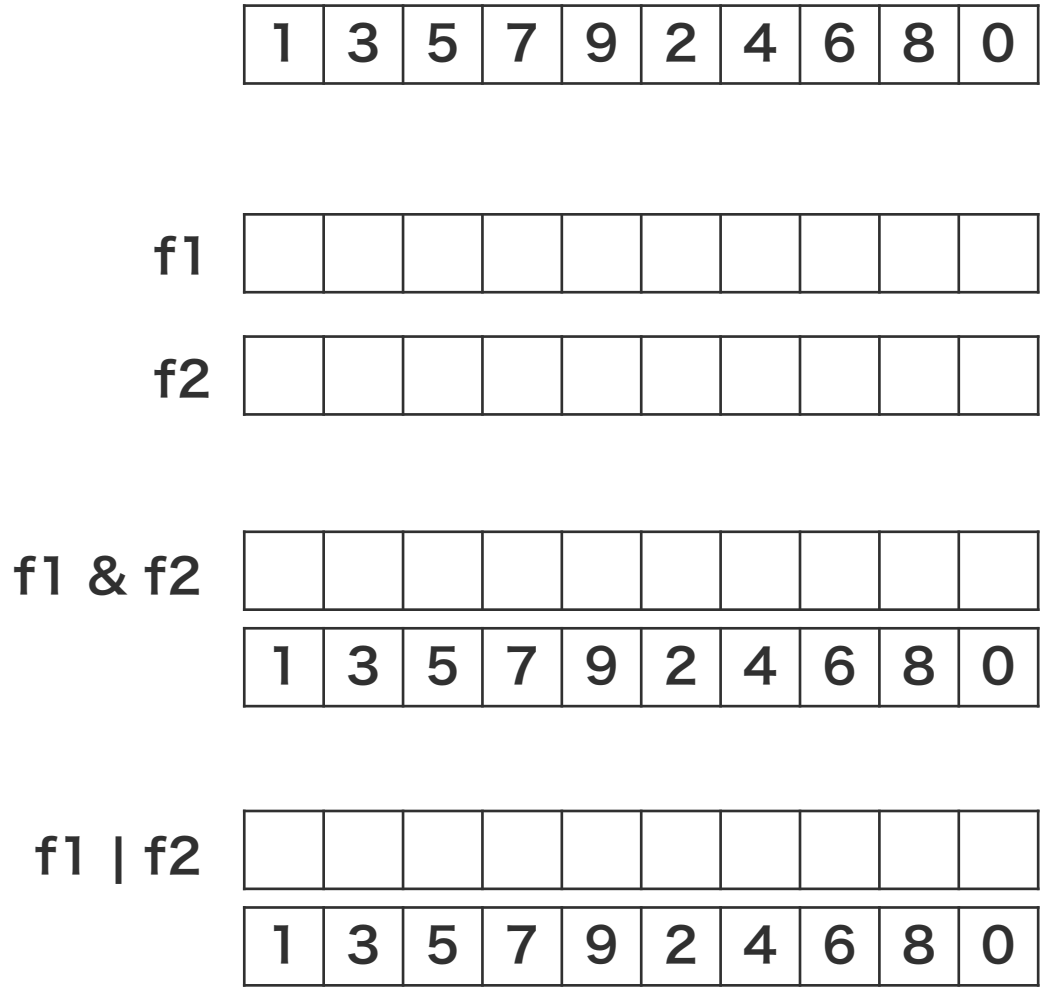
ベクトル要素参照

複数のフィルターを重ねて使用することもできる。この場合、フィルターを重ねるときに AND 演算（&）で重ねるか、OR 演算（|）で重ねるかを指定する必要がある。

演算子	意味	例
&	ベクトル論理積	(a<6) & (a%2==0)
|	ベクトル論理和	(a<6) | (a%2==0)

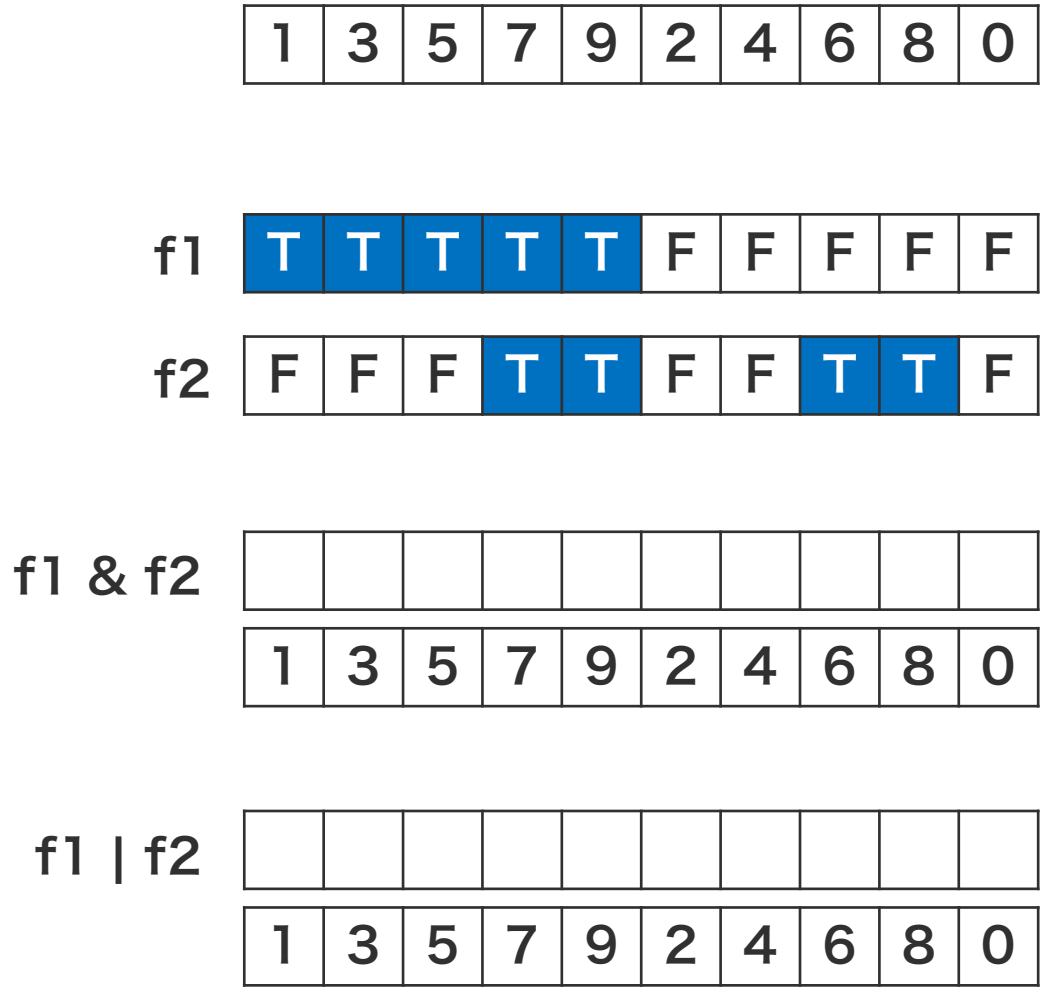
```
a <- c(1, 3, 5, 7, 9, 2, 4, 6, 8, 0)



f1 <- (a %% 2 == 1)

f2 <- (a > 5)

a[f1 & f2]



a[f1 | f2]
```

ベクトル要素参照



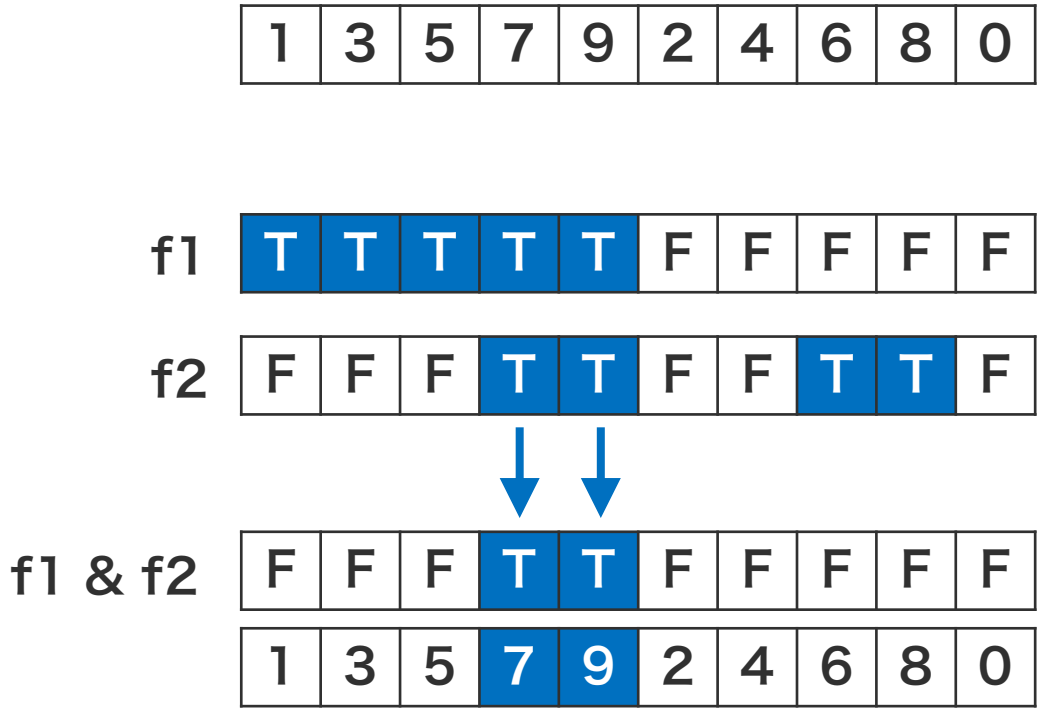

```
a <- c(1, 3, 5, 7, 9, 2, 4, 6, 8, 0)

f1 <- (a %% 2 == 1)

f2 <- (a > 5)

a[f1 & f2]

a[f1 | f2]
```

ベクトル要素参照



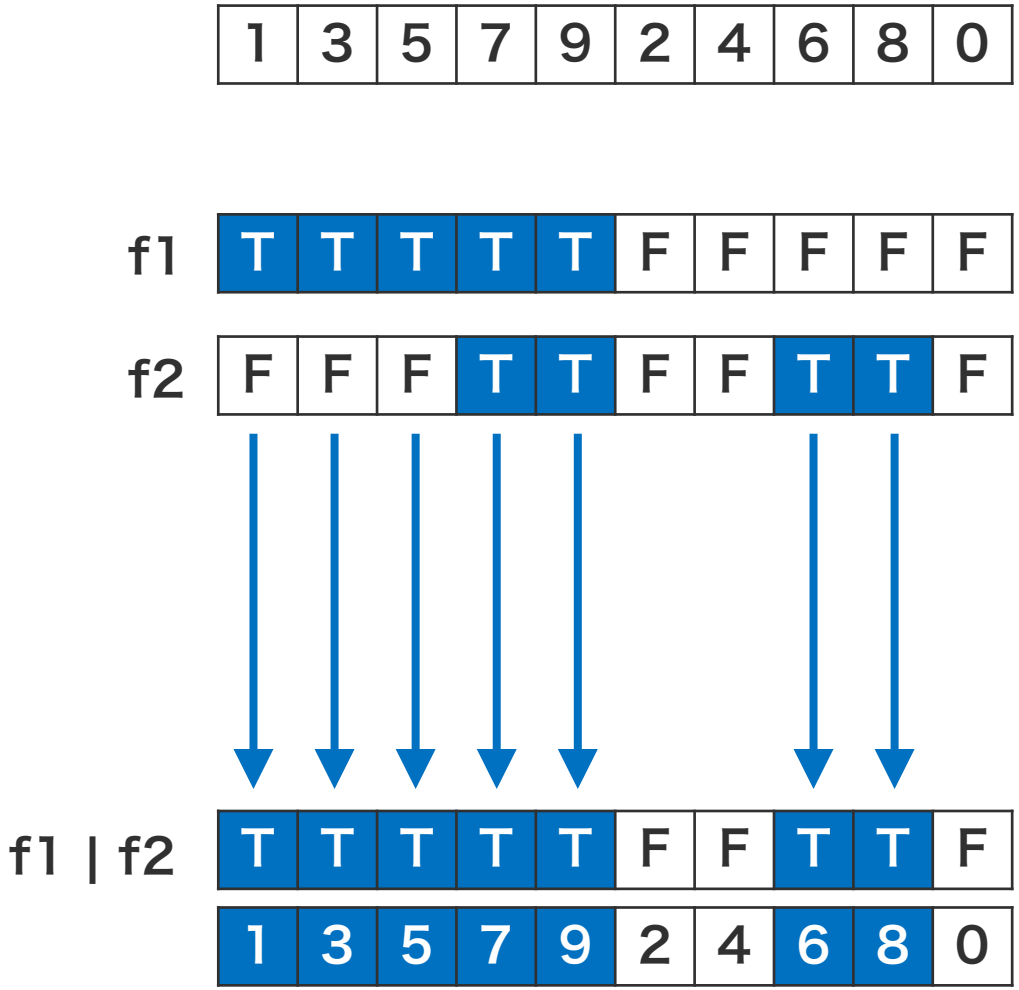

```
a <- c(1, 3, 5, 7, 9, 2, 4, 6, 8, 0)

f1 <- (a %% 2 == 1)

f2 <- (a > 5)

a[f1 & f2]

a[f1 | f2]
```

ベクトル要素参照




```
a <- c(1, 3, 5, 7, 9, 2, 4, 6, 8, 0)



f1 <- (a %% 2 == 1)

f2 <- (a > 5)


a[f1 & f2]
# [1] 7 9


a[f1 | f2]
```

ベクトル要素参照




```
a <- c(1, 3, 5, 7, 9, 2, 4, 6, 8, 0)
```

```
f1 <- (a %% 2 == 1)
```

```
f2 <- (a > 5)
```

```
a[f1 & f2]
# [1] 7 9
```

```
a[f1 | f2]
# [1] 1 3 5 7 9 6 8
```

問題 01-3

3 min

奇数を取り出すフィルターを作成して、奇数要素を取り出しなさい

```
a <- c(6, 7, 8, 5, 2, 3, 9)
```

問題 01-4

3 min

4 よりも大きい奇数を取り出すフィルターを作成して、

奇数要素を取り出しなさい

```
a <- c(6, 7, 8, 5, 2, 3, 9)
```

any 関数 / all 関数

TRUE/FALSE からなるあるベクトルに対して、そのベクトルの要素のすべてが TRUE なのか、それとも一部だけが TRUE なのかを調べることもできる。

関数	機能
`all(a)`	ベクトル a の要素すべてが TRUE ならば TRUE を返す
`any(a)`	ベクトル a の要素のうち少なくとも TRUE が 1 つ以上あれば TRUE を返す
`table(a)`	ベクトル a の要素のうち、TRUE と FALSE の数を返す

table 関数は頻度を調べる関数である。TRUE/FALSE からなるベクトルを代入することで、TRUE および FALSE の出現回数が調べられる。

```
a <- c(T, T, T, T, T)

all(a)
# [1] TRUE

any(a)
# [1] TRUE


b <- c(T, T, F, T, T)

all(b)
# [1] FALSE

any(b)
# [1] TRUE

table(b)
# FALSE  TRUE
#     1     4
```

which 関数

TRUE/FALSE からなるベクトルが与えられたとき、

TRUE の位置番号を調べるには which 関数を使う。

```
a <- c(T, T, F, T, F)
a
# [1] T T F T F

p <- which(a)
p
# [1] 1 2 4

b <- c(8, 4, 5, 3, 6)
e <- (b > 5)
e
# [1] T F F F T

q <- which(e)
q
# [1] 1 5

which(b > 5)
# [1] 1 5
```

which 関数

TRUE/FALSE からなるベクトルが与えられたとき、TRUE の位置番号を調べるには which 関数を使う。which 関数を使うことである条件を満たす特定の位置の値を、他の値に書き換えることができる。

```
a <- c(8, 4, 5, 3, 6)
b <- c(8, 4, 5, 3, 6)
```

which 関数で a > 5 を満たす位置の番号を調べて、その位置に 0 を代入する。 ▶

```
f <- (a > 5)
a[which(f)] <- 0
a
# [1] 0 4 5 3 0
```

「b > 5」となるフィルターを用いて要素 b をフィルタリングして、フィルタリングされた結果に 0 を代入する。 ▶

```
f <- (b > 5)
b[f] <- 0
b
# [1] 0 4 5 3 0
```

ifelse 関数

あるベクトルに対して条件判定を行い、その判定結果に応じて新たなベクトルを作成するときは ifelse 関数を利用する。

```
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)


# y <- (x %% 2 == 0)
y <- ifelse(x %% 2 == 0, TRUE, FALSE)
y
# [1] F T F T F T F T F


y <- ifelse(x %% 2 == 0, x, -x)
y
# [1] -1  2 -3  4 -5  6 -7  8 -9


y <- ifelse(x %% 2 == 0, 1, 0)
y
# [1] 0 1 0 1 0 1 0 1 0
```

問題 01-5

5 min

ベクトル x の要素のうち、奇数要素の個数を求めて n に代入し、奇数要素の総和を計算して s に代入せよ。

```
x <- c(3, 5, 3, 4, 8, 9)
```

```
n
# 4
s
# 20
```

```
x <- c(4, 3, 2, 8, 7)
```

```
n
# 2
s
# 10
```

問題 01-6

10 min

xy 平面上に、-1 ≤ x ≤ 1 および -1 ≤ y ≤ 1 の区間に 100 万個の点をランダムにばらまいたとき、単位円内側に含まれる点の数を求めよ。ただし、一様分布から乱数を生成する関数として runif 関数を用いてもよい。

```
x <- runif(1000000, min = -1, max = 1)
```

非数値・欠損値

R の中で数値でない要素を表すために、特殊な文字（NULL, NA, NaN, Inf）が割り当てられている。これらの特殊文字を調べるには is 関数を使う。

文字	意味
NULL	プログラミング言語の中で「何も示さない」を表すために使われる。
NA	欠損値。
NaN	非数値。0/0 の計算で NaN となる。
Inf	無限。log(0) や 1/0 などの計算で Inf あるいは -Inf となる。

```
x <- c(2, 5, NA, 3, 0/0, 2, log(0))
x
# [1]   2   5   NA   3   NaN   2   -Inf


is.null(x)
# [1] F


is.na(x)
# [1] F F T F T F F


is.nan(x)
# [1] F F F F T F F


is.infinite(x)
# [1] F F F F F F T
```

問題 01-7

5 min

ベクトル x から欠損値を取り除き、残った値を変数 y に代入せよ。

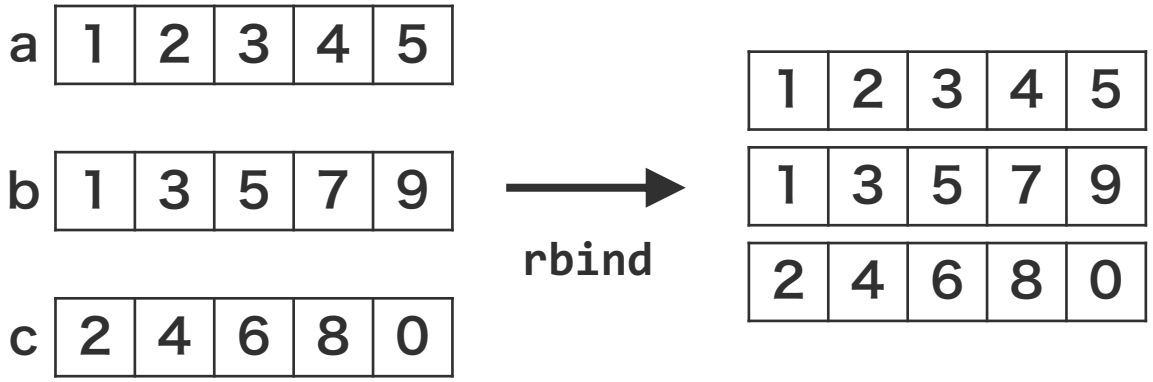
```
x <- c(1, NA, 3, 4, 9, 5, NA, 6)
```

データ型

- ベクトル
- 行列
- データフレーム
- リスト
- 文字列

行列

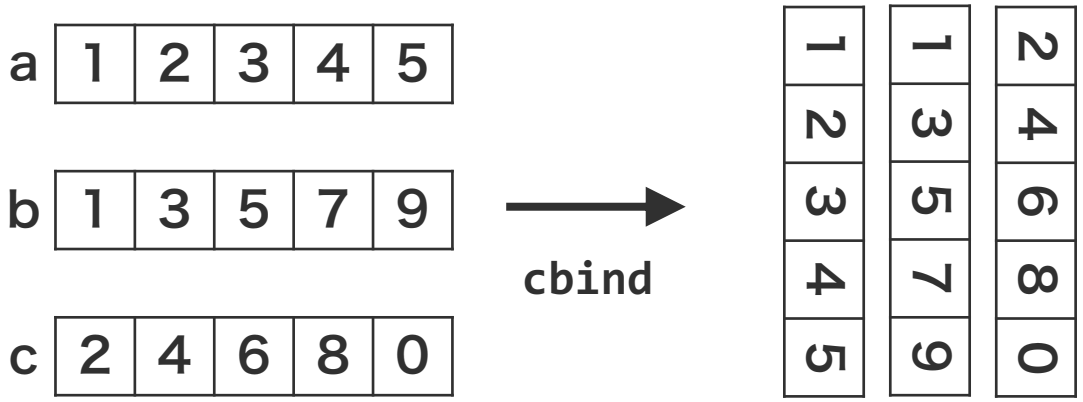
同じ長さのベクトルを複数束ねてできるオブジェクトを行列という。複数のベクトルを行方向に束ねるときは rbind 関数を使用し、列方向に束ねるときは cbind 関数を使用する。




```
a <- c(1, 2, 3, 4, 5)
b <- c(1, 3, 5, 7, 9)
c <- c(2, 4, 6, 8, 0)

n <- rbind(a, b, c)
n
#   [,1] [,2] [,3] [,4] [,5]
# a    1    2    3    4    5
# b    1    3    5    7    9
# c    2    4    6    8    0
```

行列

同じ長さのベクトルを複数束ねてできるオブジェクトを行列という。複数のベクトルを行方向に束ねるときは rbind 関数を使用し、列方向に束ねるときは cbind 関数を使用する。




```
a <- c(1, 2, 3, 4, 5)
b <- c(1, 3, 5, 7, 9)
c <- c(2, 4, 6, 8, 0)

m <- cbind(a, b, c)
m
#      a b c
# [1,] 1 1 2
# [2,] 2 3 4
# [3,] 3 5 6
# [4,] 4 7 8
# [5,] 5 9 0
```

行列

行列の行数や列数を調べるには dim、ncol や nrow 関数を使用できる。dim 関数は行数と列数を同時に取得するが、ncol 関数は列数、nrow は行数のみを取得する関数である。

```
a <- c(1, 2, 3, 4, 5)
b <- c(1, 3, 5, 7, 9)
c <- c(2, 4, 6, 8, 0)

n <- rbind(a, b, c)
n
#   [,1] [,2] [,3] [,4] [,5]
# a    1    2    3    4    5
# b    1    3    5    7    9
# c    2    4    6    8    0

dim(n)
# [1] 3 5

ncol(n)
# 5

nrow(n)
# 3
```

行列要素参照

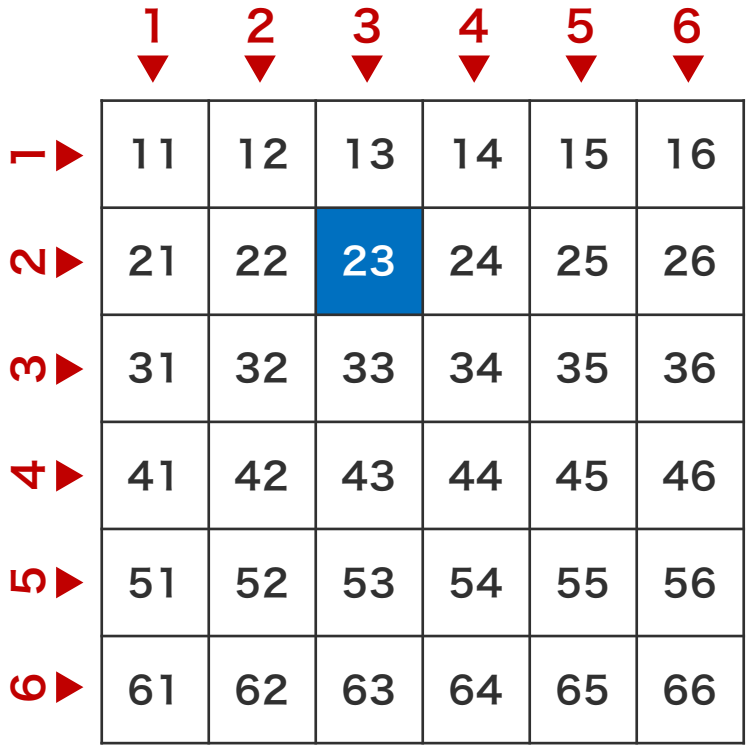
行列は行方向と列方向の 2 次元からなるため、要素を参照するには、行番号と列番号の両方を指定する必要がある。

	1	2	3	4	5	6
1	11	12	13	14	15	16
2	21	22	23	24	25	26
3	31	32	33	34	35	36
4	41	42	43	44	45	46
5	51	52	53	54	55	56
6	61	62	63	64	65	66

```
b <- rbind(c(11, 12, 13, 14, 15, 16),
           c(21, 22, 23, 24, 25, 26),
           c(31, 32, 33, 34, 35, 36),
           c(41, 42, 43, 44, 45, 46),
           c(51, 52, 53, 54, 55, 56),
           c(61, 62, 63, 64, 65, 66))

b[2, 3]


b[2, 3:5]


b[2:5, 4]


b[3:4, 2:6]
```

行列要素参照

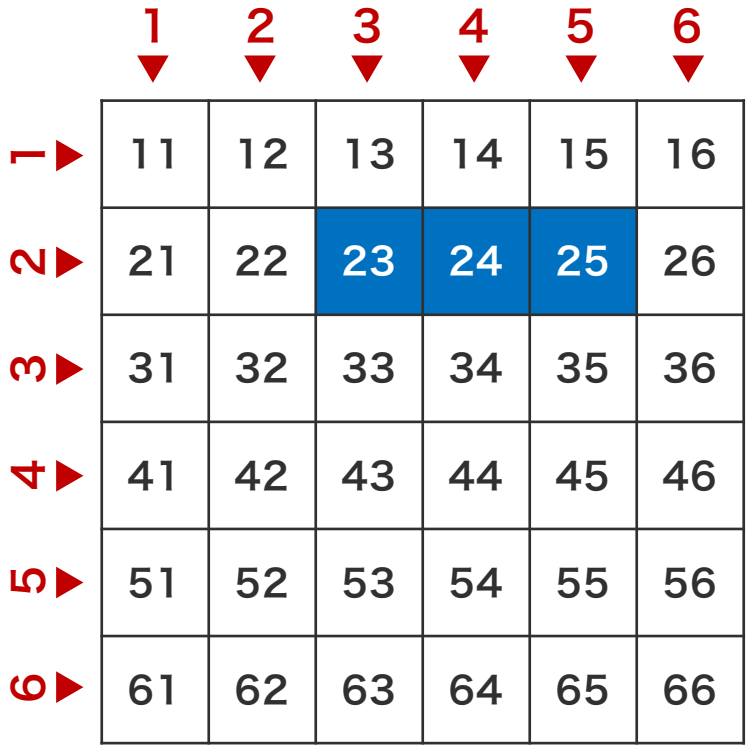
行列は行方向と列方向の 2 次元からなるため、要素を参照するには、行番号と列番号の両方を指定する必要がある。




```
b <- rbind(c(11, 12, 13, 14, 15, 16),
           c(21, 22, 23, 24, 25, 26),
           c(31, 32, 33, 34, 35, 36),
           c(41, 42, 43, 44, 45, 46),
           c(51, 52, 53, 54, 55, 56),
           c(61, 62, 63, 64, 65, 66))

b[2, 3]
# [1] 23

b[2, 3:5]


b[2:5, 4]


b[3:4, 2:6]
```

行列要素参照

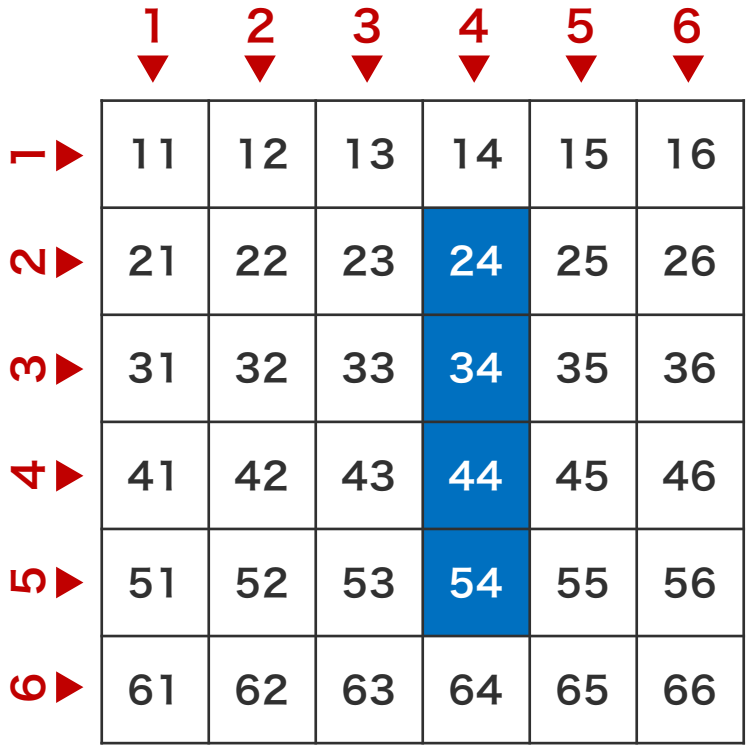
行列は行方向と列方向の 2 次元からなるため、要素を参照するには、行番号と列番号の両方を指定する必要がある。




```
b <- rbind(c(11, 12, 13, 14, 15, 16),
           c(21, 22, 23, 24, 25, 26),
           c(31, 32, 33, 34, 35, 36),
           c(41, 42, 43, 44, 45, 46),
           c(51, 52, 53, 54, 55, 56),
           c(61, 62, 63, 64, 65, 66))

b[2, 3]
# [1] 23

b[2, 3:5]
# [1] 23 24 25

b[2:5, 4]


b[3:4, 2:6]
```

行列要素参照

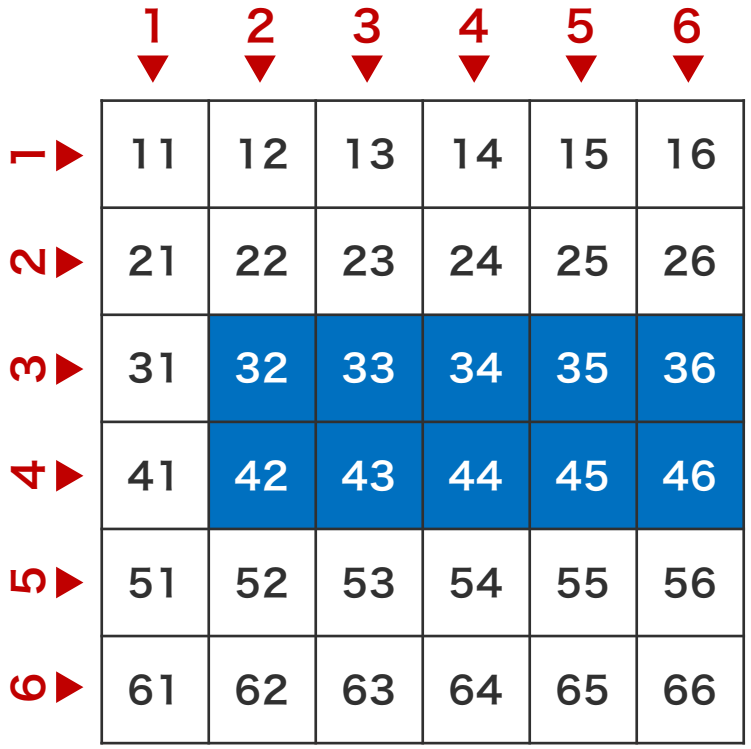
行列は行方向と列方向の 2 次元からなるため、要素を参照するには、行番号と列番号の両方を指定する必要がある。




```
b <- rbind(c(11, 12, 13, 14, 15, 16),
           c(21, 22, 23, 24, 25, 26),
           c(31, 32, 33, 34, 35, 36),
           c(41, 42, 43, 44, 45, 46),
           c(51, 52, 53, 54, 55, 56),
           c(61, 62, 63, 64, 65, 66))

b[2, 3]
# [1] 23

b[2, 3:5]
# [1] 23 24 25

b[2:5, 4]
# [1] 24 34 44 54

b[3:4, 2:6]
```

行列要素参照

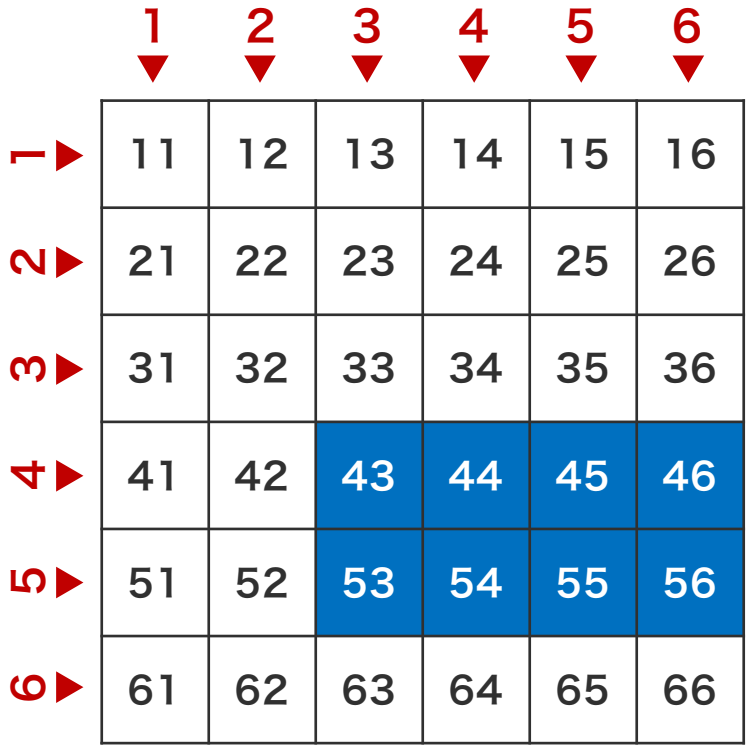
行列は行方向と列方向の 2 次元からなるため、要素を参照するには、行番号と列番号の両方を指定する必要がある。




```
b <- rbind(c(11, 12, 13, 14, 15, 16),
           c(21, 22, 23, 24, 25, 26),
           c(31, 32, 33, 34, 35, 36),
           c(41, 42, 43, 44, 45, 46),
           c(51, 52, 53, 54, 55, 56),
           c(61, 62, 63, 64, 65, 66))

b[2, 3]
# [1] 23

b[2, 3:5]
# [1] 23 24 25

b[2:5, 4]
# [1] 24 34 44 54

b[3:4, 2:6]
#      [,1] [,2] [,3] [,4] [,5]
# [1,]   32   33   34   35   36
# [2,]   42   43   44   45   46
```

行列要素参照

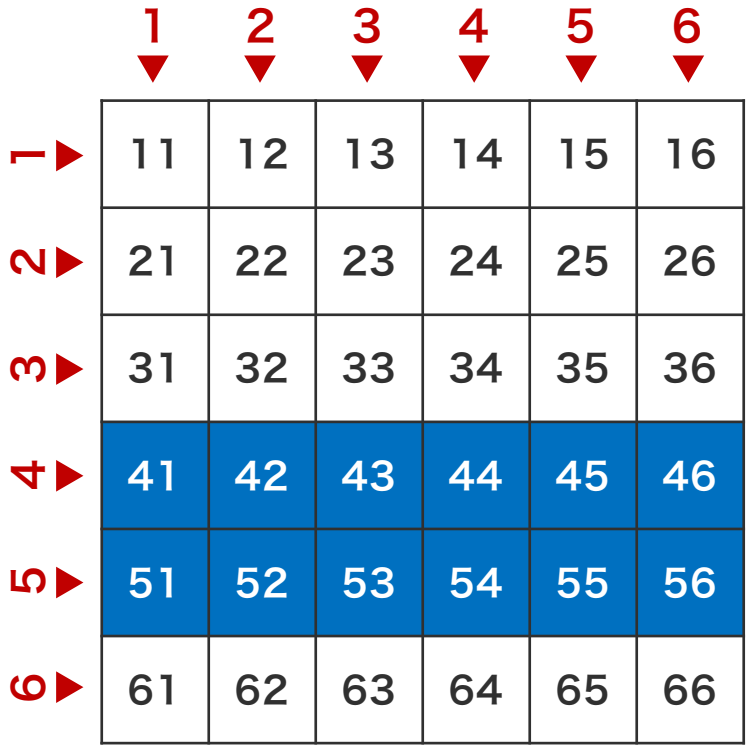
行列は行方向と列方向の 2 次元からなるため、要素を参照するには、行番号と列番号の両方を指定する必要がある。




```
b <- rbind(c(11, 12, 13, 14, 15, 16),
           c(21, 22, 23, 24, 25, 26),
           c(31, 32, 33, 34, 35, 36),
           c(41, 42, 43, 44, 45, 46),
           c(51, 52, 53, 54, 55, 56),
           c(61, 62, 63, 64, 65, 66))

b[4:5, 3:ncol(b)]
#      [,1] [,2] [,3] [,4]
# [1,]   43   44   45   46
# [2,]   53   54   55   56

b[4:5, ]
```

行列要素参照

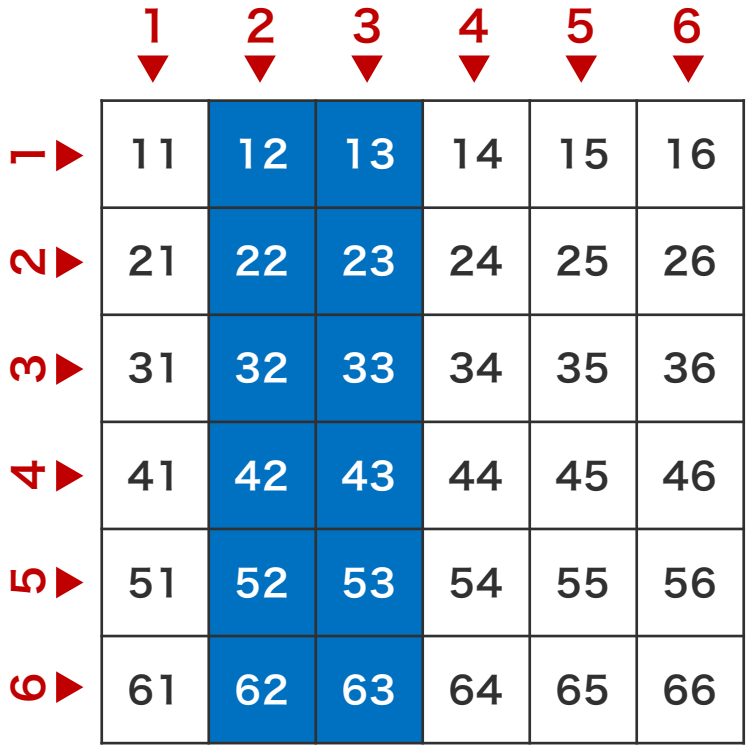
行列は行方向と列方向の 2 次元からなるため、要素を参照するには、行番号と列番号の両方を指定する必要がある。




```
b <- rbind(c(11, 12, 13, 14, 15, 16),
           c(21, 22, 23, 24, 25, 26),
           c(31, 32, 33, 34, 35, 36),
           c(41, 42, 43, 44, 45, 46),
           c(51, 52, 53, 54, 55, 56),
           c(61, 62, 63, 64, 65, 66))

b[4:5, 3:ncol(b)]
#      [,1] [,2] [,3] [,4]
# [1,]   43   44   45   46
# [2,]   53   54   55   56

b[4:5, ]
#      [,1] [,2] [,3] [,4] [,5] [,6]
# [1,]   41   42   43   44   45   46
# [2,]   51   52   53   54   55   56
```

行列要素参照

行列は行方向と列方向の 2 次元からなるため、要素を参照するには、行番号と列番号の両方を指定する必要がある。




```
b <- rbind(c(11, 12, 13, 14, 15, 16),
           c(21, 22, 23, 24, 25, 26),
           c(31, 32, 33, 34, 35, 36),
           c(41, 42, 43, 44, 45, 46),
           c(51, 52, 53, 54, 55, 56),
           c(61, 62, 63, 64, 65, 66))

b[, 2:3]
#      [,1] [,2]
# [1,]   12   13
# [2,]   22   23
# [3,]   32   33
# [4,]   42   43
# [5,]   52   53
# [6,]   62   63
```

問題 O2-1

5 min

行列 x の偶数列目を取り出せ。

```
x <- cbind(c(3, 5, 3, 1),
           c(9, 4, 8, 5),
           c(3, 6, 5, 1),
           c(3, 5, 6, 0))
x
#      [,1] [,2] [,3] [,4]
# [1,]    3    9    3    3
# [2,]    5    4    6    5
# [3,]    3    8    5    6
# [4,]    1    5    1    0
```

行列計算

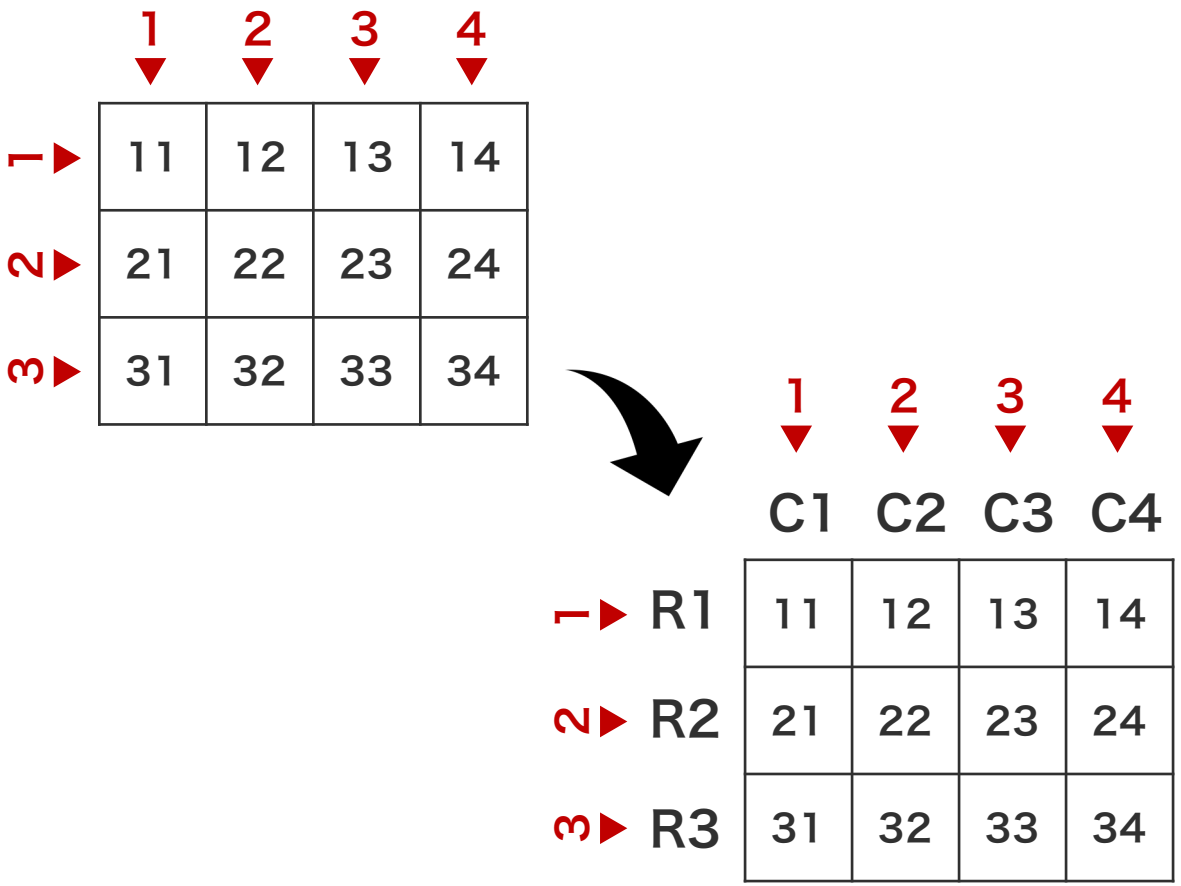
行列同士では、通常の要素同士の四則演算のほかに、内積や外積を求めたりすることができる。

演算子	計算
a + b	行列の各要素同士の加算
a - b	行列の各要素同士の減算
a * b	行列の各要素同士の乗算
a / b	行列の各要素同士の除算
a %*% b	行列同士の内積
a %o% b	行列同士の外積
sum(a)	行列 a の全要素の合計
t(a)	行列 a を転置
solve(a)	行列 a の逆行列
eigen(a)	行列 a の固有値と固有ベクトル
det(a)	行列 a の行列式

```
a <- matrix(c(1, 0, 0, 1), ncol = 2)
b <- matrix(c(1, 2, 3, 4), ncol = 2)

a + b
#      [,1] [,2]
# [1,]    2    3
# [2,]    2    5

a * b
#      [,1] [,2]
# [1,]    1    0
# [2,]    0    4

a %*% b
#      [,1] [,2]
# [1,]    1    3
# [2,]    2    4
```

行名・列名

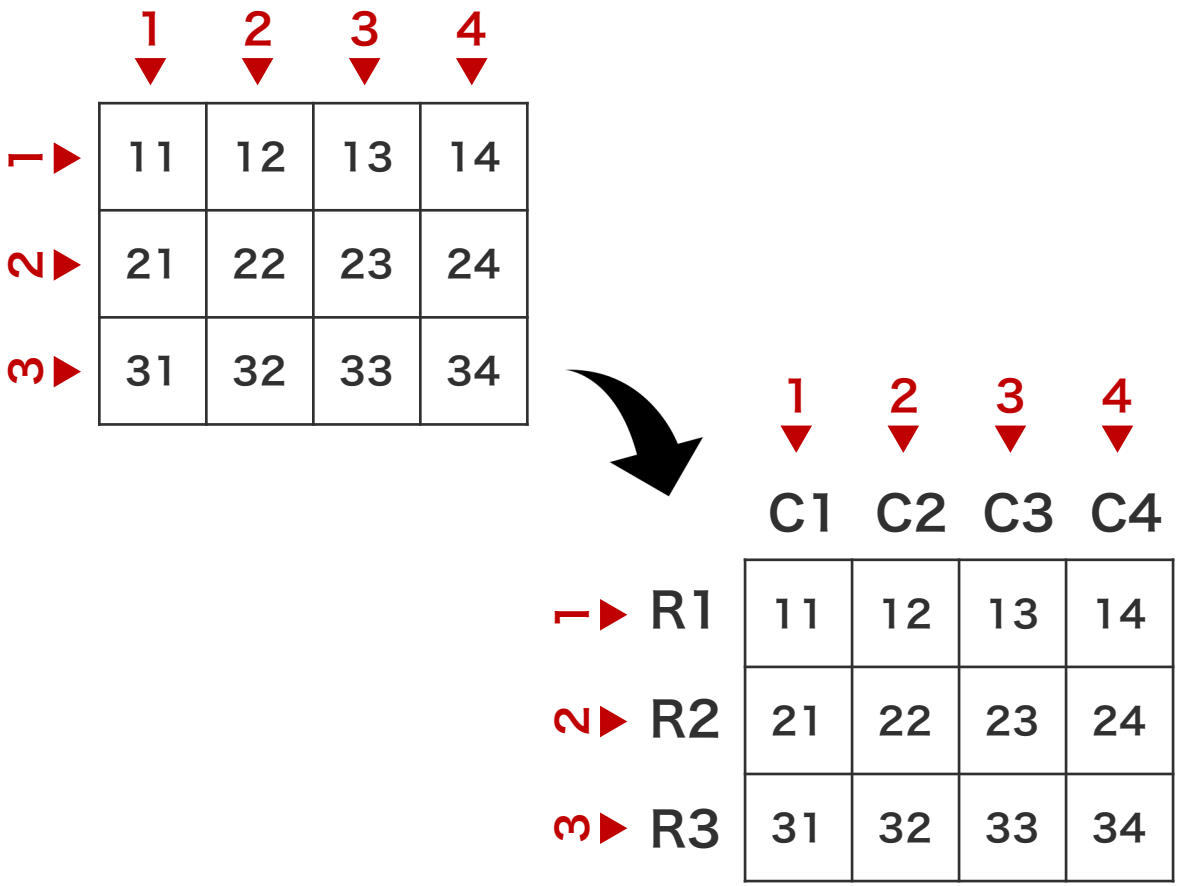
rownames と colnames 関数を使用して、行列に行名と列名を付けることができる。




```
b <- rbind(c(11, 12, 13, 14),
           c(21, 22, 23, 24),
           c(31, 32, 33, 34))

rownames(b) <- c('R1', 'R2', 'R3')
colnames(b) <- c('C1', 'C2', 'C3', 'C4')


b
#    C1 C2 C3 C4
# R1 11 12 13 14
# R2 21 22 23 24
# R3 31 32 33 34
```

行名・列名

行名または列名を持つ行列に対して、名前で要素を取得することもできる。




```
b <- rbind(c(11, 12, 13, 14),
           c(21, 22, 23, 24),
           c(31, 32, 33, 34))

rownames(b) <- c('R1', 'R2', 'R3')
colnames(b) <- c('C1', 'C2', 'C3', 'C4')


b['R2', ]
# C1 C2 C3 C4
# 21 22 23 24


b[, c('C1', 'C3')]
#    C1 C3
# R1 11 13
# R2 21 23
# R3 31 33
```

apply

行列の各行に対して、同じ処理を行う場合は、apply 関数を使用すると便利である。例えば、各行の平均を計算するのに、apply 関数を用いたときとそうでないときは、右のような書き方ができる。

x

4	6	5
11	13	12
21	22	24

row_mean

5.00000
12.0000
22.33333

apply 関数の第 1 引数には処理対象の行列を与える。第 2 引数には、数字の 1 または 2 を与える。1 は行列の各行に対して、2 は各列に対して演算処理を行う。演算処理を行う関数は第 3 引数で与える。

```
x <- rbind(c(4,  6,  5),
           c(11, 13, 12),
           c(21, 22, 24))


x1 <- mean(x[1, ])
x2 <- mean(x[2, ])
x3 <- mean(x[3, ])
row_mean <- c(x1, x2, x3)
row_mean
# [1]  5.00000 12.00000 22.33333


row_mean <- apply(x, 1, mean)
row_mean
# [1]  5.00000 12.00000 22.33333
```

apply

各列に対して同じ処理を行う場合も同様に書ける。

x

4	6	5
11	13	12
21	22	24

col_mean

12.00	13.67	13.67

apply 関数の第 1 引数には処理対象の行列を与える。第 2 引数には、数字の 1 または 2 を与える。1 は行列の各行に対して、2 は各列に対して演算処理を行う。演算処理を行う関数は第 3 引数で与える。

```
x <- rbind(c(4,  6,  5),
           c(11, 13, 12),
           c(21, 22, 24))


x1 <- mean(x[, 1])
x2 <- mean(x[, 2])
x3 <- mean(x[, 3])
col_mean <- c(x1, x2, x3)
col_mean
# [1] 12.00000 13.66667 13.66667


col_mean <- apply(x, 2, mean)
col_mean
# [1] 12.00000 13.66667 13.66667
```

apply

apply 関数の第 3 引数に与える関数は、R で用意された関数以外に、ユーザー自信が定義した関数を与えることもできる。例えば、右のサンプルコードでは、apply 関数は次のように処理を行っている。

1. 行列 x の 1 行目のデータを y に代入する。ベクトル y に対して、function 内部の処理を行う。 計算結果を返す。
2. 行列 x の 2 行目のデータを y に代入する。 ベクトル y に対して、function 内部の処理を行う。 計算結果を返す。
3. 行列 x の 3 行目のデータを y に代入する。 ベクトル y に対して、function 内部の処理を行う。 計算結果を返す。

```
x <- rbind(c(4,  6,  5,  6,  3,  1),
           c(11, 13, 12, 13, 14, 10),
           c(21, 22, 24, 23, 25, 23))

maxmin_diff <- function(y) {
  d <- max(y) - min(y)
  return(d)
}

x1 <- maxmin_diff(x[1, ])
x2 <- maxmin_diff(x[2, ])
x3 <- maxmin_diff(x[3, ])
z <- c(x1, x2, x3)
z
# [1] 5 4 4

z <- apply(x, 1, maxmin_diff)
z
# [1] 5 4 4
```

各行に対して処理を行う。

apply

apply 関数の第 3 引数に与える関数は、R で用意された関数以外に、ユーザー自信が定義した関数を与えることもできる。例えば、右のサンプルコードでは、apply 関数は次のように処理を行っている。

1. 行列 x の 1 列目のデータを y に代入する。ベクトル y に対して、function 内部の処理を行う。 計算結果を返す。
2. 行列 x の 2 列目のデータを y に代入する。 ベクトル y に対して、function 内部の処理を行う。 計算結果を返す。
3. 行列 x の 3 列目のデータを y に代入する。 ベクトル y に対して、function 内部の処理を行う。 計算結果を返す。

```
x <- rbind(c(4,  6,  5,   6,  3,  1),
           c(11, 13, 12, 13, 14, 10),
           c(21, 22, 24, 23, 25, 23))

maxmin_diff <- function(y) {
  d <- max(y) - min(y)
  return(d)
}

x1 <- maxmin_diff(x[, 1])
x2 <- maxmin_diff(x[, 2])
x3 <- maxmin_diff(x[, 3])
z <- c(x1, x2, x3)
z
# [1] 17 16 19

z <- apply(x, 2, maxmin_diff)
z
# [1] 17 16 19
```

各列に対して処理を行う。

問題 O2-2

5 min

次の行列に対して、apply 関数を用いて、各行および各列の最大値と最小値を計算せよ。

```
x <- cbind(c(3, 5, 3, 1),
           c(9, 4, 8, 5),
           c(3, 6, 5, 1))
x
#      [,1] [,2] [,3]
# [1,]    3    9    3
# [2,]    5    4    6
# [3,]    3    8    5
# [4,]    1    5    1
```

問題 O2-3

5 min

次の行列に対して、apply 関数を用いて、各行および各列の最大値と最小値の差を計算せよ。

```
x <- cbind(c(3, 5, 3, 1),
           c(9, 4, 8, 5),
           c(3, 6, 5, 1))
x
#      [,1] [,2] [,3]
# [1,]    3    9    3
# [2,]    5    4    6
# [3,]    3    8    5
# [4,]    1    5    1
```

データ型

- ベクトル
- 行列
- データフレーム
- リスト
- 文字列

データフレーム

行列

行列は、行と列を持つ２次元配置のデータである。行列の各要素の単位は同じである。行方向に対しても、列方向に対しても、分析を行うことが可能である。

	2010	2012	2014	2016	2018
Aomori	43.4	44.1	48.3	45.3	42.8
Iwate	65.2	65.3	66.4	61.8	63.6
Akita	65.3	66.8	64.8	69.3	71.3
Niigata	60.1	66.4	64.7	63.8	62.0
Miyagi	42.4	46.5	49.2	45.2	45.9
Nagano	30.3	30.1	30.2	30.5	30.4
Chiba	39.4	40.3	40.4	41.3	40.3

データフレーム

データフレームは、行列と同じく２次元配置のデータである。データフレームは列ごとに属性が異なることもある。そのため、データフレームは基本的に列を重視した操作になる。

area	year	yield
Akita	2014	64.8
Akita	2016	69.3
Akita	2018	71.3
Aomori	2014	3.5
Aomori	2016	4.6
Aomori	2018	3.2
Chiba	2014	64.7

データフレーム

データフレームは列ごとに属性（例えば、データの単位）が異なっている。そのため、データフレームを作成するときは、基本的に複数の列ベクトルを束ねて作成する。行列を作成するときに使用する cbind と同じイメージである。これに対して、データフレームを作成するときは、data.frame 関数を使用する。データフレームは列ごとに意味が異なり、列を重視したデータ構造である。そのため、データフレームを作成するときに、各列に列名をつける必要がある。

```
x <- c('a', 'b', 'c', 'd', 'e')
h <- c(11.2, 11.7, 10.3, 12.3, 12.5)
w <- c(4.5, 4.2, 3.9, 5.7, 5.1)

d <- data.frame(plantid = x,
                height = h,
                weight = w)

head(d)
#   plantid height weight
# 1       a   11.2    4.5
# 2       b   11.7    4.2
# 3       c   10.3    3.9
# 4       d   12.3    5.7
# 5       e   12.5    5.1
```

データフレーム

データフレームは列ごとに属性（例えば、データの単位）が異なっている。そのため、データフレームを作成するときは、基本的に複数の列ベクトルを束ねて作成する。行列を作成するときに使用する cbind と同じイメージである。これに対して、データフレームを作成するときは、data.frame 関数を使用する。データフレームは列ごとに意味が異なり、列を重視したデータ構造である。そのため、データフレームを作成するときに、各列に列名をつける必要がある。

```
x <- c('a', 'b', 'c', 'd', 'e')
h <- c(11.2, 11.7, 10.3, 12.3, 12.5)
w <- c(4.5, 4.2, 3.9, 5.7, 5.1)

d <- data.frame(plantid = x,
                height = h,
                weight = w)

mean(d$height)
# [1] 11.6

mean(d$h)
# [1] 11.6
```

列名の自動補完が行われる。似たような列名が存在する時にバグの原因になるので十分に注意すること。

データフレーム

cbind 関数でベクトルを行列に束ねてから、それを as.data.frame 関数でデータフレームに変換することも可能だが、データフレームのデータの属性が変化することがある。バグの原因となるため、このような操作は可能な限り避けるとよい。

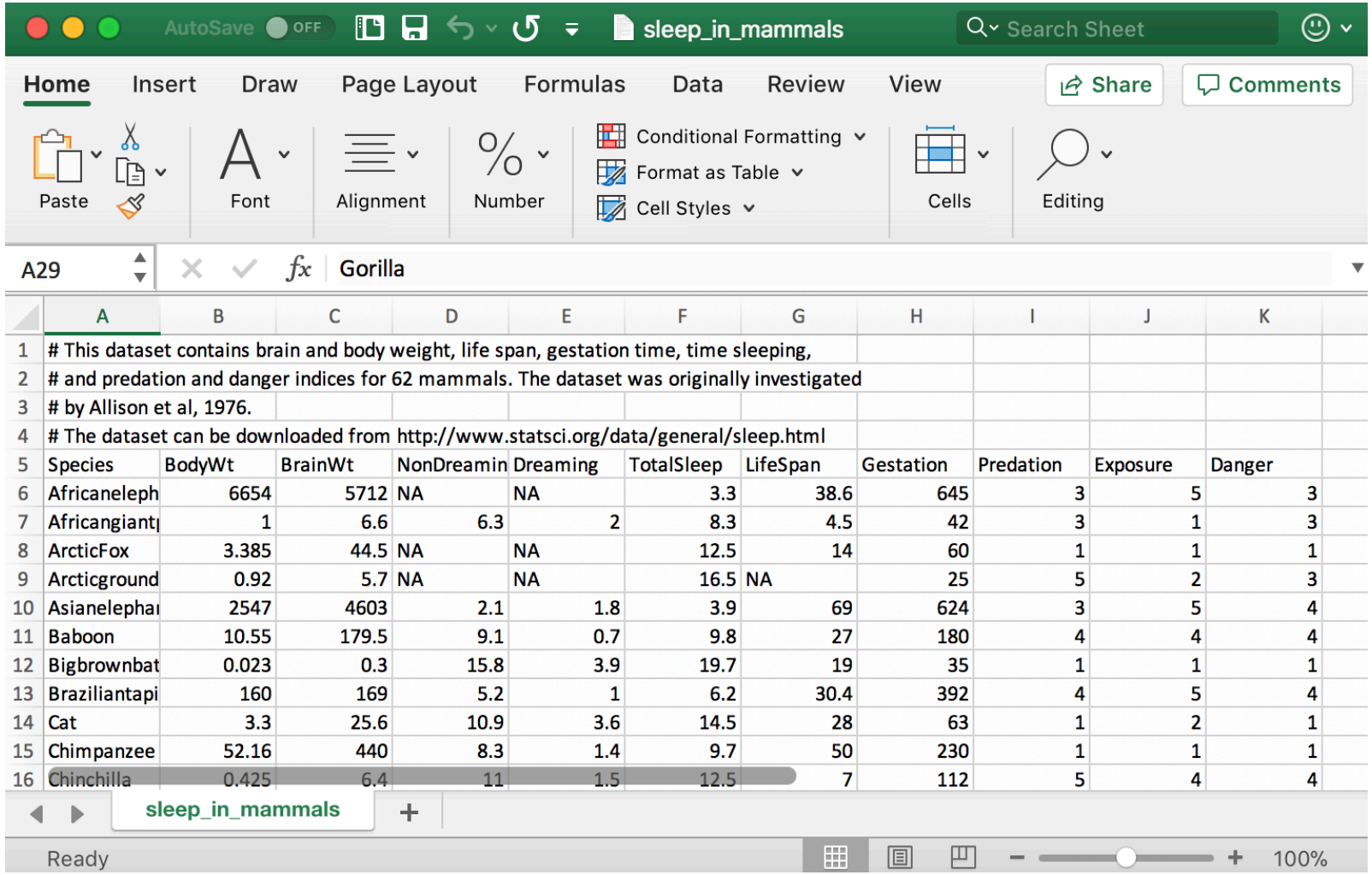
```
x <- c('a', 'b', 'c', 'd', 'e')
h <- c(11.2, 11.7, 10.3, 12.3, 12.5)
w <- c(4.5, 4.2, 3.9, 5.7, 5.1)

d <- as.data.frame(cbind(x, h, w))

head(d)
#   x    h   w
# 1 a 11.2 4.5
# 2 b 11.7 4.2
# 3 c 10.3 3.9
# 4 d 12.3 5.7
# 5 e 12.5 5.1

d$h
# [1] "11.2" "11.7" "10.3" "12.3" "12.5"

sum(d$h)
# Error in sum(d$h) : invalid 'type'
(character) of argument
```

x が文字列であるため、h と w も文字列に変換されて行列となる。

数値に見えるが、実は文字列である。

表データ

生物学で取り扱うデータは、一般的に、各列には属性、各行にサンプルを書く構造をとっている。このような表形式のデータを記録したり、整理したりする場合は、Excel 等の表計算アプリケーションを使うのが一般的である。しかし、プログラミング言語でデータを解析する場合は、Excel データのままでは非常に不便である。ほとんどの場合、Excel からデータをタブ区切りファイル（TSV）またはカンマ区切りファイル（CSV）として書き出してから解析するのが一般的である。



	A	B	C	D	E	F	G	H	I	J	K
1	# This dataset contains brain and body weight, life span, gestation time, time sleeping,										
2	# and predation and danger indices for 62 mammals. The dataset was originally investigated										
3	# by Allison et al, 1976.										
4	# The dataset can be downloaded from http://www.statsci.org/data/general/sleep.html										
5	Species	BodyWt	BrainWt	NonDreamin	Dreaming	TotalSleep	LifeSpan	Gestation	Predation	Exposure	Danger
6	Africaneleph	6654	5712	NA	NA	3.3	38.6	645	3	5	3
7	Africangiant	1	6.6	6.3	2	8.3	4.5	42	3	1	3
8	ArcticFox	3.385	44.5	NA	NA	12.5	14	60	1	1	1
9	Arcticground	0.92	5.7	NA	NA	16.5	NA	25	5	2	3
10	Asianelephar	2547	4603	2.1	1.8	3.9	69	624	3	5	4
11	Baboon	10.55	179.5	9.1	0.7	9.8	27	180	4	4	4
12	Bigbrownbat	0.023	0.3	15.8	3.9	19.7	19	35	1	1	1
13	Braziliantapi	160	169	5.2	1	6.2	30.4	392	4	5	4
14	Cat	3.3	25.6	10.9	3.6	14.5	28	63	1	2	1
15	Chimpanzee	52.16	440	8.3	1.4	9.7	50	230	1	1	1
16	Chinchilla	0.425	6.4	11	1.5	12.5	7	112	5	4	4

表データ

コメント

データ

This dataset contains brain and body weight, life span, gestation time, time sleeping,
and predation and danger indices for 62 mammals. The dataset was originally investigated
by Allison et al, 1976.
The dataset can be downloaded from http://www.statsci.org/data/general/sleep.html

Species	BodyWt	BrainWt	NonDreamin	Dreaming	TotalSleep	LifeSpan	Gestation
Africaneleph	6654	5712	NA	NA	3.3	38.6	645
Africangiant	1	6.6	6.3	2	8.3	4.5	42
ArcticFox	3.385	44.5	NA	NA	12.5	14	60
Arcticground	0.92	5.7	NA	NA	16.5	NA	25
Asianelepha	2547	4603	2.1	1.8	3.9	69	624
Baboon	10.55	179.5	9.1	0.7	9.8	27	180
Bigbrownbat	0.023	0.3	15.8	3.9	19.7	19	35
Braziliantapi	160	169	5.2	1	6.2	30.4	392
Cat	3.3	25.6	10.9	3.6	14.5	28	63

表データ

属性（特徴量）

ヘッダー →

サンプル

Species	BodyWt	BrainWt	NonDreamin	Dreaming	TotalSleep	LifeSpan	Gestation
Africaneleph	6654	5712	NA	NA	3.3	38.6	645
Africangiant	1	6.6	6.3	2	8.3	4.5	42
ArcticFox	3.385	44.5	NA	NA	12.5	14	60
Arcticground	0.92	5.7	NA	NA	16.5	NA	25
Asianelepha	2547	4603	2.1	1.8	3.9	69	624
Baboon	10.55	179.5	9.1	0.7	9.8	27	180
Bigbrownbat	0.023	0.3	15.8	3.9	19.7	19	35
Braziliantapi	160	169	5.2	1	6.2	30.4	392
Cat	3.3	25.6	10.9	3.6	14.5	28	63

欠損値

ファイル読み込み

CSV または TSV データを読み込むとき、read.table 関数を使用する。

```
filename <- 'sleep_in_mammals.txt'
d <- read.table(filename)


head(d)
```

ヘッダー行もデータの一部として読み込まれている。 ▶

```
                    V1     V2      V3          V4       V5         V6       V7
1              Species BodyWt BrainWt NonDreaming Dreaming TotalSleep LifeSpan
2       Africanelephant   6654    5712        <NA>     <NA>       3.3     38.6
3 Africangiantpouchedrat      1     6.6         6.3        2       8.3      4.5
4            ArcticFox  3.385    44.5        <NA>     <NA>      12.5       14
5  Arcticgroundsquirrel   0.92     5.7        <NA>     <NA>      16.5     <NA>
6         Asianelephant   2547    4603         2.1      1.8       3.9       69
        V8        V9       V10    V11
1 Gestation Predation Exposure Danger
2      645         3         5      3
3       42         3         1      3
4       60         1         1      1
5       25         5         2      3
6      624         3         5      4
```

https://aabbdd.jp/data/sleep_in_mammals.txt

ファイル読み込み

read.table 関数を使用するとき、ヘッダー付きのタブ区切りのファイルを読み込むことを明記して実行すると、ヘッダー行がデータフレームの列名となる。

```
filename <- 'sleep_in_mammals.txt'
d <- read.table(filename,
                sep = '\t',
                header = TRUE)

head(d)
```

ヘッダー行がデータフレームの列名となった。 ▶

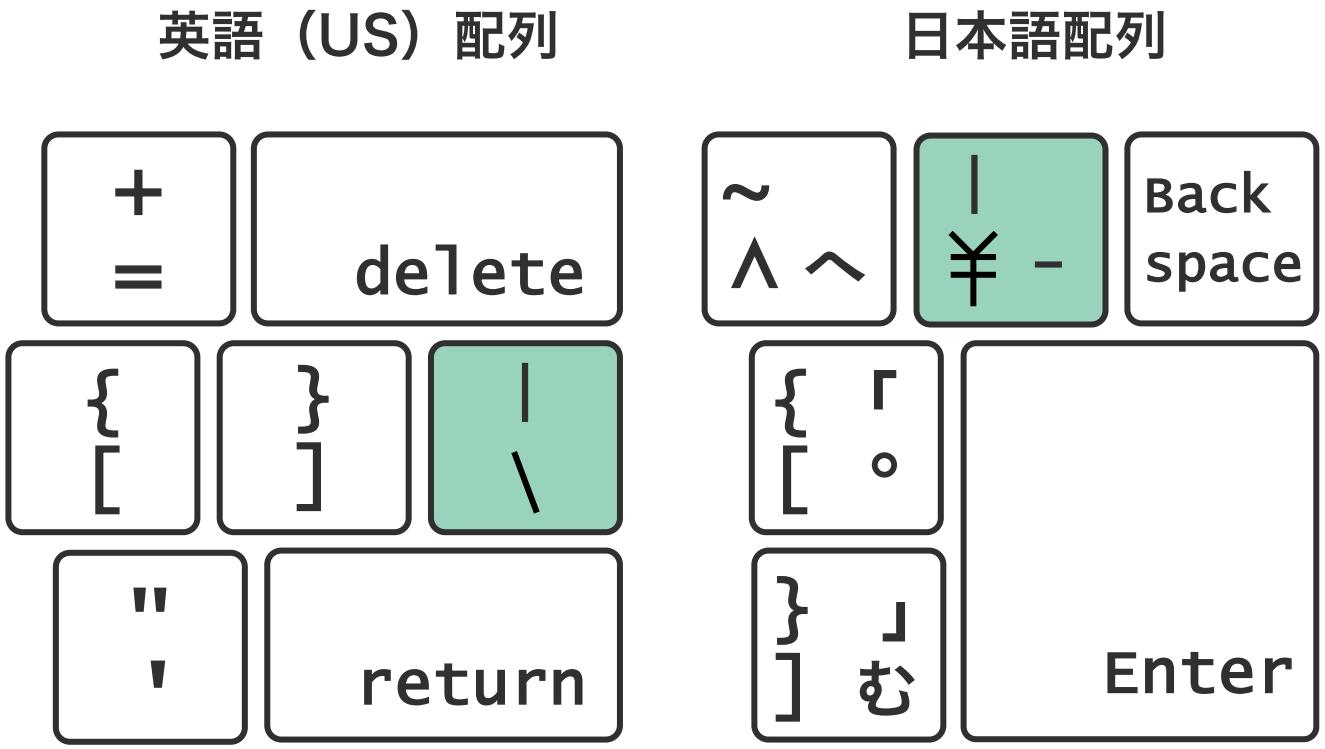
	Species	BodyWt	BrainWt	NonDreaming	Dreaming	TotalSleep	LifeSpan
1	Africanelephant	6654.000	5712.0	NA	NA	3.3	38.6
2	Africangiantpouchedrat	1.000	6.6	6.3	2.0	8.3	4.5
3	ArcticFox	3.385	44.5	NA	NA	12.5	14.0
4	Arcticgroundsquirrel	0.920	5.7	NA	NA	16.5	NA
5	Asianelephant	2547.000	4603.0	2.1	1.8	3.9	69.0
6	Baboon	10.550	179.5	9.1	0.7	9.8	27.0

	Gestation	Predation	Exposure	Danger
1	645	3	5	3
2	42	3	1	3
3	60	1	1	1
4	25	5	2	3
5	624	3	5	4
6	180	4	4	4

https://aabbdd.jp/data/sleep_in_mammals.txt

バックスラッシュ（＼）・円マーク（¥）

バックスラッシュは、OS の種類、言語環境や文字コードによって、ディスプレイでの表示が異なる。日本語環境のシステムであれば円マーク「¥」として表示され、それ以外の言語環境では「＼」として表示される。バックスラッシュと円マークは、表示が異なるものの、コンピューター上では同一のコード 0x5C として扱われる。また、バックスラッシュは特別な制御を行うための文字であり、一般的な文字として使用されることは少ない。



macOS 日本語環境を使用している場合は、＼ と ¥ の両方を入力できる。Option を押しながら＼または¥キーを押すことで、＼と¥の入力の切り替えができる。

データフレーム

read.table 関数を使用して CSV または TSV データを読み込むとき、データファイルの 1 列目が列名として読み込みたい場合は、row.names = 1 オプションを指定する。

```
filename <- 'sleep_in_mammals.txt'
d <- read.table(filename,
                header = TRUE,
                row.names = 1,
                sep = '\t')
head(d)
```

データの 1 列目が行名となった。

▼

ヘッダー行がデータフレームの列名となった。 ▶

	BodyWt	BrainWt	NonDreaming	Dreaming	TotalSleep	LifeSpan
Africanelephant	6654.000	5712.0	NA	NA	3.3	38.6
Africangiantpouchedrat	1.000	6.6	6.3	2.0	8.3	4.5
ArcticFox	3.385	44.5	NA	NA	12.5	14.0
Arcticgroundsquirrel	0.920	5.7	NA	NA	16.5	NA
Asianelephant	2547.000	4603.0	2.1	1.8	3.9	69.0
Baboon	10.550	179.5	9.1	0.7	9.8	27.0
	Gestation	Predation	Exposure	Danger		
Africanelephant	645	3	5	3		
Africangiantpouchedrat	42	3	1	3		
ArcticFox	60	1	1	1		
Arcticgroundsquirrel	25	5	2	3		
Asianelephant	624	3	5	4		
Baboon	180	4	4	4		

データフレーム

データフレームの要素を参照するとき、基本的に行列の要素参照と同じ方法で行える。行または列の位置番号あるいは行名・列名の両方で要素を取得できる。

```
filename <- 'sleep_in_mammals.txt'
d <- read.table(filename,
                header = TRUE,
                row.names = 1,
                sep = '\t')


species <- c(9, 48, 37)
features <- c(1, 5)
d[species, features]
#       BodyWt TotalSleep
# Cat    3.300       14.5
# Rat    0.280       13.2
# Mouse  0.023       13.2
```

データフレーム

データフレームの要素を参照するとき、基本的に行列の要素参照と同じ方法で行える。行または列の位置番号あるいは行名・列名の両方で要素を取得できる。

```
filename <- 'sleep_in_mammals.txt'
d <- read.table(filename,
                header = TRUE,
                row.names = 1,
                sep = '\t')


species <- c('Cat', 'Rat', 'Mouse')
features <- c('BodyWt', 'TotalSleep')
d[species, features]
#       BodyWt TotalSleep
# Cat    3.300       14.5
# Rat    0.280       13.2
# Mouse  0.023       13.2
```

データフレーム

データフレームから特定の列を取り出す際に、$ または [[]] を使用して、ベクトルとして取り出せる。

```
filename <- 'sleep_in_mammals.txt'
d <- read.table(filename,
                header = TRUE,
                row.names = 1,
                sep = '\t')


head(d$BodyWt)
# [1] 6654.000  1.000  3.385  0.920
# [5] 2547.000 10.550

head(d[['TotalSleep']])
# [1] 3.3 8.3 12.5 16.5 3.9 9.8
```

データフレーム

データフレームではある列の情報に基づいてデータ全体を操作したりすることができる。例えば、右のコードは、睡眠時間（TotalSleep）が 10 時間以上の動物のデータの行を出力している例である。

```
filename <- 'sleep_in_mammals.txt'
d <- read.table(filename,
                header = TRUE,
                row.names = 1,
                sep = '\t')


totalsleep <- d$TotalSleep
keep <- (totalsleep > 19)
d[keep, ]
```

	BodyWt	BrainWt	NonDreaming	Dreaming	TotalSleep	LifeSpan	Gestation
Bigbrownbat	0.023	0.30	15.8	3.9	19.7	19	35
NA	NA	NA	NA	NA	NA	NA	NA
NA.1	NA	NA	NA	NA	NA	NA	NA
Littlebrownbat	0.010	0.25	17.9	2.0	19.9	24	50
NAmericanopossum	1.700	6.30	13.8	5.6	19.4	5	12
NA.2	NA	NA	NA	NA	NA	NA	NA
Wateropossum	3.500	3.90	12.8	6.6	19.4	3	14
NA.3	NA	NA	NA	NA	NA	NA	NA

	Predation	Exposure	Danger
Bigbrownbat	1	1	1
NA	NA	NA	NA
NA.1	NA	NA	NA
Littlebrownbat	1	1	1
NAmericanopossum	2	1	1
NA.2	NA	NA	NA
Wateropossum	2	1	1
NA.3	NA	NA	NA

データフレーム

データフレームではある列の情報に基づいてデータ全体を操作したりすることができる。例えば、右のコードは、睡眠時間（TotalSleep）が 10 時間以上の動物のデータの行を出力している例である。

	BodyWt	BrainWt	NonDreaming	Dreaming	TotalSleep	LifeSpan	Gestation
Bigbrownbat	0.023	0.30	15.8	3.9	19.7	19	35
NA	NA	NA	NA	NA	NA	NA	NA
NA.1	NA	NA	NA	NA	NA	NA	NA
Littlebrownbat	0.010	0.25	17.9	2.0	19.9	24	50
NAmericanopossum	1.700	6.30	13.8	5.6	19.4	5	12
NA.2	NA	NA	NA	NA	NA	NA	NA
Wateropossum	3.500	3.90	12.8	6.6	19.4	3	14
NA.3	NA	NA	NA	NA	NA	NA	NA

	Predation	Exposure	Danger
Bigbrownbat	1	1	1
NA	NA	NA	NA
NA.1	NA	NA	NA
Littlebrownbat	1	1	1
NAmericanopossum	2	1	1
NA.2	NA	NA	NA
Wateropossum	2	1	1
NA.3	NA	NA	NA

```
filename <- 'sleep_in_mammals.txt'
d <- read.table(filename,
                header = TRUE,
                row.names = 1,
                sep = '\t')


totalsleep <- d$TotalSleep
keep <- (totalsleep > 19)
d[keep, ]


d[(d$TotalSleep > 19), ]
```

データフレーム

データフレームではある列の情報に基づいてデータ全体を操作したりすることができる。例えば、右のコードは、重量（BodyWt）の最も重い動物データの行を出力している例である。

```
filename <- 'sleep_in_mammals.txt'
d <- read.table(filename,
                header = TRUE,
                row.names = 1,
                sep = '\t')

wt <- d$BodyWt
max.wt <- max(wt)
max.wt
# [1] 6654

keep <- (wt == max.wt)
d[keep, ]
```

```
                BodyWt BrainWt NonDreaming Dreaming TotalSleep LifeSpan Gestation
Africanelephant   6654    5712          NA       NA        3.3     38.6       645
                Predation Exposure Danger
Africanelephant         3        5      3
```

```
d[d$BodyWt == max(d$BodyWt), ]
```

問題 O3-1

10 min

1. diversity_galapagos.txt ファイルを読み込んで、島の名前 Island、島の面積 Area、種の数Speciesの3列からなるデータフレームを作れ。
2. 面積の最も大きい島の名前とその島にある種の数を答えよ。ただし、ベクトルの中から最大値を求める関数として max を使うこと。
3. 各島における面積あたりの種数を計算し、面積当たりの種数が最も大きい島の名前を答えよ。

```
filename <- 'diversity_galapagos.txt'
```

https://aabbdd.jp/data/diversity_galapagos.txt

問題 O3-2

10 min

1. rice.txt ファイルを読み込み、wt 系統（variety）の F10 処理群（trt）の root_dry_mass の平均および分散を求めよ。
2. rice.txt ファイルを読み込み、wt 系統の F10 処理群の乾燥重量（root_dry_mass と shoot_dry_mass の合計値）の平均および分散を求めよ。

https://aabbdd.jp/data/rice.txt

```
filename <- 'rice.txt'
```

ファイル保存

データフレームまたは行列からなる計算結果を保存する際に write.table を使用する。write.table で使用するオプションは、read.table で使用するものと同じです。

```
filename <- 'sleep_in_mammals.txt'
d <- read.table(filename,
                header = TRUE,
                row.names = 1,
                sep = '\t')

species <- c('Cat', 'Rat', 'Mouse')
features <- c('BodyWt', 'TotalSleep')
d[species, features]

output.filename <- 'subset.txt'
write.table(d, file = output.filename,
            col.names = TRUE,
            row.names = TRUE,
            sep = '\t')
```

https://aabbdd.jp/data/sleep_in_mammals.txt

因子型オブジェクト

古いバージョンの R では、ファイルを読み込んだりすると、ファイル中の文字列が文字列型オブジェクトとしてではなく、因子型オブジェクトとして読み込まれる。これが原因となって、想定外の動作を行うことがある。これを防ぐために、文字列が因子型でなければならないという特別な理由がなければ、read.table 関数を使用するときに stringsAsFactors を FALSE に指定するとよい。

```
filename <- 'sleep_in_mammals.txt'
d <- read.table(filename,
                header = TRUE,
                sep = '\t',
          stringsAsFactors = FALSE)
```

データ型

- ベクトル
- 行列
- データフレーム
- リスト
- 文字列

リスト

R のリストは、データフレームに似た操作を行えるデータ構造である。データフレームは列ベクトルを束ねて名前をつけて管理しているのに対して、リストは各要素を束ねて名前をつけて管理している。また、データフレームは、各要素（各列）が長さ同じのベクトルである必要があるのに対して、リストは各要素が異なる型のオブジェクト（データ）でも構わない。なお、リストを作成するときに各要素に名前を付けなかった場合、各要素が順に 1, 2, ... のようなインデックスで管理される。

```
x <- vector('list', length = 5)
# [[1]]
# NULL
# [[2]]
# NULL
# [[3]]
# NULL
# [[4]]
# NULL
# [[5]]
# NULL
```

リスト

リストの各要素が異なる型のオブジェクトであってもよい。例えば、リストの 1 つ目の要素にベクトルを格納し、2 つ目の要素に行列を格納することもできる。このリストを作成するときに、名前を付けなかったために、リストの各要素は 1 および 2 のインデックスで管理される。

```
x <- list()
x <- c(x, list(1:9))
x <- c(x, list(matrix(1:9, ncol = 3)))


x
# [[1]]
# [1] 1 2 3 4 5 6 7 8 9
#
# [[2]]
#      [,1] [,2] [,3]
# [1,]    1    4    7
# [2,]    2    5    8
# [3,]    3    6    9
```

リスト

リストの要素に名前を付けるとき names 関数を付ける。このとき、リストの各要素を 1 および 2 の添字のほかに、名前でも操作できるようになる。なお、リストの各要素にすでに名前が付けられているときに、names 関数を使用すると、要素名が更新される。

```
x <- list()
x <- c(x, list(1:9))
x <- c(x, list(matrix(1:9, ncol = 3)))
names(x) <- c('vec', 'mat')

x
# $vec
# [1] 1 2 3 4 5 6 7 8 9
#
# $mat
#      [,1] [,2] [,3]
# [1,]    1    4    7
# [2,]    2    5    8
# [3,]    3    6    9
```

リスト

リストを作成するときに、各要素に名前を付けるときは、次のようにする。

```
x <- list(vec = 1:9,
          mat = matrix(1:9, ncol = 3))
x
# $vec
# [1] 1 2 3 4 5 6 7 8 9
#
# $mat
#      [,1] [,2] [,3]
# [1,]    1    4    7
# [2,]    2    5    8
# [3,]    3    6    9
```

リスト要素参照

リストから各要素を取り出すときには、インデックスまたは名前を利用する。インデックスを使用して要素を取り出すとき [[と]] を使用する、名前を使用して要素を取り出すときは $ を使用する。

```
x <- list(vec = 1:9,
          mat = matrix(1:9, ncol = 3))
x

x[[1]]
# [1] 1 2 3 4 5 6 7 8 9

x$vec
# [1] 1 2 3 4 5 6 7 8 9

x$mat
#      [,1] [,2] [,3]
# [1,]    1    4    7
# [2,]    2    5    8
# [3,]    3    6    9
```

リスト要素参照

リストの名前が数字から始まるときは、`` で囲むと文字列のように扱えるようになる。

```
x <- list(`2` = 1:9,
          `1` = matrix(1:9, ncol = 3))
x
# $`2`
# [1] 1 2 3 4 5 6 7 8 9
#
# $`1`
#      [,1] [,2] [,3]
# [1,]    1    4    7
# [2,]    2    5    8
# [3,]    3    6    9


x[[1]]
# [1] 1 2 3 4 5 6 7 8 9

x$`2`
# [1] 1 2 3 4 5 6 7 8 9
```

apply ファミリー

行列の各行または各列に対して同じ処理を行いたいときは apply 関数を使用する。これと同様にリストの各要素に対して同じ処理を行いたいときは、lapply や sapply 関数などを使用する。これらの関数はまとめて apply ファミリーの関数とよばれたりする。

関数	動作
lapply(x, FUN)	リスト x の各要素に対して FUN で定義された処理を行い、その結果をリストとして返す。
sapply(x, FUN)	リスト x の各要素に対して FUN で定義された処理を行い、その結果をベクトルとして返す。

```
x <- list(mat1 = matrix(1:9, ncol = 3),
          mat2 = matrix(2:10, ncol = 3))



y <- lapply(x, sum)
y
# $mat1
# [1] 45
#
# $mat2
# [1] 54



z <- sapply(x, sum)
z
# mat1 mat2
#   45   54
```

問題 04-1

3 min

apply ファミリーの関数を使って、リストの各要素に対して、最大値を求めよ。

```
x <- list(apple = c(6, 0, 3, 5),
          orange = c(9, 4, 6, 1),
          cherry = c(5, 3, 1, 2))
```

データ型

- ベクトル
- 行列
- データフレーム
- リスト
- 文字列

文字列

R は統計用のプログラミング言語であり、ベクトル演算などを得意とする。逆に、文字列を扱う機能では、他のプログラミング言語などに劣る。ただし、以下のような基本的な機能は実装されている。

- 文字列検索
- 部分文字列の切り出し
- 置換
- 文字列結合
- 文字列分割

```
grep
match
charmatch
pmatch
regexpr
substr
substring
sub
gsub
chartr
paste
paste0
strsplit
```

文字列検索 / grep regexpr

ある文字列パターンが、他の文字列に含まれているかどうかを調べる際に、grep、regexpr match などの関数を使用する。grep および regexpr 関数は部分一致（パターンマッチング）を行う。両者とも正規表現をサポートしている。これに対して、match 関数は完全一致による文字列検索を行う。

```
f <- c('a.R', 'b1.jpg', 'c.txt', 'd1.R')

f2 <- grep('R', f)
f2
# [1] 1 4

f3 <- grep('py', f)
f3
# integer(0)

f4 <- regexpr('R', f)
f4
# [1]  3 -1 -1  4
# attr(,"match.length")
# [1]  1 -1 -1  1
# attr(,"index.type")
# [1] "chars"
# attr(,"useBytes")
# [1] TRUE
```

文字列検索 / match

ある文字列パターンが、他の文字列に含まれているかどうかを調べる際に、grep、regexpr match などの関数を使用する。grep および regexpr 関数は部分一致（パターンマッチング）を行う。両者とも正規表現をサポートしている。これに対して、match 関数は完全一致による文字列検索を行う。

```
f <- c('a.R', 'b1.jpg', 'c.txt', 'd1.R')

f2 <- match('R', f)
f2
# NA

f3 <- match('c.txt', f)
f3
# 3
```

部分文字列切り出し / substr

ある文字列の中から特定位置にある部分文字列を切り出すためには substr 関数を使用する。同様な機能をもつ関数としては substring 関数もあるが、この関数は内部では substr 関数を呼び出して使用している。

```
x <- c('AT1G01','AT2G02','AT3G03')
```

x の各要素に対して、3〜4 つ目の位置にある部分文字列を切り出す。 ▶

```
substr(x, 3, 4)
# [1] "1G" "2G" "3G"
```

x の 1 番目の要素に対して 3〜4、2 番目の要素に対して 3〜4、3 番目の要素に対して 3〜6 つ目の位置にある部分文字列を切り出す。 ▶

```
substr(x, c(3, 3, 3), c(4, 4, 6))
# [1] "1G"   "2G"   "3G03"
```

置換 / substr

substr 関数を利用して置換を行うこともできる。本来、substr 関数で切り出す予定の位置に、新しい文字列を代入することで置換処理になる。なお、置換後の文字列と置換対象部分の文字列の長さが同じでなければならない。

```
x <- c('AT1G01','AT2G02','AT3G03')

substr(x, 1, 2) <- 'at'
x
# [1] "at1G01" "at2G02" "at3G03"


x <- c('AT1G01','AT2G02','AT3G03')

substr(x, 1, 2) <- 'a'
x
# [1] "aT1G01" "aT2G02" "aT3G03"


x <- c('AT1G01','AT2G02','AT3G03')

substr(x, 1, 2) <- 'aaaaa'
x
# [1] "aa1G01" "aa2G02" "aa3G03"
```

置換 / gsub

gsub 関数を使用すると、文字列中から対象部分文字列を自動的に検索し、その部分を置換する。位置番号で指定する必要はなく、効率がいい。

```
x <- c('AT1G01','AT2G02','AT1G03')

x <- gsub('AT', 'at', x)
x
# [1] "at1G01" "at2G02" "at1G03"
```

置換 / chartr

chartr 関数は文字列中の各文字つ 1 つずつを置換していく関数である。

```
x <- c('ACCAGT','CAGTC','CAGTT')

x <- chartr('ACGT', '0123', x)
x
# [1] "011023" "10231"  "10233"
```

文字列結合 / paste

複数の文字列を 1 つの文字として結合したい場合に paste 関数を使用する。

```
a <- 'AT'
b <- '1G0010020'
paste(a, b, sep = '-')
# [1] "AT-1G0010020"

s <- c('AT', '1G', '0010020')
paste(s, collapse = ';')
# [1] "AT;1G;0010020"

a <- c('AT', 'AT')
b <- c('1G0010', '1G0020')
paste(a, b, sep = '-')
# [1] "AT-1G0010" "AT-1G0020

paste(a, b, collapse = ';')
# [1] "AT 1G0010;AT 1G0020"

paste(a, b, sep = '-', collapse = ';')
# [1] "AT-1G0010;AT-1G0020"
```

文字列分割

区切り文字をもとに 1 つの文字列を複数の文字列に分割するとき strsplit 関数を用いる。分割後の文字列はリストとして返される。

```
x <- c('AT1G01','AT2G02','AT3G03')


y <- strsplit(x, 'T')
y
# [[1]]
# [1] "A"    "1G01"
#
# [[2]]
# [1] "A"    "2G02"
#
# [[3]]
# [1] "A"    "3G03"
```

strsplit 関数の結果をリスト処理用の sapply 関数に代入することで整形できる。 ▶

```
z <- sapply(strsplit(x, 'T'), '[', 2)
z
# [1] "1G01" "2G02" "3G03"
```