2022/7/30

『R プログラミング超入門』

この資料は R プログラミングを始めて学習する入門者あるいは R プログラミングの基礎を簡単に復習したい人向けの演習題材となっている。書籍『R ではじめるケモ・マテリアルズ・インフォマティクス』の第 2.1 節を読んだ後に取り組むのがベストであるが、第 2 章学習後に復習を兼ねて学習してもらってもよい。

目次

例題 2.A.1　スカラー変数と基本演算

まず R を電卓代わりに利用できるようになろう。そのために、スカラー変数を定義し基本的な演算とよく利用する組み込み関数(自分でプログラミングしないで関数を呼び出すだけで利用できる関数のこと)をまず学ぶ。スカラー変数は 1 つの変数名に一つのデータ値を割り当てる場合に利用される。スカラー変数を定義し、四則演算の仕方および組み込み関数の利用方法に慣れよう。

[プログラム 2_A_1]　Rbasic01.R

```
#1 definition of variables
x <- 3; y <- 4; z <- 5
x; y; z
#2 four arithmetic operation
a <-(x+y)*z+(x-y)/z
a;
#3 square root & power
b1 <- sqrt(x); b2 <- x^0.5
b1; b2
#4 Pi & trigonometric functions
c <- pi/3
c1 <- sin(c); c2 <- cos(c); c3 <- tan(c)
pi; c1; c2; c3
#5 exponential and logarithmic functions
d <- exp(1)
d1 <- log(d)
d2 <- log10(d)
d; d1; d2
#6 absolute value & truncation
e <- -2.5
e1 <- abs(e)
e2 <- trunc(e)
e1; e2
```

プログラム 2_A_1 の出力結果

```
> #1 definition of variables
```

```
> x <- 3; y <- 4; z <- 5
> x; y; z
[1] 3
[1] 4
[1] 5
> #2 four arithmetic operation
> a <-(x+y)*z+(x-y)/z
> a;
[1] 34.8
> #3 square root & power
> b1 <- sqrt(x); b2 <- x^0.5
> b1; b2
[1] 1.732051
[1] 1.732051
> #3 Pi & trigonometric functions
> c <- pi/3
> c1 <- sin(c); c2 <- cos(c); c3 <- tan(c)
> pi; c1; c2; c3
[1] 3.141593
[1] 0.8660254
[1] 0.5
[1] 1.732051
> #4 exponential and logarithmic functions
> d <- exp(1)
> d1 <- log(d)
> d2 <- log10(d)
> d; d1; d2
[1] 2.718282
[1] 1
[1] 0.4342945
> #5 absolute value & truncation
> e <- -2.5
> e1 <- abs(e)
> e2 <- trunc(e)
> e1; e2
[1] 2.5
```

```
[1] -2
```

R のプログラムでは他のプログラミング言語のように先頭や最後に宣言文は不要である。このプログラムの 1 行目には先頭に”#”の文字が入っているので、この行はすべて注釈(コメント)となる。たまたま 1 行目が注釈行になっているが、1 行目を注釈行にする必要はない。また変数の定義あるいは演算式を書いた後に、同じ行でその後に注釈を入れても構わない。その行の中で”#”以降の部分は注釈部分となる。プログラムの実行単位の区切りごとに”#”を入れておくとメモ代わりになり、またプログラムが見やすくなる。

二行目では”<-“を用いてスカラー変数を定義しその値を指定している。”;”を区切り記号に用いて 3 変数 x,y,z の値を指定する。いくつ指定しても構わないし、変数名前と演算記号の間に空白文字はいくつあっても構わない。1 行の中に複数の実行文を書いて構わないことを意味している。演算の定義が終了しないうちに改行して演算の続きをそのまま入力しても構わない。ただし 1 つの変数名の間に空間文字を入れると 2 つの変数になってしまう。また変数名の中の大文字と小文字は区別されるので勝手に書き換えると別の変数として扱われるので注意が必要である。

3 行目で既に定義した x,y,z の変数名だけを”;”で繋いで実行すると、それらに格納されている内容を表示してくれる。もし定義していない変数名を打ち込むと、エラーメッセージ「オブジェクト＊＊＊がありません。」と表示される。変数名を打ち込む代わりに、print 関数を用いて引数に変数名を入れても画面上に表示することができる。数値の種類には整数、実数、複素数があるが、特別に種類を指定しなければすべて実数として処理される。ある目的のために実数値型と整数値型、あるいは整数値型と文字型との間の変換をしたい場合の方法は 2.A.5 項のデータの型で説明する。

#2 では四則演算を実行している。演算の優先順位は小学校で習った通りである。結果はすべて実数として計算される。#3 の”sqrt”は平方根を計算する関数、”^”は累乗を計算する関数である。#4 にあるように、”pi”と打ち込めば円周率が計算され、”sin”,”cos”,”tan”関数により三角関数の計算ができる。引数は度でなく、ラジアンである。#5 ではまず指数関数”exp”を用いてネイピア数(e)を計算し、その値に対する底が e と 10 の場合の対数関数を計算する。最後に、#6 では”abs”関数を用いて絶対値の計算を、さらに”trunc”関数を用いて整数部分を取り出す計算を実行する。組み込まれている関数でなく自作の関数を作成して利用する場合については書籍の 2.3.4 節で説明する。

自明な計算エラーについてもわかりやすいエラーメッセージが表示される。”1/0”を実行すると、「無限大」を意味する”Inf”が表示され、”sqrt(-1)”を実行すると、「計算結果が NaN になりました」と表示される。”NaN”は”Not a Number”という意味である。

一度作成した変数は rm 関数で削除するまで残り、またその値は書き換えられるまで残っている。すべての変数を完全に削除したい場合は RStudio 画面の右上にある Environment ペインにある「ほうき」マークを選択する。画面左下の Console 画面を「ほうき」マークで消した場合は出力表示画面の内容が消去されるだけで変数とその値は残っている。すなわち、削除しないで同じプログラムを 2 回流すと、2 回目の実行は 1 回目実行時の変数の値を保持した状態から開始されることを覚えておこう。

例題 2.A.2　繰り返し制御と条件分岐

より複雑な計算を実行しようとすると繰り返し制御と条件分岐を利用する必要がある。

[プログラム 2_A_2]　Rbasic02.R

```
#1 for-loop
x1 <- 0; x2 <- 0
for (i in 1:10) {
  if(i%%2 == 1) x1 <- x1 + i
  else x2 <- x2 + i
}
x1; x2
#2 while-loop
y1 <- 0; y2 <- 0; i <- 1
while (i<=10) {
  if(i%%2 != 0) y1 <- y1 + i
  else y2 <- y2 + i
  i <- i + 1
}
y1; y2
```

プログラム 2_A_2 の出力結果

```
> #1 for-loop
> x1 <- 0; x2 <- 0
> for (i in 1:10) {
+   if(i%%2 == 1) x1 <- x1 + i
+   else x2 <- x2 + i
+ }
> x1; x2
[1] 25
[1] 30
> #2 while-loop
> y1 <- 0; y2 <- 0; i <- 1
> while (i<=10) {
+   if(i%%2 != 0) y1 <- y1 + i
```

```
+   else y2 <- y2 + i
+   i <- i + 1
+ }
> y1; y2
[1] 25
[1] 30
```

この例題では”for“文あるいは”while”文を繰り返し制御に、また”if”文を条件分岐に利用する。このプログラムは 1 から 10 までの自然数の中で、奇数部分の和および偶数部分の和を計算し、x1 と x2、y1 と y2 として表示する。

プログラム前半の“for”文の書式は”for (i in 1:10) {・・・}”のようになる。i を 1 から 1 ずつ増分し 10 になるまで”・・・”の演算を繰り返す。このプログラムの演算部分はさらに”if・・・else”文の構造になっている。i が奇数ならば x1 に i を加え、そうでなければ x2 に i を加える。「i が奇数ならば」と言う条件は i/2 の割り算の余りが 1 ということと同じなので、その条件文を”i%%2 == 1”で表現している。”%%”は割り算の余りの計算を、”==”は「等しい」という条件を意味する。”else”の後の演算は i が奇数でない場合に実行され、偶数の i は x2 に積算される。

プログラム後半は同じ計算を”while”文を利用して実行する。”while”文の書式は”while (i<=10)”のようになる。この行の直前で i の値は 1 にセットされ、”while”文の最後に”i <- i + 1”があるので、i が 1 からスタートし 1 ずつ増分して 10 を超えた時点で、その時の”while”文の内容を実行せずに終了することになる。”if・・・else”文の構造は”for”文と同じであるが、ここでは”i%%2 != 0”を用いている。”==”は「等しい」という条件であったが、”!=”は「等しくない」という条件になっている。

例題 2.A.3　スカラーおよびベクトルの作成と簡単な操作

ここまででは 1 個の数値データを 1 個のスカラーの変数として取り扱ったが、これ以降は複数のデータをまとめて処理するためのベクトルと行列を学ぶ。さらに実数型データ以外のデータの型やそれらの相互変換についても学習する。まずは実数型のスカラー、ベクトル、行列の違いを理解しよう。スカラーは 1 個の数値、ベクトルは複数の数値データを横あるいは縦に一列に並べた 1 次元の配列、行列は複数の数値データを長方形の升目状に並べた 2 次元の配列であるということができる。

まず、ベクトルを作成してみよう。

[プログラム 2_A_3]　Rbasic03.R

```
#1 scalar
x1 <- 10
x1
#2 vector
y1 <- numeric(5)
y2 <- c(2,4,6,8,10)
y3 <- 1:5
y4 <- seq(1,90,by=10)
y5 <- rep(2,10)
y6 <- c(y1,y2,y3)
y1; y2; y3; y4; y5; y6
length(y6)
y1[] <- 3
y1
y3[5]
y3[5] <- 6
y3
names(y3) <- c("1st","2nd","3rd","4th","5th")
y3
names(y3[3])
names(y3) <- NULL
y3
```

プログラム 2_A_3 の出力結果

```
> #1 scalar
> x1 <- 10
> x1
[1] 10
> #2 vector
> y1 <- numeric(5)
> y2 <- c(2,4,6,8,10)
> y3 <- 1:5
> y4 <- seq(1,90,by=10)
> y5 <- rep(2,10)
> y6 <- c(y1,y2,y3)
> y1; y2; y3; y4; y5; y6
[1] 0 0 0 0 0
[1]  2  4  6  8 10
[1] 1 2 3 4 5
[1]  1 11 21 31 41 51 61 71 81
 [1] 2 2 2 2 2 2 2 2 2 2
 [1]  0  0  0  0  0  2  4  6  8 10  1  2  3  4  5
> length(y6)
[1] 15
> y1[] <- 3
> y1
[1] 3 3 3 3 3
> y3[5]
[1] 5
> y3[5] <- 6
> y3
[1] 1 2 3 4 6
> names(y3) <- c("1st","2nd","3rd","4th","5th")
> y3
1st 2nd 3rd 4th 5th
  1   2   3   4   6
> names(y3[3])
[1] "3rd"
> names(y3) <- NULL
> y3
```

```
[1] 1 2 3 4 6
```

2 行目のように x1 という変数に 10 という 1 つの数値を代入すると x1 は自動的にスカラー変数と定義される。ベクトル変数を定義するためには、5 行目から 10 行目にあるような 6 種類の作成方法を実行する。

① ”numeric(5)”は 5 個の要素を持つベクトル y1 を作成する。要素の値を指定していないので 5 個の要素には自動的に 0 が格納される。

② ”c(2,4,6,8,10)”の c 関数は定数、文字、変数等を結合する関数であり、ここでは 2,4,6,8,10 という 5 個の実数値を 5 個の要素とするベクトル y2 を作成する。関数の名称”c”は Combine Values into a Vector or List に由来する。

③ ”1:5”は 1 から 5 まで 1 ずつ増分した 5 個の実数値を要素とするベクトル y3 を作成する。

④ ”seq(1,90.by=10)”は 1 からスタートし 10 ずつ増分して 90 に達するまで、すなわち 9 個の実数値を要素とするベクトル y4 を作成する。関数の名称”seq”は Sequence Generation に由来する。

⑤ ”rep(2,10)”は 2 を 10 個繰り返した実数値を要素とするベクトル y5 を作成する。関数の名称”rep”は Replicate Elements of Vectors and List に由来する。

⑥ ”c(y1,y2,y3)”は y1,y2,y3 の 3 個のベクトルの要素を繋いだベクトル y6 を作成する。c 関数の引数はスカラーとベクトルが混在していても構わない。

作成したベクトルの内容を確認したい場合はその変数名を打ち込めば内容を表示してくれる。もしベクトルの長さ(要素の個数)を知りたい場合は length 関数を実行すればよい。R 言語ではベクトルの添え字(何番目のデータかを示すインデックス)は 1 から始まる。Python のように 0 から始まらないことに注意しよう。

既に作成している y1 のすべての要素を 3 で置き換えたい場合は”y1[] <- 3” を実行する。ここで、”y1 <- 3”としてしまうと、せっかく 5 つの成分で構成されるベクトル変数として定義した y1 が 1 成分のスカラー変数に書き換えられてしまうので注意が必要である。一つの要素、例えばベクトル y3 の 5 番目の要素だけを表示する場合は”y3[5]”を、またベクトル y3 の 5 番目の要素だけを 6 に置き換えたい場合は”y3[5] <- 6”を実行する。ベクトルの添え字(要素の番号)を指定する場合は”()”でなく”[]”(角かっこ)を指定する点に注意する。

さらに、R プログラミングでは names 関数を用いて各要素の場所を識別するためのラベル(names 属性と呼ばれる)を付けることもできる。例題中の names 関数を実行している行では、”1st”から”5th”までの 5 個の文字列を y3 の 5 個の要素ラベルとして定義する。今回初めて「実数値」でなく「文字列」が出てきたが、()中の文字が””で囲まれているかどうかで文字列か実数値かが識別される。例えば「c(5)」の 5 は実数値であり、「c(“5”)」の 5 は文字列である。実数値であれば四則演算等を目的としているが、文字列の場合は識別用の名前が主な用途であり、同じグループ(カテゴリーとも呼ぶ)に属するデータを識別する、あるいはグラフ上に記号として表示させる用途で利用する。ラベル付きのベクトル y3 の 3 番目のラベルのみ知りたい、あるいは表示させたい場合は”names(y3[3])を打ち込めばよい。もしラベルを削除したい場合は names 関数に”NULL”(「空」という意味)を代入すればよい。ベクトルからデータそのものを取り出す・格納する場合とラベル名を取り出す・格納する場合は異なることにも注意しよう。

例題 2.A.4　行列の作成と簡単な操作

　行列を作成し簡単な操作を実行しよう。

[プログラム 2_A_4]　Rbasic04.R

```
#3 matrix
z1 <- matrix(0,nrow=3,ncol=3)
z1
z2 <- matrix(c(1,3,2,4,5,6,9,8,7),nrow=3,ncol=3)
z2
z2 <- matrix(c(1,3,2,4,5,6,9,8,7),nrow=3,ncol=3,byrow=TRUE)
z2
y1 <- c(1,3)
y2 <- c(2,4,5,6)
y3 <- c(9,8,7)
z3 <- matrix(c(y1,y2,y3),nrow=3,ncol=3)
z3
z4 <- matrix(c(y1,y2,y3),nrow=3,ncol=3,byrow=TRUE)
z4
dim(z4);nrow(z4);ncol(z4)
z4[2,3]
z5 <- z4[2,]
z5
z6 <- z4[,2]
z6
z7 <- z4[2,,drop=FALSE]
z7
z8 <- z4[,2,drop=FALSE]
z8
rownames(z4) <- c("row-1","row-2","row-3")
colnames(z4) <- c("col-1","col-2","col-3")
z4
rownames(z4) <- NULL
colnames(z4) <- NULL
z4
```

プログラム 2_2_2 の出力結果

```
> #3 matrix
> z1 <- matrix(0,nrow=3,ncol=3)
> z1
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
> z2 <- matrix(c(1,3,2,4,5,6,9,8,7),nrow=3,ncol=3)
> z2
     [,1] [,2] [,3]
[1,]    1    4    9
[2,]    3    5    8
[3,]    2    6    7
> z2 <- matrix(c(1,3,2,4,5,6,9,8,7),nrow=3,ncol=3,byrow=TRUE)
> z2
     [,1] [,2] [,3]
[1,]    1    3    2
[2,]    4    5    6
[3,]    9    8    7
> y1 <- c(1,3)
> y2 <- c(2,4,5,6)
> y3 <- c(9,8,7)
> z3 <- matrix(c(y1,y2,y3),nrow=3,ncol=3)
> z3
     [,1] [,2] [,3]
[1,]    1    4    9
[2,]    3    5    8
[3,]    2    6    7
> z4 <- matrix(c(y1,y2,y3),nrow=3,ncol=3,byrow=TRUE)
> z4
     [,1] [,2] [,3]
[1,]    1    3    2
[2,]    4    5    6
[3,]    9    8    7
```

```
> dim(z4);nrow(z4);ncol(z4)
[1] 3 3
[1] 3
[1] 3
> z4[2,3]
[1] 6
> z5 <- z4[2,]
> z5
[1] 4 5 6
> z6 <- z4[,2]
> z6
[1] 3 5 8
> z7 <- z4[2,,drop=FALSE]
> z7
     [,1] [,2] [,3]
[1,]    4    5    6
> z8 <- z4[,2,drop=FALSE]
> z8
     [,1]
[1,]    3
[2,]    5
[3,]    8
> rownames(z4) <- c("row-1","row-2","row-3")
> colnames(z4) <- c("col-1","col-2","col-3")
> z4
      col-1 col-2 col-3
row-1     1     3     2
row-2     4     5     6
row-3     9     8     7
> rownames(z4) <- NULL
> colnames(z4) <- NULL
> z4
     [,1] [,2] [,3]
[1,]    1    3    2
[2,]    4    5    6
[3,]    9    8    7
```

行列の作成方法はベクトルの作成方法と類似している。ここでは頻繁に利用する 4 通りの作成方法を実行する。

① matrix 関数を用いて 3x3 の行列 z1 を作成し初期値としてすべての要素に”0”を設定する。引数である nrow と ncol は行数(縦方向に並ぶ数)と列数(横方向に並ぶ数)を指定する。片方の引数だけ指定してある場合はもう一方は要素数から自動的に算出される。”nrow=”と”NCOL=”の名前を省略して数字だけ書いてもよいが、慣れるまでは省略しない方が間違えなくてよい。

② matrix 関数の第一引数として 9 個の要素の数値を指定して 3x3 の行列 z2 を作成する。小さなサイズの行列を作成する場合は簡単で確実な方法である。それぞれの要素を行列に配置していく順番は、デフォルトでは 1 列目、2 列目、3 列目の順番(縦方向に詰める)となる。もし 1 行目、2 行目、3 行目の順番(横方向に詰める)に配置したい場合は引数”byrow=TRUE”を追加してやればよい。

③ 3 つのベクトル y1,y1,y3 を始めに作成し、それらを並べて 3x3 の行列 z3 を作成する。並べるベクトルの数と大きさは、合計の要素数が作成したい行列の要素数にさえ合えば任意で構わない。要素が配置される順番は②と同じであり、引数”byrow=TRUE”を追加してやれば行列 z4 のように順番を変えられる。

作成した行列の行数と列数を調べる場合は dim 関数を、行数あるいは列数の一方だけ知りたい場合は nrow 関数あるいは ncol 関数を利用する。一方、特定の要素の内容を調べる場合、例えば行列 z4 の 2 行 3 列目の要素の場合は”z4[2,3]”を打ち込めばよい。また 2 行目のすべて要素あるいは 2 列目のすべての要素を調べる場合は、”z4[2,]”あるいは”z4[,2]”の表記を用いて、調べたい行の番号あるいは列の番号だけ指定するとよい。Z5,z6 のような形でベクトルに変換されて表示される。Z5 あるいは z6 のようなデータ列を作成する時に、ベクトルに変換するのではなく元の行列の形を保持したい場合がある。その場合は引数”drop=FALSE”を追加することによって、要素 3 のベクトルではなく、1x3 の行列 z7 あるいは 3x1 の行列 z8 にすることができる。

ベクトルにラベルを付けることができたように、行列にも行ラベルおよび列ラベルを付けることができる。Rownames 関数と colnames 関数を使えばよい。一度付けたラベルを削除する場合は、両方の関数に”NULL”を代入する点はベクトルの場合と同じである。

例題 2.A.5　データの型

　よく利用されるデータの型には、実数型、整数型、文字型、因子型、論理型、複素数型、因子型がある。各種のデータの型を区別して使用できるようになろう。

[プログラム 2_A_5]　Rbasic05.R

```
#1 numeric
x1 <- c(1.1, -1.1)
x1; is.numeric(x1); x1[1]+x1[2]
#2 integer
x2 <- as.integer(x1)
x2; is.integer(x2); x2[1]+x2[2]
#3 matrix
x3 <- matrix(1,nrow=3,ncol=3)
x3; is.matrix(x3)
#4 character
x4 <- as.character(x1)
x4; is.character(x4); x4[1]+x4[2]
#5 logical
x5 <- c(1 < 2, 1 == 2)
x5[1]; is.logical(x5[1])
x5[2]; is.logical(x5[2])
#6 complex
x6 <- c(1+2i)
x6; is.complex(x6)
#7 check of data type
class(x3)
#8 overwriting data
x8 <- c(1)
x8; class(x8)
x8 <- c("abc")
x8; class(x8)
#9 conversion of data type
x9 <- c(1)
x9; class(x9)
x9 <- as.character(x9)
```

```
x9; class(x9)
x9 <- as.numeric(x9)
x9; class(x9)
#10 factor
x10 <- c("B","A","A","C","B","C","C","B","A","A")
x10; class(x10)
x10 <- factor(x10)
x10; class(x10)
levels(x10)
table(x10)
```

まず、#1 の numeric の部分を実行すると、以下が出力される。

```
> #1 numeric
> x1 <- c(1.1, -1.1)
> x1; is.numeric(x1); x1[1]+x1[2]
[1]  1.1 -1.1
[1] TRUE
[1] 0
```

c 関数を用いて 2 個の数値を x1 に格納する場合、データの型を指定しなければ x1 は自動的に実数型になる。X1 の内容を表示させた後、is.numeric 関数で x1 が実数型かどうかをチェックすると、”TRUE”が表示されているので実数型である。そうでない場合は”FALSE”が表示される。実数型は四則演算ができるので、最後に x1 の 1 番目と 2 番目の要素の和を” x1[1]+x1[2]”により計算した。

次に整数型を扱う場合を#2 で見てみよう。

```
> #2 integer
> x2 <- as.integer(x1)
> x2; is.integer(x2); x2[1]+x2[2]
[1]  1 -1
[1] TRUE
[1] 0
```

X1 は実数型の数値として定義しているが、as.integer 関数を用いて整数型に変換することができ、変換後のデータを x2 にコピーする。is.integer 関数で整数型かどうかを確認すると、”TRUE”が表示される。実数型と同じで足し算もでき、x2[1]+x2[2]の結果として”0”が出力される。

is.xxx(xxx は”numeric”,” integer”,” vactor”,” matrix”,” character”,” factor”,” logical”等)のように表記される関数はデータの型以外にデータの構造およびデータの内容も表示してくれる。例題2.4 に出てきた行列(matrix)であるかどうかは#3 に示すように is.matrix 関数で確認することができる。is.matrix 関数を用いて X3 が行列かどうか確認すると、” TRUE”が返ってくる。

```
> #3 matrix
> x3 <- matrix(1,nrow=3,ncol=3)
> x3; is.matrix(x3)
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
[1] TRUE
```

数値以外のデータの型として文字型と論理型がある。#4 では実数型データ x1 を as.character 関数により文字型データ z4 に変換している。” x4”を打ち込んで内容を表示させた時の値は数字(1.1 あるいは-1.1)に見えるが、” ”で囲まれていることに注意してもらいたい。これは文字型データになっていることを意味している。文字型データである X4 の第 1 と第 2 の要素の足し算を実行すると、「数値ではありません」というエラーメッセージが表示される。

```
> #4 character
> x4 <- as.character(x1)
> x4; is.character(x4); x4[1]+x4[2]
[1] "1.1"  "-1.1"
[1] TRUE
 x4[1] + x4[2] でエラー:  二項演算子の引数が数値ではありません
```

論理型データとは、例えば変数同士あるいは数式同士が等しいかどうか、あるいは関係式の大小関係等を判定した結果、TRUE（真）あるいは FALSE(偽)の値を持つ変数のことである。論理型データであるかどうかは is.logical 関数で確かめることができる。#5 では①1 より 2 が大きい、②1 と 2 が等しいという 2 つの論理の判定結果を x4 の第 1 および第 2 要素として格納する。第 1 の要素の値は「真(“TRUE”)」であり、データの型は「論理型(“TRUE”)」であることがわかる。また第 2 の要素の値は「偽(“FALSE”)」であり、データの型は「論理型(“TRUE”)」であることがわかる。

```
> #5 logical
> x5 <- c(1 < 2, 1 == 2)
```

```
> x5[1]; is.logical(x5[1])
[1] TRUE
[1] TRUE
> x5[2]; is.logical(x5[2])
[1] FALSE
[1] TRUE
```

複素数型についても調べてみよう。実数部分は実数のままで、虚数部分は虚数単位”i”を加えた形式で表現される。#6 の例の x5 は 1+2i となっている。is.complex 関数で複素数かどうかを確認でき、x6 を定義したのと同じ形式で表示される。

```
> #6 complex
> x6 <- c(1+2i)
> x6; is.complex(x6)
[1] 1+2i
[1] TRUE
```

同じ変数名でデータを上書きするとデータの内容だけでなく、データの型と要素の数も更新され、現時点のデータ属性(行列かどうか、型は何か)がわからなくなる時がある。その場合は class 関数が利用できる。その他、typeof 関数、mode 関数、str 関数も目的に応じて利用できる。#7 で示すように、データ X3 は行列”matrix”であり、また配列”array”でもあることがわかる。配列についてはこの後の例題 2.6 で説明する。X8 は最初に実数型データであったが、上書きされた後は文字型データに変換されていることがわかる。X9 は最初に実数型データとして定義されるが、as.character 関数により文字型データに変換され、最後は as.numeric 関数により実数型データに戻っている。

```
> #7 check of data type
> class(x3)
[1] "matrix" "array"
> #8 overwriting data
> x8 <- c(1)
> x8; class(x8)
[1] 1
[1] "numeric"
> x8 <- c("abc")
> x8; class(x8)
[1] "abc"
```

```
[1] "character"
> #9 conversion of data type
> x9 <- c(1)
> x9; class(x9)
[1] 1
[1] "numeric"
> x9 <- as.character(x9)
> x9; class(x9)
[1] "1"
[1] "character"
> x9 <- as.numeric(x9)
> x9; class(x9)
[1] 1
[1] "numeric"
```

最後に、特殊なデータの型である因子型を説明する。例えば、「魚」、「肉」、「野菜」とか、「赤」、「青」、「黄」とか、カテゴリーに分けてデータを管理したい場合に因子型データを利用すると便利である。#10 の例は 10 個のデータが A,B,C の 3 種類のいずれかに分類できる場合である。X10 にデータをランダムに入れた後、levels 関数を用いていくつのカテゴリー(この例では A,B,C の 3 種類)になっているかを調べることができる。さらに table 関数によりそれぞれのカテゴリーに属する要素の数を調べることができる。

```
> #10 factor
> x10 <- c("B","A","A","C","B","C","C","B","A","A")
> x10; class(x10)
 [1] "B" "A" "A" "C" "B" "C" "C" "B" "A" "A"
[1] "character"
> x10 <- factor(x10)
> x10; class(x10)
 [1] B A A C B C C B A A
Levels: A B C
[1] "factor"
> levels(x10)
[1] "A" "B" "C"
> table(x10)
x10
A B C
```

4 3 3

例題 2.A.6　データの構造

　これまでデータ構造にはスカラー、ベクトル、行列があることを学んだが、それらをさらに拡張した配列がある。また R 言語の特長でもあるリスト、データフレームと呼ばれるデータ構造も良く利用される。それらの使い方を見てみよう。

[プログラム 2_A_6]　Rbasic06.R

```
#1 array
x1a <- matrix(1:9,nrow=3,ncol=3)
x1a
x1b <- array(1:8, dim = c(2,2,2))
x1b
class(x1a); class(x1b)
#2 list
x2a <- list(c(1,2,3),"name",x1a,TRUE)
x2a
class(x2a); length(x2a); sapply(x2a,mode)
x2a[[1]]; x2a[[2]]; x2a[[3]]; x2a[[4]]
x2a[[1]][[3]]; x2a[[3]][2,3]
x2b <- list(list1=c(1,2,3),list2="name",list3=x1a,list4=TRUE)
x2b
x2b[["list1"]][[3]]; x2b[["list3"]][2,3]
x2b$list1[[3]]; x2b$list3[2,3]
#3 data frame
x3 <- data.frame(
  product = c("A", "B", "C", "D"),
  line = c("line-1","line-2", "line-2", "line-1"),
  quantity = c(1, 2, 3, 4)
)
x3
str(x3)
x3$quantity
x3[2,]
x3$quantity[3]; x3[[3]][3]; x3[3,3]
x3$line <- as.factor(x3$line)
str(x3)
```

ベクトルとマトリックスはそれぞれ 1 次元と 2 次元のデータ構造を持っているが、任意の次元のデータ構造を array により定義することができる。#1 の部分では matrix 関数を用いて 3x3 の 2 次元構造のデータを x1a の変数名で定義した後、array 関数を用いて 2x2x2 の 3 次元構造のデータを x1b の変数名で定義する。class 関数を用いると、x1a は matrix および array として、また x1b は array として扱われていることがわかる。データを表示させると以下のようになる。

```
> #1 array
> x1a <- matrix(1:9,nrow=3,ncol=3)
> x1a
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> x1b <- array(1:8, dim = c(2,2,2))
> x1b
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

> class(x1a); class(x1b)
[1] "matrix" "array"
[1] "array"
```

#2 ではリスト構造のデータの作成および取り出し方法を示す。リスト構造は異なる型のデータをひとまとめに取り扱うことができ、柔軟な構造を有している。データ X2b はベクトル、文字、行列、論理の 4 種類のデータを含む。class 関数を用いてリスト型のデータがあることを確認でき、length 関数によりデータの長さが 4 であること、sapply 関数(最初の引数は変数名、2 番目の引数 mode はデータの型を調べることを意味する)によりデータ構造を調べることができる。数値型のベクトルあるいは行列データ

は”numeric”と表示される。sapply 関数の使い方については書籍の 2.3.5 項で学ぶ。

```
> #2 list
> x2a <- list(c(1,2,3),"name",x1a,TRUE)
> x2a
[[1]]
[1] 1 2 3

[[2]]
[1] "name"
[[3]]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

[[4]]
[1] TRUE
> class(x2a); length(x2a); sapply(x2a,mode)
[1] "list"
[1] 4
[1] "numeric"   "character" "numeric"   "logical"
```

自分でリスト構造のデータを作成する必要性をあまり感じないかもしれないが、書籍中に出てくる各種の高度な統計あるいは機械学習の関数の計算結果はリスト構造になっているケースが多い。そのため計算結果を解析するためにはリスト構造のデータから特定のデータを取り出す方法を知っていなければならない。ベクトル、行列、リストのいずれの場合も要素番号を指定する時には角かっこを使う。ベクトル x の 3 番目の要素を取り出す場合は”x[3]”となり、行列 y の 2 行 3 列目の要素を取り出す場合は”y[2,3]”となる。一方、リスト構造のデータ x2 の 1 番目から 4 番目までの要素を順番に取り出すためには例題中の”x2a[[1]]; x2a[[2]]; x2a[[3]]; x2a[[4]]”のように、二重の角かっこを使う必要がある点に注意しよう。要素の中のさらに何番目の要素かまで指定する場合は角かっこを横に 2 つ繋げる。X2a の最初の要素(ベクトル)の 3 番目の要素であれば”x2a[[1]][[3]]”とする。あるいは”x2a [[1]][3]” でもよい。X2a の 3 番目の要素(行列)の 2 行 3 列の要素であれば”x2a [[3]] [2,3]” のようにする。データが階層的な構造になっていることを意識して取り出す必要がある。

```
> x2a[[1]]; x2a[[2]]; x2a[[3]]; x2a[[4]]
[1] 1 2 3
[1] "name"
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
[1] TRUE
> x2a[[1]][[3]]; x2a[[3]][2,3]
[1] 3
[1] 8
```

リスト構造のデータのそれぞれの要素に名前を付けると、何番目かの要素であるかを指定する代わりに要素の名前を使ってデータを取り出すことができる。例題の x2b はその場合の利用例である。要素の名前を”list1”,”list2”,”list3”としている。

```
> x2b <- list(list1=c(1,2,3),list2="name",list3=x1a,list4=TRUE)
> x2b
$list1
[1] 1 2 3
$list2
[1] "name"
$list3
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
$list4
[1] TRUE
> x2b[["list1"]][[3]]; x2b[["list3"]][2,3]
[1] 3
[1] 8
> x2b$list1[[3]]; x2b$list3[2,3]
[1] 3
[1] 8
```

最後に、R の標準的なデータ構造であるデータフレームを見てみよう。異なる型のデータをひとまとめに取り扱くことができる点ではリスト構造と同じであるが、各要素の長さ(ベクトルの長さ)は同じでなければならない。各要素の長さが同じなのでそれぞれのデータが対応の取れる特性を持っていることになり、表形式(excel のテーブルと同じ)で取り扱うことができる。#3 の例では、x3 の最初の要素は製品名(文字型でータ)、2 番目の要素は製造ライン名(文字型データ)、3 番目は生産数(実数型データ)を表している。data.frame 関数を用いて定義し、引数はリスト関数と同じ形式である。この例では各列のデータが何を意味しているかわかるように各要素にラベルを付けている。”x3”を打ち込むとデータの内容だけ表示されるが、str 関数を用いるとデータの型等の追加情報も見ることができる。

```
> #3 data frame
> x3 <- data.frame(
+   product = c("A", "B", "C", "D"),
+   line = c("line-1","line-2", "line-2", "line-1"),
+   quantity = c(1, 2, 3, 4)
+ )
> x3
  product   line quantity
1       A line-1        1
2       B line-2        2
3       C line-2        3
4       D line-1        4
> str(x3)
'data.frame':      4 obs. of  3 variables:
 $ product : chr  "A" "B" "C" "D"
 $ line    : chr  "line-1" "line-2" "line-2" "line-1"
 $ quantity: num  1 2 3 4
```

各要素のデータを取り出す場合は”x3”で表示されるテーブルを行列とみなすとわかりやすい。3 列目の”quantity”のデータをすべて取り出す場合は、”x3$quantity”あるいは”x3[,3]”とすればよい。ラベルを用いる場合は”$”で繋げる必要がある。3 列目を指定し、すべての行のデータを取り出す場合は”x3[,3]”のように行に対応する数値は省略し列数の 3 だけ指定すればよい。さらに、行と列の両方を指定してデータを取り出す場合は、” x3$quantity[3]; x3[[3]][3]; x3[3,3]”のいずれかの表記を使う。”x3$quantity[3]”は quantity の列の 3 行目である。”x3[[3]][3]”の最初の[[3]]は 3 列目を、次の[3]は 3 行目を意味する。一方、” x3[3,3]”の最初の 3 は 3 行目を、次の 3 は 3 列目を意味するので、”x3[[3]][3]”と比べて行と列が逆になっていることに注意する必要がある。

```
> x3$quantity
[1] 1 2 3 4
> x3[2,]
  product   line quantity
2       B line-2        2
> x3$quantity[3]; x3[[3]][3]; x3[3,3]
[1] 3
[1] 3
[1] 3
```

ある列のデータ全部を因子型に変更して利用したい場合は as.factor 関数を用いて変換することもできる。#3 の最後は line の列を因子型に変更した例である。

```
> x3$line <- as.factor(x3$line)
> str(x3)
'data.frame':      4 obs. of  3 variables:
 $ product : chr  "A" "B" "C" "D"
 $ line    : Factor w/ 2 levels "line-1","line-2": 1 2 2 1
 $ quantity: num  1 2 3 4
```