GNU R

プログラミング入門

第 0.4 版

中村勝則

2022 年 8 月 30 日

免責事項

本書の内容は参考的資料であり，掲載したプログラムリストは全て試作品である．本書の使用に伴って発生した不利益，損害の一切の責任を筆者は負わない．

目 次

1　はじめに

GNU R は統計解析とデータの可視化を主たる目的として開発されたプログラミング言語処理系とプログラム開発環境であり，その源流は AT&T ベル研究所で開発された **S 言語**[1] である．GNU R は S 言語との互換性を意識して設計されており，その特徴が随所に見られる．GNU R は「**R 言語**」あるいは単に「**R**」と呼ばれることも多い．GNU R はオークランド大学の Ross Ihaka と Robert Clifford Gentleman により開発され，現在では R Development Core Team により保守されており，プログラム本体と関連文書は公式インターネットサイト

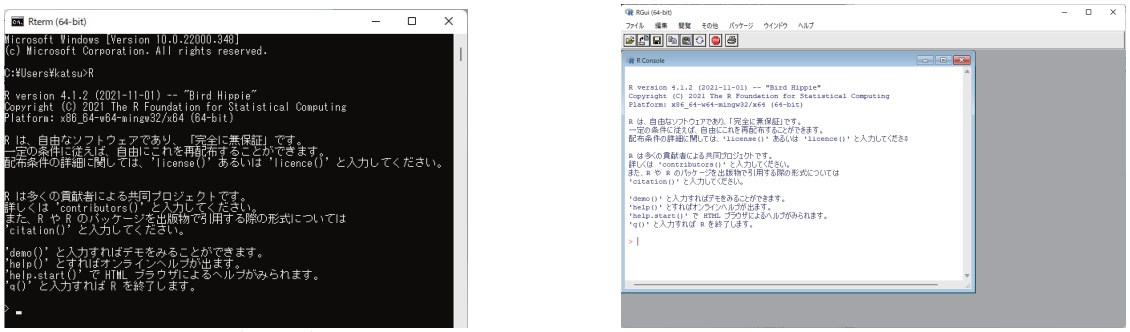
https://www.r-project.org/

から入手することができる．GNU R の入手とインストールの方法に関しては巻末付録 A 「R の入手先とインストール方法」（p.104）を参照のこと．GNU R はオープンソースのフリーソフトウェアである．

本書では簡単のために，基本的に GNU R を「R」と表記する．

1.1　R の構成と起動方法

R は，中核となる言語処理系と GUI 形式の開発環境の 2 つから成るシステムであると見ることができる．R 言語処理系を OS のターミナルウィンドウ（OS のシェル）から直接的に起動して使用する（図 1 の (a)）こともできるが，多くの場合は GUI 形式の開発環境[2] （図 1 の (b)）の上で使用する．



(a) Windows のコマンドプロンプトから起動した例　　(b) GUI の開発環境として起動した例

図 1: R を起動したところ

GUI 形式の開発環境を起動するには R のアイコン（図 2）で表示されるランチャーを用いる．これは Windows の場合はスタートメニュー内に配置されている．また macOS の場合はアプリケーションフォルダ内に配置されている．

図 2: R のアイコン

OS のシェルから R を起動するには，処理系のディレクトリをコマンドサーチパスに加えておく必要がある．

1.2　言語処理系の操作

R を起動すると，プログラムや式の入力のためのウィンドウが表示され，そこにプログラムの入力を促すプロンプト「>」が表示される．このプロンプトに続いてプログラムの文や式を入力して Enter キーを押すと実行結果や評価結果が表示される．

例. 単純な数値計算の実行

```
> 1 + 2    Enter        ←計算式の入力
[1] 3                   ←計算結果の表示
```

[1] 最終的な実装は **S-PLUS** である．

[2] RStudio

この実行例にあるように，計算結果の表示の先頭に「[1]」と表示されているが，これは計算結果のベクトルの要素の位置を示すものである．これに関しては後で詳しく説明する．

R を終了するにはプロンプトに続いて q() と入力する．

例. R の終了（Windows の場合）

```
> q()  Enter                          ←終了処理
作業スペースを保存しますか? [y/n/c]:
```

この例に示すように，q() を実行すると「作業スペースを保存しますか?」と表示される．これに続いて y と入力して Enter を押すと作業内容（入力した文や式）がカレントディレクトリの .Rhistory ファイル[3] に保存されて R が終了する.（n と入力した場合は保存されない） この .Rhistory は後で説明する source 関数を使用して R システムに再度読み込んで実行することができる．

1.2.1 テキストファイルに記述したプログラムを読み込んで実行する方法

関数 source を使用すると，テキストファイルに記述された R のプログラムを読み込んで実行することができる．

書き方： source(プログラムファイルのパス)

「プログラムファイルのパス」[4] が示すファイルからプログラムを読み込んで実行する．例えば次のような R プログラムのファイル ‘test01.r’ がカレントディレクトリにあるとする．

プログラムファイル：test01.r

```
1 print("プログラムファイル 'test01.r' を読み込んで実行します．")
```

このファイルを読み込んで実行する例を次に示す．

例. プログラムの読み込みと実行

```
> source("test01.r")  Enter     ←読み込みと実行
[1] "プログラムファイル 'test01.r' を読み込んで実行します．"      ←実行結果
```

上の例のように，プログラムファイルには拡張子「.r」もしくは「.R」を付けることが慣例であるが,「.txt」など他のものでも良い．

注意）

ターミナルウィンドウに R のプログラム入力して対話的に R を使用する場合は，プロンプトを意味する記号>」が表示されるが，テキストファイルに R のプログラムを記述する場合は，先頭にプロンプトを意味する記号「>」を記述してはならない．

1.2.2 カレントディレクトリの確認と変更

R がファイル入出力の対象とする**カレントディレクトリ**を調べるには関数 getwd を使用する．

例. カレントディレクトリを調べる（Windows での例）

```
> getwd()  Enter
[1] "C:/Users/katsu/R"      ←カレントディレクトリ
```

Windows では，ファイルのパスの表記において，ディレクトリの区切りやルートディレクトリを「¥」で表す[5] が，R では「/」で表現[6] される．

カレントディレクトリを別のディレクトリに移動するには関数 setwd を使用する．

---

[3] macOS 版の R の場合はこのファイルに加えて .Rapp.history にも出力される．

[4] ディレクトリ名，ファイル名から成るファイルの在り処を表す文字列．

[5] あるいはバックスラッシュ「\」で表される．

[6] Linux や macOS でのパスの表記では「/」が用いられる．

例. カレントディレクトリを移動する

```
> setwd("C:/Users/katsu/work")  Enter     ←カレントディレクトリの移動
> getwd()  Enter                ←確認する
[1] "C:/Users/katsu/work"       ←変更されている
```

ファイルシステムのディレクトリに関する処理については，後の「2.3.8 ファイル，ディレクトリに関する処理」（p.53）で更に詳しく解説する．

1.3 本書のサンプルプログラムの掲載の形式

本書に掲載するサンプルプログラムの実行例はほとんどの場合，ターミナルウィンドウ上で対話的に実行した形式で表示する．従って，R の式や文の先頭にプロンプトを意味する記号「>」を表示する．また，長い行の途中で Enter で改行して次の行に継続する場合は，行の継続を意味する記号「+」を次の行の先頭に表示する．

1.4 式や文の記述

1.4.1 複数の行に渡る記述

基本的に R の式や文は 1 行に記述するが，入力途中で改行することもできる．（次の例）

例. 複数の行に渡る式の記述

```
> 2 * 3 * 4 * 5  Enter      ← 2 × 3 × 4 × 5 を計算する式
[1] 120                     ←計算結果
> 2 * 3 *  Enter            ←計算式の途中（*の後ろ）で改行
+ 4 * 5  Enter              ←続きの式（先頭にプラス記号が表示される）
[1] 120                     ←計算結果
```

この例は 2 × 3 × 4 × 5 を計算するものである．1 つ目の入力は式を 1 行に書いたものであるが，2 つ目の式は途中で改行している．ターミナル環境において，式や文の途中で改行すると，この例のように次の行の先頭にプラス記号「+」が自動的に表示され，続きの式や文の入力を受け付ける．この場合のプラス記号は加算の演算子ではないことに注意すること．テキストファイルに R のプログラムを記述する場合は，行を分割する際にこのようなプラス記号を記述してはならない．

1.4.2 複数の式や文を 1 行に記述する方法

セミコロン「;」を使用すると，複数の異なる式や文を 1 行に書き連ねることができる．（次の例）

例. セミコロンで複数の式を書き連ねる

```
> print(1); print(2); print(3)  Enter     ← 3 つの print 関数を 1 行に記述
[1] 1           ←「print(1)」による出力
[1] 2           ←「print(2)」による出力
[1] 3           ←「print(3)」による出力
```

1.4.3 コメントの記述

「#」を記述すると，それ以降（右側）の記述が**コメント**となり，R の式や文とは見なされない．

例. コメントの記述（その 1）

```
> # This is a comment.  Enter     ←コメント（式や文ではない）
>                                 ←何も得られない
```

例. コメントの記述（その 2）

```
> print(1); print(2); # print(3)   Enter      ←途中からコメント
[1] 1                          ←「#」より左側の部分が実行されている
[1] 2
```

1.5 数値の扱い

R で扱う基本的な数値は**整数**（integer 型の値）と倍精度の**浮動小数点数**（double 型の値）である．また特別な操作をしない限り，与えられた数値は浮動小数点数として扱われる．

例. 数値の型を調べる

```
> typeof( 3.14 )   Enter      ←数値 3.14 の型を調べる
[1] "double"                  ← double 型（倍精度浮動小数点数）
> typeof( 3 )   Enter         ←数値 3 の型を調べる
[1] "double"                  ← double 型（倍精度浮動小数点数）
```

これは typeof 関数を使用して数値の型を調べる例で，実行結果からわかるように，小数点以下の記述が無い数値 3 もその型が double となっていることがわかる．数値をあえて整数型に変換するには as.integer を使用する．

例. 数値を整数型に変換する

```
> as.integer( 3 )   Enter      ←数値 3 を整数型の値に変換する
[1] 3                          ←得られた値
> typeof( as.integer( 3 ) )   Enter      ←上記の値の型を調べる
[1] "integer"                          ← integer 型（整数）
```

この他にも，数値表現の末尾に「L」を記すことで明に整数型の数値を与える方法もある．

例. 数値表現の末尾に「L」を記す方法

```
> typeof( 3L )   Enter      ←整数型の 3
[1] "integer"                ← integer 型（整数）
```

R の整数は32 ビット符号付き整数で，扱える値の範囲を超える数値を as.integer で整数に変換することはできない．

例. 大きすぎる数値を整数に変換する試み

```
> 2^35   Enter      ← 2^35（double 型の演算）
[1] 34359738368      ←演算結果
> as.integer( 2^35 )   Enter      ←上記の値を整数型に変換しようとすると…
[1] NA          ←変換結果
警告メッセージ:
NAs introduced by coercion to integer range
```

これは整数として扱えない大きさの値を整数に変換しようとした例である．変換結果として NA が得られており，これは**欠損値**と呼ばれるものである．この NA は数値の 1 つとして扱うべきではなく，プログラムの実装の中では適切に処理（削除や他の値への変換など）する必要がある．

1.5.1 算術演算

R では数値の演算に +，-，*，/ などの演算子（表 1）が使用できる．

表 1: 算術演算をはじめとする基本的な演算

記 述	解　　説	記 述	解　　説	記 述	解　　説
x + y	x と y の加算	x - y	x と y の減算	x * y	x と y の乗算
x / y	x と y の除算	x %/% y	x ÷ y の整数の商	x^y	x の y 乗（x^y）
x %% y	x ÷ y の剰余				

例. 除算，商，剰余

```
> 10 / 6   Enter      ←除算
[1] 1.666667          ←演算結果
> 10 %/% 6  Enter     ←整数の商を求める
[1] 1                 ←商
> 10 %% 6   Enter     ← 10 ÷ 6 の剰余を求める
[1] 4                 ←剰余
```

1.5.2　有効桁数の変更，丸めなどの処理

表 2 に挙げる関数を使用することで，数値の小数部分を削除することができる．

表 2: 小数部の削除に関する関数

関数の記述	解　　説	関数の記述	解　　説
ceiling(値)	「値」の小数部の切り上げ	floor(値)	「値」の小数部の切り下げ
trunc(値)	「値」の小数部の切り捨て		

例. 小数部の切り捨て

```
> trunc( 123.456 )   Enter    ←正の数の小数部の切り捨て
[1] 123                        ←処理結果
> trunc( -123.456 )  Enter    ←負の数の小数部の切り捨て
[1] -123                       ←処理結果
```

例. 小数部の切り上げ

```
> ceiling( 123.456 )   Enter    ←正の数の小数部の切り上げ
[1] 124                          ←処理結果
> ceiling( -123.456 )  Enter    ←負の数の小数部の切り上げ
[1] -123                         ←処理結果（元の数より大きい側の整数が得られている）
```

例. 小数部の切り下げ

```
> floor( 123.456 )   Enter    ←正の数の小数部の切り下げ
[1] 123                        ←処理結果
> floor( -123.456 )  Enter    ←負の数の小数部の切り下げ
[1] -124                       ←処理結果（元の数より小さい側の整数が得られている）
```

表 3 に挙げる関数を使用することで，数値の精度を調整することができる．

表 3: 精度の変更，丸めなどに関する関数

関数の記述	解　　説
round(値,digits=桁数)	「値」の小数部を「桁数」の長さに丸める．
signif(値,digits=桁数)	「値」の有効桁数を「桁数」にする．

例. 小数部の桁数を指定する形の丸め

```
> round( 1234.5678, digits=2 )  Enter      ←小数部を 2 桁に丸める
[1] 1234.57                                ←処理結果
> round( 1234.5678, digits=-2 )  Enter     ←整数部の下 2 桁を 0 にする形で丸める
[1] 1200                                   ←処理結果
```

例. 有効桁数を指定する形の丸め

```
> signif( 1234.5678, digits=5 )  Enter     ←有効桁数を先頭 5 桁に丸める
[1] 1234.6                                 ←処理結果
> signif( 1234.5678, digits=3 )  Enter     ←有効桁数を先頭 3 桁に丸める
[1] 1230                                   ←処理結果
```

1.5.3 数学関数

R では表 4 に示すような数学関数が使用できる.

表 4: R で使用できる数学関数（一部）

関数の記述	解　　説	関数の記述	解　　説	関数の記述	解　　説
`sin(x)`	正弦関数	`cos(x)`	余弦関数	`tan(x)`	正接関数
`asin(x)`	逆正弦関数	`acos(x)`	逆余弦関数	`atan(x)`	逆正接関数
`sinh(x)`	双曲線正弦関数	`cosh(x)`	双曲線余弦関数	`tanh(x)`	双曲線正接関数
`asinh(x)`	逆双曲線正弦関数	`acosh(x)`	逆双曲線余弦関数	`atanh(x)`	逆双曲線正接関数
`log(x)`	$\log_e(x)$	`log10(x)`	$\log_{10}(x)$	`log2(x)`	$\log_2(x)$
`log1p(x)`	$\log_e(1+x)$	`exp(x)`	e^x	`expm1(x)`	$e^x - 1$
`sqrt(x)`	$\sqrt{x}$	`pi`	π（関数ではなく定数）		

例. 円周率 π と正弦関数

```
> pi  Enter      ←円周率
[1] 3.141593     ←値
> sin(pi/2)  Enter      ← sin(π/2)
[1] 1            ←値
```

例. 指数関数と対数関数（その 1）

```
> exp(1)  Enter      ← e
[1] 2.718282         ←値
> log( exp(2) )  Enter      ← log_e e^2
[1] 2                ←値
```

例. 指数関数と対数関数（その 2）

```
> expm1(1)  Enter      ← e − 1
[1] 1.718282           ←値
> log1p( expm1(1) )  Enter      ← log_e{(e − 1) + 1}
[1] 1                  ←値
```

例. 対数関数

```
> log10( 1000 )  Enter      ← log_10 1000
[1] 3                ←値
> log2( 16 )  Enter      ← log_2 16
[1] 4                ←値
```

例. 平方根

```
> sqrt(2)  Enter      ← √2
[1] 1.414214          ←値
```

2　R プログラミング

2.1　データと変数

R では各種の値を**ベクトル**（vector）の形で扱う．R 以外のプログラミング言語では**スカラー**の値とそれらを束ねるデータ構造を別のものとして扱うが，R ではスカラーに見える値も「要素の個数が 1 つであるベクトル」として扱う．R のベクトルは同じ型の要素を束ねるものである．

R では「<-」によって**変数**に値を割り当てることができる．

例. 変数への値の割り当て

```
> a <- 3   Enter          ←変数「a」に値「3」を割り当てる
> a   Enter               ←値の確認
[1] 3                     ←値の表示
```

R は**動的型付け**の言語処理系であり，変数の使用に先立って型の宣言をする必要がない．

変数への値の割り当てには「=」や「->」を使用することもできる．

例. 変数への値の割り当て（その 2）

```
> 4 -> b   Enter          ←変数「b」に値「4」を割り当てる
> b   Enter               ←値の確認
[1] 4                     ←値の表示
> c = 5   Enter           ←変数「c」に値「5」を割り当てる
> c   Enter               ←値の確認
[1] 5                     ←値の表示
```

R での変数への値の割り当ては「<-」を用いるのが最も標準的である．

値が割り当てられていない変数を参照するとエラーとなる．

例. 値が割り当てられていない変数 x の参照

```
> x   Enter               ←値が割り当てられていない変数 x
エラー:  オブジェクト 'x' がありません        ←エラーメッセージ
```

値の割り当ての式もそれ自体が値を返す．

例. 変数への値の割り当て（通常の形式）

```
> x <- 10   Enter   ←変数 x に値 10 を割り当てる
>                   ←プロンプトの表示（割り当て処理の結果の値は得られない）
```

この形式では，変数への値の割当処理が行われただけであり，この式の実行結果としては値は得られない（値は表示されない）．しかし次の例のように，値の割り当ての式を括弧で括って実行すると，処理結果の値（割り当てた値）が返される．

例. 変数への値の割り当て（括弧で括る）

```
> (x <- 10)   Enter   ←変数 x に値 10 を割り当てる式を括弧で括る
[1] 10                ←値が返される
```

これを応用すると，複数の変数への同じ値の割り当ての式が記述できる．

例. 複数の変数に同じ値を割り当てる

```
> y <- x <- 123   Enter      ←変数 x，y に値 123 を割り当てる
> x   Enter      ←変数 x の値の確認
[1] 123          ←値が割り当てられている
> y   Enter      ←変数 y の値の確認
[1] 123          ←値が割り当てられている
```

2.1.1 変数の管理

値が割り当てられている変数の一覧を取得するには ls 関数を使用する．先の例の実行によって変数 a，b，c に値が割り当てられている状態で ls を実行する例を次に示す．

例. 値が割り当てられている変数の一覧

```
> ls()   Enter                ← ls 関数の評価
[1] "a" "b" "c"               ← 3 つの変数 a，b，c が使用されていることがわかる
```

このように ls 関数は，値が割り当てられている変数の名前のベクトルを返す．この場合，**文字列**の型の変数名のベクトルが得られる．

変数への値の割り当てを解除して変数を開放するには rm 関数を使用する．

例. 変数の開放（先の例の続き）

```
> rm(b)   Enter               ←変数 b の開放
> ls()   Enter                ←変数の一覧で確認
[1] "a" "c"                   ←変数 b は存在しない
> rm("c")   Enter             ←文字列の型で変数名 "c" を指定して開放
> ls()   Enter                ←変数の一覧で確認
[1] "a"                       ←変数 c も開放された
```

2.1.2 ベクトル

複数の要素を束ねてベクトルを作成するには c 関数[7] を用いる．

書き方：　c(値 1, 値 2, … 値 n)

値 1, 値 2, … 値 n は同じ型の値とする．

例. ベクトルの作成

```
> a <- c(10,20,30,40,50,60,70,80,90)   Enter   ← 9 個の数値を要素として持つベクトル
> a   Enter                                     ←値の確認
[1] 10 20 30 40 50 60 70 80 90                  ←値の表示
```

2.1.2.1 ベクトルの要素へのアクセス

ベクトルの要素には**添字**を付けてアクセスすることができる．添字とは，当該ベクトルの「n 番目の要素」を意味する自然数 n のことを意味する．

例. 添字による要素へのアクセス（先の例の続き）

```
> a[1]   Enter      ←ベクトル a の先頭要素（1 番目の要素）の参照
[1] 10              ←要素の値
> a[9]   Enter      ←ベクトル a の最終要素（9 番目の要素）の参照
[1] 90              ←要素の値
> a[10]   Enter     ←要素が存在しない位置の添字（10 番目）を指定する試み
[1] NA              ←値が存在しないことを意味する NA
```

[7] この関数の名前は 'concatenation'（連結）に由来する．

この例では要素の存在しない位置の参照結果として NA が得られている．これは**欠損値**（missing value）と呼ばれるものであり，値として扱ってはならない．

R 以外のプログラミング言語では，データ構造の要素を指定する添字は 0 ではじまることが一般的であるが，R 言語では添字は1 から始まる自然数であることに注意すること．

添字の範囲を [n_1:n_2] と記述して，ベクトルの要素の指定した範囲（n_1 番目から n_2 番目）を別のベクトルとして取り出す[8] ことができる．

例. 範囲指定によるベクトルの部分の取り出し（先の例の続き）

```
> a[2:5]  Enter        ←ベクトル a の 2 番目から 5 番目までの要素の取り出し
[1] 20 30 40 50        ←抽出結果
```

ベクトルの特定の要素に直接的に値を割り当て，既存のベクトルの内容を変更することができる．

例. ベクトルの要素の変更（先の例の続き）

```
> a[3] <- 33  Enter                ←ベクトル a の 3 番目の要素に値を割り当てる
> a  Enter                         ←内容確認
[1] 10 20 33 40 50 60 70 80 90     ←ベクトル a の 3 番目の要素が書き換えられている
```

同様の方法で，ベクトルの特定の範囲の要素を一括して変更することもできる．

例. ベクトルの特定の範囲の要素の変更（先の例の続き）

```
> a[7:9] <- c(77,88,99)  Enter     ←ベクトル a の 7〜9 番目の要素に値を割り当てる
> a  Enter                         ←内容確認
[1] 10 20 33 40 50 60 77 88 99     ←ベクトル a の 7〜9 番目の要素が書き換えられている
```

2.1.2.2　ベクトルの演算

ベクトル同士の算術演算（加減乗除）は，対応する要素毎に個々に（並列的に）実行される．

例. ベクトル同士の算術演算（先の例の続き）

```
> a <- c(10,20,30,40,50,60,70,80,90)  Enter  ←ベクトルの作成
> a + c(1,2,3,4,5,6,7,8,9)  Enter          ←ベクトル同士の加算
[1] 11 22 33 44 55 66 77 88 99             ←演算結果
> a - c(1,2,3,4,5,6,7,8,9)  Enter          ←ベクトル同士の減算
[1] 9 18 27 36 45 54 63 72 81              ←演算結果
> a * c(1,2,3,4,5,6,7,8,9)  Enter          ←ベクトル同士の乗算
[1] 10 40 90 160 250 360 490 640 810       ←演算結果
> a / c(5,4,3,2,1,6,7,8,9)  Enter          ←ベクトル同士の除算
[1] 2 5 10 20 50 10 10 10 10               ←演算結果
```

算術演算などの基本的な演算は p.5 の表 1「算術演算」に示したものが使用できる．また，p.6 の表 4 「R で使用できる数学関数」に示した数学関数もベクトルに対して使用できる．

例. ベクトルに対する数学関数の評価

```
> v <- c(1,2,3,4,5)  Enter     ←ベクトルの作成
> sqrt(v)  Enter               ←上記ベクトルの平方根を求める
[1] 1.000000 1.414214 1.732051 2.000000 2.236068      ←演算結果
```

長さの異なるベクトル同士の演算も可能である．

[8] Python 言語にも同様の記述方法があるが，Python の場合は「n_1 番目から $n_2 - 1$ 番目」の要素を取り出す．Python と R を混同しないように注意すること．

例. 長さの異なるベクトル同士の加算

```
> c(10,20,30,40,50,60,70,80,90) + c(1,2,3)   Enter      ←加算
[1] 11 22 33 41 52 63 71 82 93                          ←演算結果
```

この例からわかるように，短い方のベクトルを反復して長い方のベクトルに長さを合わせて加算（図 3）している．

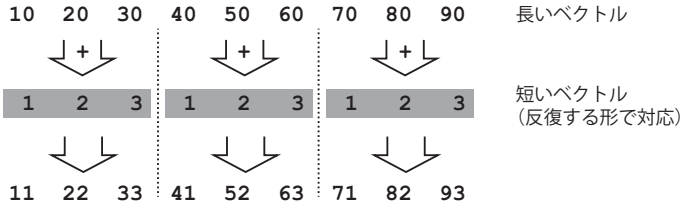


図 3: 長さの異なるベクトル同士の加算の例

課題. 長さの異なるベクトル同士の演算を，加算以外の演算でも確認せよ．

長い方のベクトルの長さが短い方のベクトルの長さの倍数になっていない場合も演算は可能である．ただし，その場合は警告メッセージが出力される．

例. 長さの異なるベクトル同士の加算（その 2）

```
> c(10,20,30,40,50,60,70,80,90) + c(1,2,3,4)   Enter      ←加算
[1] 11 22 33 44 51 62 73 84 91                            ←演算結果
 警告メッセージ:
 c(10, 20, 30, 40, 50, 60, 70, 80, 90) + c(1, 2, 3, 4) で:
   長いオブジェクトの長さが短いオブジェクトの長さの倍数になっていません
```

当然のことであるが，長さ 1 のベクトルとの演算も可能である．

例. 長さの異なるベクトル同士の加算（その 3）

```
> c(1,2,3,4) * 2   Enter      ←ベクトルの全要素を 2 倍する
[1] 2 4 6 8                   ←演算結果
```

これは，ベクトルの全要素に対する共通のスカラー演算を行う場合に応用できる．

2.1.2.3 特殊な値

数値演算の結果として特殊な値が得られることがある．例えば除算 1/0 や 0/0 は数値としての結果が得られない．

例. ゼロによる除算

```
> 1/0   Enter      ← 1 ÷ 0 を試みると…
[1] Inf            ←無限大を意味する Inf が得られる
> 0/0   Enter      ← 0 ÷ 0 を試みると…
[1] NaN            ←非数値を意味する NaN が得られる
```

1/0 の演算結果は R においては**無限大**であると見なされ，無限大を意味する Inf という値が得られる．また，0/0 の演算の結果は R においては**非数値**であるとみなされ，非数値を意味する NaN が得られる．これらの特殊な値は数値データとして扱うべきではなく，実装するアプリケーションプログラムにおいて適切に処理する必要がある．得られた値が Inf かどうかを調べるには is.infinite を，NaN かどうかを調べるには is nan を使用する．

例. 無限大かどうかの判定

```
> is.infinite( 3 )   Enter      ← 3 は無限大か？
[1] FALSE                       ←偽
> is.infinite( 1/0 )   Enter    ← 1/0 は無限大か？
[1] TRUE                        ←真
```

例. 非数値かどうかの判定

```
> is.nan( 4 )  Enter      ← 4 は非数値か？
[1] FALSE                 ←偽
> is.nan( 0/0 )  Enter    ← 0/0 は非数値か？
[1] TRUE                  ←真
```

is.infinite，is.nan の判定で得られる TRUE や FALSE は真偽を表す**論理型**[9] の値であり，これを条件判定[10] に使用することができる．

2.1.2.4　ベクトルの連結

c 関数の引数に複数のベクトルを与えることで，それらベクトルを連結することができる．

```
> v1 <- c(1,2,3)  Enter     ←ベクトル v1 の作成
> v2 <- c(4,5,6)  Enter     ←ベクトル v2 の作成
> c(v1,v2)  Enter           ← v1，v2 の連結
[1] 1 2 3 4 5 6             ←連結結果
```

関数 append によってもベクトルの連結や挿入ができる．

書き方 (1)：　append(ベクトル 1, ベクトル 2)

書き方 (2)：　append(ベクトル 1, ベクトル 2, after=挿入位置)

「ベクトル 1」に「ベクトル 2」を追加（挿入）したものを返す．書き方 (1) の場合は「ベクトル 1」の後ろに「ベクトル 2」を連結したものを返す．引数「after=」を与えた場合は「ベクトル 1」内の挿入位置を指定することができる．

例. ベクトルに他のベクトルを連結する

```
> v <- c(10,20,50)  Enter          ←ベクトル v を作成
> append(v,c(60,70))  Enter        ←ベクトル v の末尾に別のベクトルを追加
[1] 10 20 50 60 70                 ←追加結果
```

例. ベクトルに他のベクトルを挿入する（先の例の続き）

```
> append(v,c(30,40),after=2)  Enter  ←ベクトル v の要素位置 2 の後ろに別のベクトルを挿入
[1] 10 20 30 40 50                   ←挿入結果
```

append 関数は元のベクトルを変更しない．

2.1.2.5　ベクトルの長さ

ベクトルの長さ（要素の個数）は length 関数で取得できる．

例. ベクトルの長さの取得

```
> v <- c(1,2,3,4,5)  Enter     ←ベクトル v の作成
> length(v)  Enter             ←ベクトル v の長さの取得
[1] 5                          ←長さは 5
```

2.1.2.6　数列の作成

コロン「:」の記述で数列を作成することができる．

書き方：　$n_1 : n_2$

この式を評価することで開始の値 n_1，終了の値 n_2，増分 1 の数列が作成される．n_1，n_2 に浮動小数点数を与えても良い．

---

[9] 詳しくは「2.1.4 論理型」（p.17）で解説する．

[10] 詳しくは「2.2.4 条件分岐 (1)：if」（p.41）で解説する．

例. コロンで数列を作成する (1)

```
> 1:10   Enter                    ←数列の作成
 [1] 1 2 3 4 5 6 7 8 9 10        ←得られた数列
```

例. コロンで数列を作成する (2)

```
> 2.1:5.1   Enter                 ← 2.1 以上 5.1 以下の数列の作成
 [1] 2.1 3.1 4.1 5.1             ←得られた数列
> 2.1:7   Enter                   ← 2.1 以上 7 以下の数列の作成
 [1] 2.1 3.1 4.1 5.1 6.1         ←得られた数列
```

より自由な形で数列を作成するには seq 関数を使用する.

書き方 (1)： seq(開始値, 終了値, length=要素数)

「開始値」から「終了値」までの値を等分して「要素数」の個数の数列を作成する.

例. 数列の作成 (1)

```
> seq( 2, 5, length=5 )   Enter   ← 2〜5 を当分して 5 個の値を得る
 [1] 2.00 2.75 3.50 4.25 5.00     ←作成された数列
```

増分を指定して等差数列を作成するには次のように記述する.

書き方 (2)： seq(開始値, 終了値, by=増分)

「開始値」以上「終了値」以下の範囲で指定した「増分」で数列を作成する.

例. 数列の作成 (2)

```
> seq( 2, 5, by=0.7 )   Enter   ← 2 以上 5 以下の範囲で増分 0.7 の等差数列を作成する
 [1] 2.0 2.7 3.4 4.1 4.8         ←作成された数列
```

2.1.2.7 ターミナル環境におけるベクトルの表示に関すること

これまで見てきたように，ターミナル環境で R を使用する場合，ベクトルを出力するとその先頭に**インデックス**（要素の位置）が表示される.

例. ターミナル環境でのベクトルの出力

```
> c(1,2,3)   Enter   ←端末環境でベクトルを出力すると…
 [1] 1 2 3          ←先頭に [1] が表示される.
```

この例における [1] は，出力対象のベクトルの左端の要素のインデックスが「1」であることを意味する．この機能は，更に要素数の多いベクトルを表示する際の視認性の向上に役立つ.

例. 長いベクトルの表示

```
> 1:80   Enter   ←長いベクトル
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
[51] 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
[76] 76 77 78 79 80
```

このように，長いベクトルが折り返し表示された際，各行の先頭（左端）の要素のインデックスがわかる.

2.1.2.8 同じ値の要素を並べたベクトルの作成

関数 rep を使用すると，同じ値の要素を並べたベクトルを作成することができる.

書き方： rep(値, 個数)

指定した「個数」の「値」を並べたベクトルを返す.

例. 同じ値の要素を並べたベクトル

```
> v <- rep( 0.5, 10 )  Enter      ← 0.5 を 10 個数並べたベクトルを作成
> v  Enter                        ←内容確認
[1] 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5      ←得られたベクトルの内容
```

2.1.2.9 ベクトルの要素に名前を与える

ベクトルの要素には名前を与えることができる．これには関数 names を用いる．

書き方： names(ベクトル) <- c(名前 1, 名前 2, …, 名前 n)

要素数 n の「ベクトル」の各要素に「名前 1」,「名前 2」,…,「名前 n」の名前を与える．

例. ベクトルの要素に名前を与える

```
> v <- c(1,2,3,4,5)  Enter                          ←ベクトル v を作成
> v  Enter                                          ←内容確認
[1] 1 2 3 4 5                                       ←まだ名前は無い
> names(v) <- c("d1","d2","d3","d4","d5")  Enter    ←要素への名前の設定
> v  Enter                                          ←内容確認
d1 d2 d3 d4 d5                                      ←各要素に名前が
 1  2  3  4  5                                        付けられている
```

要素に名前があると，それを用いて要素にアクセスすることができる．

例. 名前を指定してベクトルの要素にアクセスする（先の例の続き）

```
> v["d3"]  Enter     ←”d3” の名前を持つベクトル v の要素を参照
d3                   ←名前 ”d3” の
 3                     要素の値
> v[4]  Enter        ←もちろん自然数の添字で
d4                     要素を参照することも
 4                     可能である
```

names 関数はベクトルの要素の名前を取得する場合にも使用することができる．

例. ベクトルの要素の名前を取得する（先の例の続き）

```
> names(v)  Enter     ←ベクトル v の要素の名前を取得する
[1] "d1" "d2" "d3" "d4" "d5"        ←得られた名前のベクトル
```

2.1.2.10 要素の含有検査

ベクトルに指定した要素が存在するかどうかを検査するには「%in%」を使用する．

書き方： 要素 %in% ベクトル

「ベクトル」に「要素」が含まれるかどうかを調べる．

例. 要素の含有検査

```
> v <- c(1,2,3,4,5)  Enter     ←ベクトル v の作成
> 3 %in% v  Enter              ← v が 3 を要素として持つかどうかを検査
[1] TRUE                            ←結果：「要素として含む」
> 7 %in% v  Enter              ← v が 7 を要素として持つかどうかを検査
[1] FALSE                           ←結果：「要素として含まない」
```

指定した要素がベクトルに含まれる場合は TRUE（真）を，含まれない場合は FALSE（偽）を返す．TRUE，FALSE は真偽を表す**論理型**[11] の値である．

[11] 詳しくは「2.1.4 論理型」（p.17）で解説する．

2.1.2.11 要素の位置の調査

指定した要素がベクトルの中のどの位置にあるかを調べるには which を使用する.

書き方： which(ベクトル==要素)

「ベクトル」中の「要素」の位置（インデックス）を返す.（演算子「==」は「等しい」ことを検査するものである）

例. ベクトル中の要素の位置を調べる

```
> v <- c(1,2,3,4,5,3,6)  Enter      ←ベクトル v の作成
> which( v==3 )  Enter      ← v の中の要素 3 が存在する位置を調べる
[1] 3 6                    ←結果
```

ベクトル v の中に要素 3 が 3 番目と 6 番目に存在していることがわかる.

2.1.2.12 要素の合計

関数 sum を使用すると数値ベクトルの要素の合計を求めることができる.

例. 要素の合計

```
> v <- c(1,2,3,4)  Enter      ←ベクトル v の
> sum(v)  Enter               ←合計を求める
[1] 10                        ←結果
```

2.1.2.13 全ての要素の積

関数 prod を使用すると数値ベクトルの全ての要素の積を求めることができる.

例. 全ての要素の積（先の例の続き）

```
> prod( v )  Enter      ←ベクトル v の全ての要素の積を求める
[1] 24                  ←結果
```

2.1.2.14 要素の差分

関数 diff を使用するとベクトルの隣接する要素の差（差分）を求めることができる.

例. ベクトルの要素の差分

```
> a <- c(1,2,3,4,5)  Enter      ←ベクトル a の
> diff(a)  Enter            ←隣接する要素の差分を求める
[1] 1 1 1 1                 ←結果
```

当然であるが，diff の戻り値のベクトルの要素数は，元のベクトルの要素数より 1 つ少ない.

2.1.2.15 要素の整列

関数 sort を使用するとベクトルの要素を昇順に整列することができる．この関数は元のベクトルを変更せず，整列したものを別のベクトルとして返す.

例. 要素を昇順に整列する

```
> v <- c(9,4,5,7,0,2,1,8,3,6)  Enter      ←ベクトル v を
> sort(v)  Enter                          ←昇順に整列する
[1] 0 1 2 3 4 5 6 7 8 9          ←整列結果
> v  Enter                       ←元のベクトルは
[1] 9 4 5 7 0 2 1 8 3 6          ←変化しない
```

sort に引数「decreasing=TRUE」を与えると降順に整列する.

例. 要素を降順に整列する（先の例の続き）

```
> sort(v,decreasing=TRUE)  Enter      ←降順に整列する
[1] 9 8 7 6 5 4 3 2 1 0               ←整列結果
```

整列処理後のベクトルの要素の元のベクトルにおけるインデックスを求めるには関数 order を使用する．

例. 関数 order による整列処理

```
> v <- c(5,3,1,2,4)  Enter      ←整列前のベクトル
> order(v)  Enter               ←整列処理
[1] 3 4 2 5 1      ←処理結果
```

この処理を図 4 に示す．

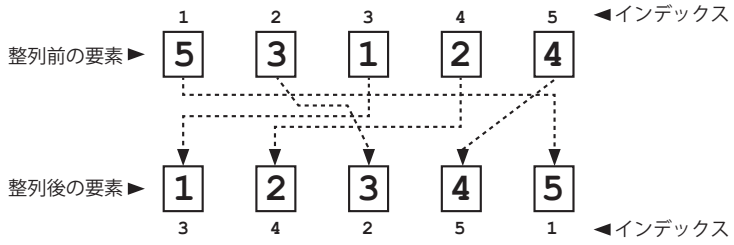


図 4: order 関数による整列

2.1.2.16　要素の順序の反転

関数 rev を使用するとベクトルの要素の順序を反転することができる．この関数は元のベクトルを変更せず，反転したものを別のベクトルとして返す．

例. 要素の順序を反転する

```
> v <- c(1,2,3,4)  Enter      ←ベクトル v の要素の順序を
> rev( v )  Enter             ←反転する
[1] 4 3 2 1          ←反転結果
> v                   ←元のベクトルは
[1] 1 2 3 4          ←変化しない
```

2.1.2.17　要素を一意に取り出す方法

関数 unique を使用するとベクトルの要素の重複を排除して一意の要素を持つベクトルを返す．この関数は元のベクトルを変更しない．

例. 要素一意に取り出す

```
> v <- c(1,5,2,4,3,0,4,2,5,1)  Enter      ←重複する要素を持つベクトル
> unique(v)  Enter              ←重複を排除する
[1] 1 5 2 4 3 0                      ←結果
```

ベクトルの要素に重複するものがあるかどうかを調べるには関数 duplicated を使用する．

例. 要素の重複の検査（先の例の続き）

```
> duplicated(v)  Enter      ←要素の重複を検査
[1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE      ←結果
> duplicated( c(1,2,3,4,5) )  Enter      ←要素の重複しないベクトルの場合
[1] FALSE FALSE FALSE FALSE FALSE      ←結果
```

この例のように，対象のベクトルの要素を先頭から順に検査し，新出の要素の位置には FALSE を，既に検出した要素の位置には TRUE を持つベクトルを返す．

2.1.3 複素数

R は**複素数**（complex）を扱うことができる．複素数は

実部 + **虚部** i

と記述する．

例. 複素数の値の記述

```
> 3 + 4i   Enter      ←複素数の値 3 + 4i
[1] 3+4i
> 4i   Enter          ←実部の記述を省くことも可能
[1] 0+4i              ←実部は 0 となる
> 3 + 0i   Enter      ←虚部を 0 としても…
[1] 3+0i              ←結果は複素数の形となる
```

複素数は

complex(re=実部, im=虚部)

と記述することもできる．「実部」「虚部」には数値のベクトルを与えることもでき，その場合は複素数のベクトルが得られる．

例. 複素数の作成（その 1）

```
> complex( re=3, im=4 )   Enter      ←複素数の値 3 + 4i
[1] 3+4i
```

例. 複素数の作成（その 2）

```
> complex( re=c(1,2,3), im=c(4,5,6) )   Enter      ←実部，虚部にベクトルを与える
[1] 1+4i 2+5i 3+6i        ←複素数のベクトル
```

複素数から実部を取り出すには Re 関数を，虚部を取り出すには Im 関数を使用する．

例. 実部，虚部の取り出し（その 1）

```
> c1 <- complex( re=3, im=4 )   Enter      ←複素数の値 3 + 4i
> Re(c1)   Enter          ←実部を求める
[1] 3                     ←実部
> Im(c1)   Enter          ←虚部を求める
[1] 4                     ←虚部
```

例. 実部，虚部の取り出し（その 2）

```
> c2 <- complex( re=c(1,2,3), im=c(4,5,6) )   Enter      ←実部，虚部にベクトルを与える
> Re(c2)   Enter          ←実部を求める
[1] 1 2 3                 ←実部（ベクトル）
> Im(c2)   Enter          ←虚部を求める
[1] 4 5 6                 ←虚部（ベクトル）
```

2.1.4 論理型

真偽を表現するための**論理型**（logical）[12] のデータがある．この型は TRUE と FALSE の 2 種類の値を持つ．R には 2 つの値が同じであるかどうかを判定する比較演算子 == があり，これによる値の比較の例を次に示す．

例. 2 つの値の比較

```
> 3 == 3   Enter      ← 3 と 3 は等しい？
[1] TRUE               ←「真」である
> 3 == 4   Enter      ← 3 と 4 は等しい？
[1] FALSE              ←「偽」である
```

上に示した値の判定の式の結果をデータとして変数に割り当てることもできる．

例. 判定結果の値（論理型）を変数に割り当てる

```
> L <- 3==3   Enter    ← 「3 と 3 は等しい？」の判定結果を変数 L に割り当てる
> L                    ←内容確認
[1] TRUE               ←論理型の TRUE を保持している
```

論理型のデータもベクトルとして扱われる．

例. ベクトル同士の比較

```
> c(1,2,3) == c(1,20,3)   Enter    ← 2 つのベクトルの比較
[1] TRUE FALSE TRUE       ←比較結果がベクトルとして得られている
```

この例からもわかるように，ベクトル同士を比較すると各要素の比較結果が論理型の値のベクトルとして得られる．

2.1.4.1 比較演算子

値の比較のための各種の演算子を表 5 に示す．

表 5: x と y を比較する演算（一部）

式	解　説	式	解　説
`a == b`	a と b は等しい	`a != b`	a と b は等しくない
`a >= b`	a は b 以上	`a <= b`	a は b 以下
`a > b`	a は b より大きい	`a < b`	a は b より小さい

複数の要素を持つベクトル同士を比較すると，各要素の比較結果を論理型のベクトルとして返す．

2.1.4.2 論理演算子

論理型のベクトル同士の対応する要素の論理積を求めるには論理演算子「&」を，論理和を求めるには「|」を使用する．

例. ベクトル同士の論理演算

```
> c(TRUE,FALSE,TRUE) & c(TRUE,FALSE,FALSE)   Enter    ←要素同士の論理積
[1] TRUE FALSE FALSE                          ←結果
> c(TRUE,FALSE,TRUE) | c(TRUE,FALSE,FALSE)   Enter    ←要素同士の論理和
[1] TRUE FALSE TRUE                           ←結果
```

TRUE，FALSE 以外の数値も論理型の値として扱うことができ，その場合 0 が FALSE，0 以外が TRUE として扱われる．

[12] R 以外の言語における**真理値**あるいは**真偽値**などと呼ばれる型に対応する．

例. ベクトル同士の論理演算（その２）

```
> c( 1, 0, 3.14 ) & c( 2.5, 0, 0 )  Enter     ←要素同士の論理積
[1] TRUE FALSE FALSE          ←結果
> c( 1, 0, 3.14 ) | c( 2.5, 0, 0 )  Enter     ←要素同士の論理和
[1] TRUE FALSE TRUE           ←結果
```

このことを応用するとビットの並びの**マスク処理**が実現できる.

例. ビットの並びのマスク処理

```
> dat <- c(1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0)  Enter     ←このデータを
> msk <- c(1,0,1,1,0,1,0,0,1,1,0,1,0,0,1,1)  Enter     ←このデータで
> res <- dat & msk  Enter                              ←マスク処理する
> as.integer(res)  Enter                       ←整数型に変換する
[1] 1 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0          ←結果
```

単一の論理型の値（長さ 1 の論理型ベクトル）同士の論理積と論理和はそれぞれ「&&」,「||」で求めることができる.

例. 単一の論理型の値の論理積

```
> TRUE && TRUE  Enter
[1] TRUE
> TRUE && FALSE  Enter
[1] FALSE
> FALSE && FALSE  Enter
[1] FALSE
```

ただし,「&&」,「||」は複数の要素を持つベクトルに対しては使用しない方が良い.

例. 良くない例

```
> c(TRUE,TRUE) && c(TRUE,FALSE)  Enter
[1] TRUE
警告メッセージ:
1:  c(TRUE, TRUE) && c(TRUE, FALSE) で:
'length(x) = 2 > 1' in coercion to 'logical(1)'
2:  c(TRUE, TRUE) && c(TRUE, FALSE) で:
'length(x) = 2 > 1' in coercion to 'logical(1)'
```

論理演算の結果として TRUE が得られているが，これは解釈しにくい．データの要素数が適切でない旨の警告メッセージが出力されている.

2.1.4.3　変数 T，F について

変数 T，F には予め TRUE，FALSE が割り当てられている.

例. 変数 T，F

```
> T  Enter      ← T には
[1] TRUE        ← TRUE が
> F  Enter      ← F には
[1] FALSE        ← FALSE が割り当てられている
```

これら変数 T，F を論理型の値 TRUE や FALSE の代わりに用いるとターミナル環境における操作が簡略化できる．ただし T，F は変数なので，別の値を割り当てることができるので十分に注意すること.

例. 危険な例

```
> T <- FALSE; F <- TRUE  Enter      ← T，F に逆の値を割り当てることも可能
> T   Enter          ← T には
[1] FALSE            ← FALSE が
> F   Enter          ← F には
[1] TRUE             ← TRUE が割り当てられている
```

論理型の値を表現するには変数 T，F ではなく TRUE，FALSE を用いて記述する方が安全である．

2.1.4.4 論理型ベクトルを用いた要素の抽出

ベクトルの要素を指定する添字 '[ ]' の中に論理型ベクトルを与えると，TRUE に対応する位置の要素を抽出することができる．

例. 論理型ベクトルを用いた要素の抽出

```
> d <- c(1,2,3,4)   Enter          ←サンプルデータ
> d[ c(FALSE,TRUE,FALSE,TRUE) ]   Enter      ←取り出したい要素を論理型ベクトルで指定
[1] 2 4                    ←結果
```

このことを応用すると，抽出条件を記述した式を与えて要素を抽出することが可能となり，柔軟な抽出処理が実現できる．

例. 1:20 から 3 の倍数を取り出す

```
> d <- 1:20   Enter          ←サンプルデータ
> d[ d%%3==0 ]   Enter       ←取り出したい要素を式で指定
[1] 3 6 9 12 15 18                  ←結果
```

2.1.4.5 数値などを論理型に変換する方法

as.logical を使用すると，数値などを論理型に変換することができる．この場合，0 を FALSE に，0 以外を TRUE に変換する．

例. 数値を論理型に変換する

```
> n <- c(1,0,1,0,2)   Enter          ←数値ベクトルを
> as.logical(n)   Enter              ←論理型ベクトルに変換する
[1] TRUE FALSE TRUE FALSE TRUE       ←変換結果
```

2.1.5 raw 型

コンピュータの記憶媒体で取り扱う最小単位である**バイト値**を取り扱うデータ型として raw 型がある．バイト値は符号なしの形で 0〜255 の範囲の整数値として表現される．

例. 整数値を raw 型に変換する

```
> a <- c(0,9,10,15,16,239,240,255)  Enter      ←整数値のベクトルを作成
> r <- as.raw(a)  Enter                        ←上記のベクトルを raw 型に変換
> r  Enter      内容確認
[1] 00 09 0a 0f 10 ef f0 ff            ← raw 型の値
```

この例は，数値のベクトル a を as.raw で raw 型に変換したものである．raw 型ベクトルをターミナル環境で出力すると 16 進数の形式で表示される．

raw 型の値を整数型に変換（逆変換）するには as.integer を使用する．

例. raw 型を整数値に変換する（先の例の続き）

```
> as.integer(r)  Enter      ←先に作成した raw 型のベクトル r を整数に変換する
[1]   0   9  10  15  16 239 240 255              ←整数のベクトルに戻った
```

バイト値の範囲を超える値を as.raw で変換しようとすると警告メッセージが表示され，変換結果は 0 となる．

例. バイト値の範囲を超える値の変換 (1)

```
> as.raw(-1)  Enter      ←負の値を raw 型に変換する試み
[1] 00                   ← 0 となる
  警告メッセージ:
  raw コネクションで、範囲外の値は 0 として扱いました
```

例. バイト値の範囲を超える値の変換 (2)

```
> as.raw(256)  Enter      ←バイト値の最大値より大きい値を raw 型に変換する試み
[1] 00                   ← 0 となる
  警告メッセージ:
  raw コネクションで、範囲外の値は 0 として扱いました
```

2.1.6　文字列

R では**文字列**（character）をデータとして扱うことができる．文字列は**ダブルクォート**「"」もしくは**シングルクォート**「’」で括って表現する．どちらのクォートでも文字列を表現できるが，R では標準的にダブルクォートを使用する．

例. 文字列

```
> "プログラミング言語 R"  Enter      ←文字列
[1] "プログラミング言語 R"          ←文字列 1 つを要素として持つベクトル
> 'abcde'  Enter                  ←シングルクォートも使用可能
[1] "abcde"                       ←ただし，評価結果はダブルクォートで括られている
```

文字列もベクトルとして扱われる．

例. 要素数 1 の文字列ベクトル

```
> s1 <- "文字列"  Enter      ←要素 1 個の文字列ベクトル s1
> length(s1)  Enter          ← s1 の要素数の確認
[1] 1                        ←要素数 1
```

例. 要素数 3 の文字列ベクトル

```
> s2 <- c("文字列 1","文字列 2","文字列 3")  Enter      ←要素 3 個の文字列ベクトル s2
> s2  Enter                                          ←内容確認
[1] "文字列 1" "文字列 2" "文字列 3"          ←文字列を要素とするベクトルになっている
> length(s2)  Enter       ← s2 の要素数の確認
[1] 3                     ←要素数 3
```

2.1.6.1　文字列の連結

文字列を連結するには paste 関数を使用する．

書き方 (1)：　paste(s1, s2, ⋯ sn)

文字列ベクトル s1, s2, ⋯ sn を連結する．各ベクトルが複数の要素を持つ場合は，対応する要素を連結したものを要素とする文字列ベクトルを返す．

例. 文字列の連結

```
> paste("ab","cd","ef")  Enter      ← 3 つの文字列の連結
[1] "ab cd ef"                      ←連結結果
```

これは 3 つの文字列 ”ab”,”cd”,”ef” を連結する例である．この例からもわかるように，連結対象が空白文字（スペース）で区切られた形の連結結果となっている．空白文字以外の文字列を区切りとして挿入するには paste 関数の引数に **sep=”区切り文字列”** を与える．

例. 区切り文字を指定して文字列を連結

```
> paste("ab","cd","ef", sep="/")  Enter      ←”/”を区切りとして連結
[1] "ab/cd/ef"                               ←連結結果
> paste("ab","cd","ef", sep="")  Enter       ←空文字列 ”” を区切りとして連結
[1] "abcdef"                                 ←連結結果
```

次に，複数要素を持つ文字列ベクトルの連結の例を示す．

例. 複数要素を持つ文字列ベクトルの連結

```
> v1 <- c("ab","cd","ef")  Enter      ← 3 つの要素の文字列ベクトル v1
> v2 <- c("12","34","56")  Enter      ← 3 つの要素の文字列ベクトル v2
> paste(v1,v2)  Enter                 ← v1, v2 の連結
[1] "ab 12" "cd 34" "ef 56"           ←連結結果
> paste(v1,v2, sep="/")  Enter        ←”/”を区切りとして連結
[1] "ab/12" "cd/34" "ef/56"           ←連結結果
```

複数要素を持つ文字列ベクトルの各要素を連結することもできる．

書き方 (2)：　paste(v,collapse=”区切り文字列”)

文字列ベクトル v の各要素を連結したものを返す．このとき「区切り文字列」を各要素の繋ぎ目に挿入する．

例. 文字列ベクトルの各要素の連結

```
> v <- c("ab","cd","ef")  Enter    ← 3 つの要素の文字列ベクトル v
> paste(v,collapse="/")  Enter    ←”/”を挿入して各要素を連結
[1] "ab/cd/ef"                    ←連結結果
> paste(v,collapse="")  Enter     ←空文字列 ”” を挿入して各要素を連結
[1] "abcdef"                      ←連結結果
```

2.1.6.2　文字列の部分の取り出し

文字列の指定した部分を取り出すには substring 関数を使用する．

書き方：　substring(s, n_1, n_2)

文字列 s の開始位置 n_1 から終了位置 n_2 までの部分を取り出す．

例. 部分文字列の取り出し

```
> substring("あいうえおかきくけこ",3,6)  Enter    ←指定した部分（3〜6 番目）の取り出し
[1] "うえおか"                                    ←得られた部分
```

複数要素を持つ文字列ベクトルに対して substring 関数を使用する例を次に示す．

例. 複数要素を持つ文字列ベクトルに対する処理

```
> s <- c("あいうえお","かきくけこ","さしす")  Enter    ← 3 要素の文字列ベクトル s
> substring(s,3,5)  Enter                  ←各要素の 3〜5 番目の取り出し
[1] "うえお" "くけこ" "す"                  ←処理結果（文字列ベクトル）
```

このように，文字列ベクトルの各要素の部分文字列が取り出され，それらをベクトルとして返していることがわかる．また，この例からわかるように，文字列の最終位置以降を部分指定の終了位置として指定すると，実際の文字列の終了位置までの部分が取り出される．

2.1.6.3　文字列の長さ（文字数）

文字列の長さ（文字数）を調べるには nchar 関数を使用する．

例. 文字列を構成する文字数を調べる

```
> nchar("プログラミング")  Enter    ←文字列の文字数を調べる
[1] 7                              ←文字数 7
> s <- c("a","bc","def","ghij")  Enter    ←文字列ベクトル s を作る
> nchar(s)  Enter                  ←文字列ベクトルの要素の文字数を調べる
[1] 1 2 3 4                        ←文字数のベクトル
```

2.1.6.4　文字列の分解

文字列に含まれる区切り文字列（正規表現によるパターン）を境にその文字列を分解するには strsplit 関数を使用する．

書き方：　strsplit(文字列, パターン)

「パターン」を区切りにして「文字列」を分解する．分解結果は**リスト**[13] の形で得られる．

[13] 詳しくは「2.1.9 リスト」（p.32）で解説する．

例. 文字列の分解 (1)

```
> s <- "a/bc/def/ghij"  Enter      ←対象の文字列 s
> r <- strsplit(s,"/")  Enter      ←区切り文字 "/" を境に s を分解する
> r  Enter                        ←分解結果の確認
[[1]]                             ←得られたリストの第 1 要素
[1] "a" "bc" "def" "ghij"         ←分解結果（リストの第 1 要素のベクトル）
```

この例では，文字列 s を区切り文字 "/" で分解した結果を r に得ている．この r の第 1 要素から分解結果の文字列ベクトルを得るには次のようにする．

例. 分解結果の文字列リストの取得（先の例の続き）

```
> unlist(r)  Enter      ←リスト r をベクトルに変換する
[1] "a" "bc" "def" "ghij"      ←文字列 s の分解結果の文字列ベクトル
```

この例に示した unlist 関数は，リストをベクトルに変換するものである．同様の方法で，複数の要素を持つ文字列ベクトルを分解する例を次に示す．

例. 文字列の分解 (2)

```
> s <- c("a/bc/def/ghij","klmn/opq/rs/t")  Enter      ←対象の文字列ベクトル s
> r <- strsplit(s,"/")  Enter      ←区切り文字 "/" を境に s を分解する
> r  Enter      ←分解結果の確認
[[1]]                             ←得られたリストの第 1 要素
[1] "a" "bc" "def" "ghij"         ←分解結果 1（リストの第 1 要素のベクトル）
[[2]]                             ←得られたリストの第 2 要素
[1] "klmn" "opq" "rs" "t"         ←分解結果 2（リストの第 2 要素のベクトル）
> unlist(r)  Enter                ←リスト r をベクトルに変換する
[1] "a" "bc" "def" "ghij" "klmn" "opq" "rs" "t"      ←文字列 s の分解結果の文字列ベクトル
```

strsplit 関数のパターン（第 2 引数）に空文字列 "" を与えると，文字列を 1 文字ずつに（構成要素に）分解することができる．

例. 文字列の分解 (3)

```
> unlist( strsplit("abcde","") )  Enter      ←文字列 "abcde" をバラバラにする
[1] "a" "b" "c" "d" "e"                      ←構成文字のベクトルが得られる
```

2.1.6.5 文字列から raw 型ベクトルへの変換

関数 charToRaw を使用すると，文字列を raw 型に変換することができる．

例. 文字列を raw 型に変換する

```
> r1 <- charToRaw("abcABC123$%&")  Enter          ←アスキー文字列を raw 型に変換
> r1  Enter                                       ←確認
[1] 61 62 63 41 42 43 31 32 33 24 25 26           ←変換結果
> r2 <- charToRaw("日本語にほんご")  Enter          ←全角文字列を raw 型に変換
> r2  Enter                                       ←確認
[1] e6 97 a5 e6 9c ac e8 aa 9e e3 81 ab e3 81 bb e3 82 93 e3 81 94      ←変換結果
```

参考）

上の例は Windows 用の R（4.2.1 版）で実行したものであり，この処理系の内部の全角文字列（日本語文字列）のエンコーディングは **UTF-8** である．異なる版の R，あるいは異なるプラットフォームの R では別のエンコーディング[14] が採用されていることがあり，その場合は上の実行結果とは異なる結果となる．

[14]例えば**シフト JIS** などがある．

2.1.6.6　raw 型ベクトルから文字列への変換

raw 型ベクトルの値を**文字コード**の並びとみなして文字列に変換するには rawToChar 関数を使用する．

例. raw 型ベクトルを文字列に変換する（先の例の続き）

```
> rawToChar(r1)  Enter          ←変換処理
[1] "abcABC123$%&"              ←変換結果
> rawToChar(r2)  Enter          ←変換処理
[1] "日本語にほんご"            ←変換結果
```

2.1.6.7　文字コードと文字の対応

文字列と raw 型ベクトルの間の変換処理を応用すると，文字に対応する**文字コード**[15] を取得する，あるいは逆に文字コードに対応する文字を取得することができる．

例. 半角英大文字 ”A” の**アスキー（ASCII）コード**を 10 進数の整数値として取得する

```
> r <- as.integer(charToRaw("A"))  Enter   ←アスキーコードの取得
> r  Enter                         ←内容確認
[1] 65                             ← 10 進数でのアスキーコード
```

例. 10 進数のアスキーコードに対応する文字を取得する（先の例の続き）

```
> rawToChar(as.raw(r))  Enter   ←アスキーコードに対する文字の取得
[1] "A"                         ←結果
```

2.1.6.8　複数の行に渡る文字列の記述

文字列の途中で「¥」（もしくはバックスラッシュ「\」）で終了して改行すると，複数の行に渡る文字列を記述することができる．

例. 複数の行に渡る文字列（ターミナル環境における操作）

```
> s <- "abcd\  Enter     ←末尾にバックスラッシュを書いて改行
+ efgh"  Enter           ←続きの文字列の記述（先頭のプラス記号は自動的に表示される）
> s  Enter               ←内容確認
[1] "abcd\nefgh"         ←改行のエスケープシーケンス「\n」が含まれている
```

このように，文字列中の改行位置にエスケープシーケンス「\n」が挿入される．

2.1.6.9　文字列以外のオブジェクトを文字列に変換する方法

as.character を使用することで，文字列以外のオブジェクトを文字列に変換することができる．

例. 数値を文字列に変換する

```
> as.character(123)  Enter          ←整数を文字列に変換
[1] "123"                           ←変換結果
> as.character(3.14159265)  Enter   ←浮動小数点数を文字列に変換
[1] "3.14159265"                    ←変換結果
```

[15] ここでは，文字を表現するバイト列を整数値で表現したものを指す．厳密な定義に関しては文字の**符号化モデル**についての解説資料を参照されたい．

2.1.7 データの型に関すること

これまで様々なデータの型について解説したが，ベクトルの要素となる基本的な**値**の型は表 6 に挙げる 6 種類であり，typeof 関数，mode 関数で調べることができる．

表 6: 基本的なデータの型

型を調べる関数	論 理 型	整　数	浮動小数点数	複 素 数	文 字 列	raw 型
typeof	logical	integer	double	complex	character	raw
mode	logical	numeric	numeric	complex	character	raw

例. typeof 関数による型の検査

```
> typeof( TRUE ) Enter
[1] "logical"
> typeof( 3L ) Enter
[1] "integer"
> typeof( 3.14 ) Enter
[1] "double"
> typeof( 2+3i ) Enter
[1] "complex"
> typeof( "abc" ) Enter
[1] "character"
```

例. mode 関数による型の検査

```
> mode( TRUE ) Enter
[1] "logical"
> mode( 3L ) Enter
[1] "numeric"
> mode( 3.14 ) Enter
[1] "numeric"
> mode( 2+3i ) Enter
[1] "complex"
> mode( "abc" ) Enter
[1] "character"
```

typeof 関数と異なり，mode 関数で値の型を調べると，整数，浮動小数点数のどちらも「numeric」となる[16]．値の型を調べる関数には storage.mode もあるが，これは他の言語処理系との互換性のためのものであり，これに関する解説は本書では割愛する．

オブジェクトの型に関する属性として class **属性**もあるが，これに関しては「2.5 オブジェクト指向プログラミング」（p.62）で解説する．

2.1.7.1 データの型の判定

「is. **型名**」でデータの型が判定できる．

例. 整数かどうかの検査

```
> is.integer( 3L ) Enter
[1] TRUE          ←判定結果
> is.integer( 3 ) Enter
[1] FALSE         ←判定結果
```

例. 論理型かどうかの検査

```
> is.logical( 0L ) Enter
[1] FALSE         ←判定結果
> is.logical( FALSE ) Enter
[1] TRUE          ←判定結果
```

例. 浮動小数点数かどうかの検査

```
> is.double( 3.14 ) Enter
[1] TRUE          ←判定結果
> is.double( 3L ) Enter
[1] FALSE         ←判定結果
```

例. 文字列型かどうかの検査

```
> is.character( "abc" ) Enter
[1] TRUE          ←判定結果
> is.character( 3.14 ) Enter
[1] FALSE         ←判定結果
```

2.1.7.2 データの型の変換

多くの場合「as. **型名**」でデータの型が変換できる．（次に示す例以外も試されたい）

例. データの型の変換

```
> as.double( "3.1415926535" ) Enter
[1] 3.141593          ←変換結果
> as.character( TRUE ) Enter
[1] "TRUE"            ←変換結果
```

[16] これは **S 言語**との互換性のための機能である．

2.1.8　行列，配列

行列は「行」と「列」から構成される 2 次元的なデータ構造であり，同じデータ型の要素で構成される．R では行列を matrix という型のオブジェクトとして扱い，次のように記述して作成することができる．

書き方 (1)：　matrix(nrow=行数, ncol=列数)

この式を評価すると指定した「行数」「列数」の行列が得られる．

例. 行列の生成

```
> m <- matrix(nrow=3,ncol=4)  Enter      ← 3 行 4 列の行列を作成する
> m  Enter      ←内容確認
     [,1] [,2] [,3] [,4]        ←行列が得られた
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
```

この例では値を与えずに行列を作成したので，全ての要素が NA（**欠損値**）になっている．ターミナル環境で行列を出力すると上の例のように [行番号, 列番号] が表示される．

行列には添字 [**行番号, 列番号**] を付けることでその値にアクセスすること（値の割り当てと参照）ができる．

例. 行列の要素へのアクセス（先の例の続き）

```
> m[1,1] <- 11  Enter      ← 1 行 1 列目の要素に 11 を割り当てる
> m[1,2] <- 12  Enter      ← 1 行 2 列目の要素に 12 を割り当てる
> m  Enter      ←内容確認
     [,1] [,2] [,3] [,4]
[1,]   11   12   NA   NA        ←行列の内容が変更された
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
```

添字にはコロン「:」で範囲を記述することもできる．

例. 指定した列範囲の要素へのアクセス（先の例の続き）

```
> m[2,2:3] <- c(22,23)  Enter      ← 2 行 2〜3 列目の要素に 22,23 を割り当てる
> m  Enter      ←内容確認
     [,1] [,2] [,3] [,4]
[1,]   11   12   NA   NA
[2,]   NA   22   23   NA        ←行列の内容が変更された
[3,]   NA   NA   NA   NA
```

例. 指定した行範囲の要素へのアクセス（先の例の続き）

```
> m[2:3,4] <- c(24,34)  Enter      ← 2〜3 行 4 列目の要素に 24,34 を割り当てる
> m  Enter      ←内容確認
     [,1] [,2] [,3] [,4]
[1,]   11   12   NA   NA
[2,]   NA   22   23   24        ←行列の内容が変更された
[3,]   NA   NA   NA   34        ←行列の内容が変更された
```

【重要】

データの中の欠損値 NA は通常の数値として扱うべきではなく，実装するアプリケーションの中で適切に処理する必要がある．値が NA かどうかを調べるには is.na を使用する．

例. NA かどうかの検査（先の例の続き）

```
> is.na( m[1,4] )  Enter      ← 配列の NA の箇所を調べる
[1] TRUE                      ←真
> is.na( m[2,4] )  Enter      ← 配列の NA でない箇所を調べる
[1] FALSE                     ←偽
```

行列生成時に予め値を与えることもできる．

書き方 (2)：　matrix(初期値, nrow=行数, ncol=列数)

「初期値」には 1 つの値，あるいは数列を与えることができる．

例. 全要素が 0 の行列の生成

```
> m <- matrix( 0, nrow=2, ncol=3 )  Enter      ← 2 行 3 列の行列を作成する
> m  Enter      ←内容確認
     [,1] [,2] [,3]
[1,]    0    0    0       ←全ての要素が 0
[2,]    0    0    0
```

行列作成時の初期値に数列を与えることもできる．

例. 数列を初期値に与えて行列を生成する（その 1）

```
> m <- matrix( 1:6, nrow=2, ncol=3 )  Enter      ← 2 行 3 列の行列を作成する
> m  Enter      ←内容確認
     [,1] [,2] [,3]
[1,]    1    3    5     ←縦方向に（上から下に向かって）
[2,]    2    4    6       数列の要素が割り当てられる
```

この例からもわかるように，行方向（縦方向）の順に数列の要素が初期値として割り当てられる．これに対して，行列作成時に引数 byrow=TRUE を与えると，列方向（横方向）の順に初期値が割り当てられる．

例. 数列を初期値に与えて行列を生成する（その 2）

```
> m <- matrix( 1:6, nrow=2, ncol=3, byrow=TRUE )  Enter      ← 2 行 3 列の行列を作成する
> m  Enter      ←内容確認
     [,1] [,2] [,3]
[1,]    1    2    3     ←横方向に（左から右に向かって）
[2,]    4    5    6       数列の要素が割り当てられる
```

2.1.8.1　行，列へのアクセス

行列オブジェクトに添字を与える際，列の指定を省略すると行が，列の指定を省略すると列がベクトルの形で得られる．

例. 行，列の参照（先の例の続き）

```
> m[1,]  Enter      ←添字の列の指定を省略
[1] 1 2 3       ← 1 番目の行がベクトルとして得られる
> m[,2]  Enter      ←添字の行の指定を省略
[1] 2 5         ← 2 番目の列がベクトルとして得られる
```

同様の記述方法で行，列に値を設定することもできる．

例. 行，列への値の設定（先の例の続き）

```
> m[1,] <- c(10,20,30)  Enter      ← 1 番目の行に値を設定
> m  Enter                          ←設定結果の確認
     [,1] [,2] [,3]                 ←設定結果
[1,]   10   20   30
[2,]    4    5    6
> m[,2] <- c(200,500)  Enter        ← 2 番目の列に値を設定
> m  Enter                          ←設定結果の確認
     [,1] [,2] [,3]                 ←設定結果
[1,]   10  200   30
[2,]    4  500    6
```

【重要】

配列の要素のデータ型は全て同じものであり，既存の配列の要素に異なる型の値を設定すると，全ての要素の型が変更されることがある．例えば先の例で作成した配列 m の要素は数値であるが，その配列の要素に文字列の値を与えると全ての要素が文字列となる．

例. 数値の配列の要素に文字列を設定する（先の例の続き）

```
> m[1,1] <- "ab"  Enter     ←数値配列の要素に文字列を設定
> m  Enter                  ←設定結果の確認
     [,1] [,2] [,3]         ←設定結果
[1,] "ab" "200" "30"          全ての要素が文字列になっている
[2,] "4"  "500" "6"
```

2.1.8.2　行列を直接編集するための便利な機能

関数 edit を使用すると，GUI で行列の内容を直接的に編集することができる．

例. 行列を edit で編集する

```
> m <- matrix(c(0,0,0,0),nrow=2,ncol=2)  Enter
> m2 <- edit(m)  Enter  ←編集開始（右のような UI が現れる）
> m2  Enter   ←内容確認（編集終了後）
     col1 col2   ←編集結果
[1,]   11   12
[2,]   21   22
```

edit は編集後の行列を別のオブジェクトとして返す．従って上の例では m の内容は変更されない．

2.1.8.3　行列の演算

p.5 の表 1 に挙げた演算を行列に対して実行することができる．その場合，対象の行列間の対応する要素の演算となる．これに関していくつか例を示す．

例. サンプルデータの作成

```
> m1 <- matrix( 1:6, nrow=2, ncol=3, byrow=TRUE )  Enter   ←行列 m1 の作成
> m1  Enter   ←内容確認
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> m2 <- matrix( seq(10,60,by=10), nrow=2, ncol=3, byrow=TRUE )  Enter   ←行列 m2 の作成
> m2  Enter   ←内容確認
     [,1] [,2] [,3]
[1,]   10   20   30
[2,]   40   50   60
```

例. 行列の加算（先の例の続き）

```
> m1 + m2  Enter      ←行列の加算
     [,1] [,2] [,3]
[1,]   11   22   33
[2,]   44   55   66
```

例. 行列の乗算（先の例の続き）

```
> m1 * m2  Enter      ←行列の乗算
     [,1] [,2] [,3]
[1,]   10   40   90
[2,]  160  250  360
```

上の例における行列の乗算は，線形代数における行列の積とは異なることに注意すること．線形代数における行列の積は %*% で実行する．次に示す例は，

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

を実行するものである．

例. 1 つ目の行列の作成

```
> m1 <- matrix( 1:4, nrow=2, ncol=2, byrow=TRUE )  Enter      ← 1 つ目の行列 m1
> m1  Enter      ←内容確認
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

例. 2 つ目の行列の作成（先の例の続き）

```
> m2 <- matrix( 5:8, nrow=2, ncol=2, byrow=TRUE )  Enter      ← 2 つ目の行列 m2
> m2  Enter      ←内容確認
     [,1] [,2]
[1,]    5    6
[2,]    7    8
```

例. 線形代数における行列の積（先の例の続き）

```
> m1 %*% m2  Enter      ←上記 2 つの行列の積
     [,1] [,2]
[1,]   19   22
[2,]   43   50
```

2.1.8.4 行，列に名前を与える

行列の行と列には名前を与えることができる．具体的には，行列オブジェクト（matrix オブジェクト）作成時に引数 **dimnames=list(行の名前のベクトル, 列の名前のベクトル)** を与える．

例. 行，列に名前を与えて matrix オブジェクトを作成する

```
> m <- matrix( 1:6, nrow=2, ncol=3,  Enter
+        dimnames=list(c("d1","d2"),c("c1","c2","c3")) )  Enter  ←行列作成
> m  Enter      ←内容確認
   c1 c2 c3          ←各行，各列に名前が付いている
d1  1  3  5
d2  2  4  6
```

行列の要素には名前を指定してアクセスすることができる．

例. 行，列の名前を指定して要素にアクセスする（先の例の続き）

```
> m["d2","c2"]   Enter   ←行と列の名前を指定して要素を参照
[1] 4                    ←得られた値
> m[2,2]   Enter        ←もちろん自然数で位置を指定して
[1] 4                      要素にアクセスすることもできる
```

dimnames 関数を使用すると既存の行列に対して行の名前，列の名前を与えることができる．

例. dimnames 関数で行，列に名前を与える．

```
> m <- matrix( 1:6, nrow=2, ncol=3 )   Enter   ←行列 m を作成
> dimnames(m) <- list(c("d1","d2"),c("c1","c2","c3"))  Enter  ← m の行, 列に名前を付ける
> m   Enter      ←内容確認
   c1 c2 c3          ←各行，各列に名前が付いている
d1  1  3  5
d2  2  4  6
```

この他にも列の名前を与える関数 colnames，行の名前を与える関数 rownames 関数もある．

例. colnames 関数で列の名前を与える

```
> m <- matrix( 1:6, nrow=2, ncol=3 )   Enter   ←行列 m を作成
> colnames(m) <- c("c1","c2","c3")  Enter     ← m の列に名前を付ける
> m   Enter      ←内容確認
     c1 c2 c3          ←各列に名前が付いている
[1,]  1  3  5
[2,]  2  4  6
```

例. rownames 関数で行の名前を与える（先の例の続き）

```
> rownames(m) <- c("d1","d2")  Enter     ← m の行に名前を付ける
> m   Enter      ←内容確認
   c1 c2 c3          ←各行，各列に名前が付いている
d1  1  3  5
d2  2  4  6
```

行，列の既存の名前を colnames，rownames 関数で参照することもできる．

例. colnames，rownames 関数による名前の参照（先の例の続き）

```
> colnames(m)   Enter     ←列の名前を参照
[1] "c1" "c2" "c3"       ←列の名前が得られている
> rownames(m)   Enter     ←行の名前を参照
[1] "d1" "d2"            ←行の名前が得られている
```

2.1.8.5 配列

次元が 2 よりも大きい配列は array オブジェクトとして扱い，次のように記述して作成することができる．

書き方 (1)：　array(dim=c(要素数 1, 要素数 2,…))

書き方 (2)：　array(初期値, dim=c(要素数 1, 要素数 2,…))

引数「dim=」には配列の各次元の要素数を与える．また（当然のことであるが）この引数に与えるベクトルの長さ（要素数）が当該配列の次元となる．書き方 (1) のように初期値を省略すると，全要素が NA である配列が作成される．

ここでは，3 次元の配列の扱いを例に挙げて array オブジェクトについて解説する．

■ 配列の作成

初期値を与えて 3 次元の配列を作成する例を示す．

例. $2 \times 2 \times 2$ のサイズの 3 次元配列の作成

```
> a <- array( 1:8, dim=c(2,2,2) )  Enter      ← 3 次元配列の作成
> a  Enter      ←内容確認
, , 1                ←「以下 a[,,1] の内容」を意味する
     [,1] [,2]
[1,]    1    3
[2,]    2    4
, , 2                ←「以下 a[,,2] の内容」を意味する
     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

この例では 3 次元の配列 a を作成し，その内容を出力して確認している．ターミナル環境ではこの様に 2 次元の行列を切り出す形で出力される．また

```
a[,,1]
```

のように 2 次元の部分を切り出す形でアクセスすると，その部分は matrix（配列）オブジェクトとして扱われ，

```
a[1,,1]
```

のように 1 次元の部分を切り出す形でアクセスすると，その部分はベクトルオブジェクトとして扱われる．

配列オブジェクトは行列の場合に準じた形で扱うことができる．

2.1.9　リスト

リストは，複数の異なる型のオブジェクトを要素として保持することができる 1 次元のデータ構造である．

2.1.9.1　リストの作成

リストは次のように記述して作成することができる．

書き方：　list(要素 1, 要素 2, …, 要素 n)

これにより「要素 1」〜「要素 n」を保持するリストが作成される．また引数を省略すると，要素を持たない**空リスト**が得られる．

例. リストの作成

```
> L <- list(1,2,3)  Enter      ←数値 1, 2, 3 を要素とするリストを作成
L  Enter      ←内容確認
[[1]]          ← 1 番目の要素
[1] 1          ←その値
[[2]]          ← 2 番目の要素
[1] 2          ←その値
[[3]]          ← 3 番目の要素
[1] 3          ←その値
```

これは 3 つの数値を要素とするリスト L を作成する例である．

2.1.9.2　リストの要素へのアクセス

リストの要素にアクセスするには括弧‘[[…]]’で添字を付ける．ターミナル環境では，各要素の出力時にその添字が出力される．

例. リストの要素にアクセスする（先の例の続き）

```
> L[[2]]  Enter      ←リスト L の 2 番目の要素を参照
[1] 2              ←参照結果
> L[[3]] <- 30  Enter      ←リスト L の 3 番目の要素に値を設定
L  Enter      ←内容確認
[[1]]          ← 1 番目の要素
[1] 1          ←その値
[[2]]          ← 2 番目の要素
[1] 2          ←その値
[[3]]          ← 3 番目の要素
[1] 30         ←その値
```

リストの要素にアクセスする際に括弧‘[…]’の添字を付けると，対象の要素がリストの形で得られる．

例. 括弧‘[…]’の添字で要素を参照（先の例の続き）

```
> L[2]  Enter      ←リスト L の 2 番目の要素を添字‘[2]’で参照
[[1]]              ←対象要素がリストの形で
[1] 2                得られる
```

存在しない要素にアクセスしようとするとエラーが発生する．

例. 存在しない要素にアクセスする試み（先の例の続き）

```
> L[[5]]  Enter          ←存在しない要素（5 番目）にアクセスしようとすると…
L[[5]] でエラー:  添え字が許される範囲外です          ←エラーが発生する
```

ただし，要素が未だ存在しない位置に値を設定することはできる．（次の例）

例. 要素が存在しない位置に値を設定する（先の例の続き）

```
> L[[5]] <- 50  Enter      ←要素が存在しない位置（5 番目）に値を設定する
L  Enter      ←内容確認
[[1]]         ← 1 番目の要素
[1] 1         ←その値
      ⋮
（途中省略）
      ⋮
[[4]]         ← 4 番目の要素（値を与えていない）
NULL          ←その値
[[5]]         ← 5 番目の要素（値を与えた）
[1] 50        ←その値
```

この例では，要素数が 3 のリストに対して 5 番目の要素を与えている．この場合は，値を与えていない 4 番目の要素として NULL が自動的にリストに加えられる．NULL は欠損値 NA とは異なる特殊なオブジェクトである．NULL は「値が存在しない」ことを意味するオブジェクトであり，これをデータとして扱うべきではなく，実装するアプリケーションプログラムの中で適切に処理する必要がある．値が NULL かどうかを調べるには is.null を使用する．

例. NULL かどうかを調べる（先の例の続き）

```
> is.null( L[[4]] )  Enter    ← NULL である箇所を調べる
[1] TRUE                      ←真
> is.null( L[[5]] )  Enter    ← NULL でない箇所を調べる
[1] FALSE                     ←偽
```

2.1.9.3 リストの長さ

リストの長さ（要素の個数）を求めるには length 関数を使用する．

例. リストの長さを求める（先の例の続き）

```
> length(L)  Enter     ←リスト L の要素の個数を求める
[1] 5        ←要素は 5 個
```

2.1.9.4 リストの要素の削除

既存のリストから特定の要素を削除するには，削除対象の要素に対して NULL を設定する．

例. リストの要素の削除

```
> L <- list(1,2,3)  Enter     ←リスト L の作成
> L[[2]] <- NULL  Enter       ←リスト L の 2 番目の要素を削除
> L  Enter      ←内容確認
[[1]]
[1] 1
[[2]]
[1] 3
```

この例は 3 つの要素を持つリストの 2 番目の要素「2」を削除する処理を示したものである．

注意）リスト L の n 番目の要素として NULL を設定する場合は

```
L[n] <- list(NULL)
```

とする．

例. リストに要素として NULL を与える

```
> L <- list(1,2,3)  Enter      ←リスト L の作成
> L[2] <- list(NULL)  Enter    ←リスト L の 2 番目の要素に NULL を与える
> L  Enter      ←内容確認
[[1]]
[1] 1

[[2]]           ← 2 番目の要素が
NULL              NULL になっている

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

2.1.9.5　リストの連結，挿入

先に，関数 append を用いてベクトルの連結，挿入について解説したが，append はリストに対しても同様の処理を行う．

例. リストの連結

```
> L1 <- list(1,2,3) Enter
> L2 <- list(4,5,6) Enter
> append(L1,L2)  Enter
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

[[4]]
[1] 4

[[5]]
[1] 5

[[6]]
[1] 6
```

例. リストの挿入（左の例の続き）

```
> append(L1,L2,after=2)  Enter
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 4

[[4]]
[1] 5

[[5]]
[1] 6

[[6]]
[1] 3
```

2.1.9.6　空リスト

要素を持たない**空リスト**は `list()` である．

例. 空リストの作成

```
> L <- list()  Enter      ←空リストを作成
> L  Enter      ←内容確認
list()            ←空リスト
> length(L)  Enter      ←リスト L の要素の個数を確認
[1] 0      ← 0 個（要素なし）
```

2.1.9.7　再帰的なリスト構造

リストは異なるデータ型の要素を含むことができる．別のリストを要素として含めることもできる．

例. 様々な型のデータを含むリスト

```
> L <- list("abc",list(21,"def"),30)  Enter      ←様々な型のデータを含むリスト
> L  Enter      ←内容確認
[[1]]          ← 1 番目の要素
[1] "abc"          ←文字列
[[2]]          ← 2 番目の要素（リスト）
[[2]][[1]]          ←その値の更に 1 番目の要素
[1] 21              ←その値（数値）
[[2]][[2]]          ← 2 番目の要素内の 2 番目の要素
[1] "def"           ←その値（文字列）
[[3]]          ← 3 番目の要素
[1] 30              ←その値（数値）
```

2.1.9.8　リストの要素に名前を与える

リストの要素には名前を与えることができる．これには関数 names を用いる．

書き方：　names(リスト) <- c(名前 1, 名前 2, …, 名前 n)

要素数 n の「リスト」の各要素に「名前 1」,「名前 2」,…,「名前 n」の名前を与える．

例. リストの要素に名前を与える

```
> L <- list(1,2,3)  Enter      ←リスト L の作成
> names(L) <- c("e1","e2","e3")  Enter      ←リスト L の各要素に名前を与える
> L  Enter      ←内容確認
$e1          ← 1 番目の要素の名前
[1] 1        ←その値
$e2          ← 2 番目の要素の名前
[1] 2        ←その値
$e3          ← 3 番目の要素の名前
[1] 3        ←その値
```

要素に名前があると，それを用いて要素にアクセスできる．

例. 名前を指定してリストの要素にアクセスする（先の例の続き）

```
> L[["e2"]]  Enter        ←"e2"の名前を持つ要素を参照
[1] 2                     ←"e2"の名前を持つ要素の値
> L[[3]]  Enter           ←もちろん自然数の添字で要素を
[1] 3                       参照することも可能である
```

名前を指定してリストの要素にアクセスするには「$」を用いて

リスト$名前

と記述する方法もある．

例. 名前を指定してリストの要素にアクセスする（先の例の続き）

```
> L$e3  Enter        ←"e3"の名前を持つ要素を参照
[1] 3        ←その値
```

この形の記述で新規に要素を追加することもできる．

例. 名前を指定した形でリストに要素を新規に追加する（先の例の続き）

```
> L$e4 <- 4   Enter        ←"e4"の名前を持つ要素を新規に追加
> L   Enter     ←内容確認
$e1          ← 1 番目の要素の名前
[1] 1        ←その値
$e2          ← 2 番目の要素の名前
[1] 2        ←その値
$e3          ← 3 番目の要素の名前
[1] 3        ←その値
$e4          ←新たな要素の名前
[1] 4        ←その値
```

関数 names はリストから要素の名前のベクトルを取り出すこともできる.

例. リストの要素の名前を取り出す（先の例の続き）

```
> names( L )   Enter        ←リスト L の要素の名前を参照
[1] "e1" "e2" "e3" "e4"        ←要素の名前が文字列のベクトルとして得られている
```

■ リストの要素に名前を与える別の方法

次のように記述することで，リストを作成する段階で要素に名前を与えることができる.

書き方：　list(名前 1=値 1, 名前 2=値 2, … , 名前 n=値 n)

この場合の名前の記述は文字列形式ではなくクォート「"」を付けない.

例. リスト作成の段階で要素に名前を与える

```
> L <- list( e1=1, e2=2, e3=3 )   Enter        ←要素が名前を持つリストの作成
> L   Enter     ←内容確認
$e1          ← 1 番目の要素の名前
[1] 1        ←その値
$e2          ← 2 番目の要素の名前
[1] 2        ←その値
$e3          ← 3 番目の要素の名前
[1] 3        ←その値
```

2.1.10　データフレーム

行，列の形式のデータ構造に**データフレーム**がある．データフレームは matrix オブジェクトや array オブジェクトとは別のものである．

2.1.10.1　データフレームの作成

次のように記述することでデータフレームを作成することができる．

書き方：　data.frame(列名 1=ベクトル 1, 列名 2=ベクトル 2, …, 列名 n=ベクトル n)

「列名 1」～「列名 n」の名前の列を持ち，それぞれのデータが「ベクトル 1」～「ベクトル n」であるようなデータフレームを返す．

例. データフレームの作成

```
> D <- data.frame( c1=c(1,2), c2=c(3,4), c3=c(5,6) )  Enter      ←データフレーム D の作成
> D   Enter          ←内容確認
  c1 c2 c3           ←列名
1  1  3  5           ←1 行目
2  2  4  6           ←2 行目
```

データフレームの行には名前を付けることができるが，上の例では各行に名前を与えていない．その場合は自然数の名前が付けられる．行に名前を付けるには rownames 関数を使用する．

例. データフレームの行に名前を付ける（先の例の続き）

```
> rownames(D) <- c("d1","d2")  Enter      ←データフレーム D の各行に名前を付ける
> D   Enter          ←内容確認
   c1 c2 c3          ←列名
d1  1  3  5          ←1 行目
d2  2  4  6          ←2 行目
```

各行に名前が付けられていることが確認できる．

■ 行列データ（matrix オブジェクト）からデータフレームを作成する

data.frame の引数に行列データを与える．

例. 行列からデータフレームを作成する

```
> m <- matrix( 1:6, nrow=2, ncol=3 )  Enter      ←行列 m の作成
> D <- data.frame(m)  Enter      ←行列 m からデータフレーム D を作成する
> D   Enter          ←内容確認
  X1 X2 X3           ←列名
1  1  3  5           ←1 行目
2  2  4  6           ←2 行目
```

この例では列名を持たない行列からデータフレームを作成している．その場合は作成されるデータフレームには自動的に「X1」「X2」… の列名が与えられる．

列名や行名を持つ行列からデータフレームを作成すると，元の行列の列名，行名がデータフレームに反映される．

例. 列名，行名を持つ行列を作成する

```
> m <- matrix( 1:6, nrow=2, ncol=3 )  Enter      ←行列 m の作成
> colnames(m) <- c("c1","c2","c3")  Enter      ←行列 m に列名を設定
> rownames(m) <- c("d1","d2")  Enter          ←行列 m に行名を設定
```

例. 先に作成した行列からデータフレームを作成する（先の例の続き）

```
> D <- data.frame(m)  Enter          ←行列 m からデータフレーム D を作成する
> D  Enter          ←内容確認
   c1 c2 c3          ←列名
d1  1  3  5          ←1 行目
d2  2  4  6          ←2 行目
```

2.1.10.2 データフレームの要素へのアクセス

データフレームの個々の要素には行列の場合と同様の方法でアクセスできる.

例. データフレームの要素へのアクセス（先の例の続き）

```
> D["d1","c2"]  Enter     ←データフレーム内の位置を指定して要素を参照
[1] 3                     ←得られた値
> D["d1","c2"] <- 30  Enter     ←指定した要素に値を設定
> D  Enter          ←内容確認
   c1 c2 c3          ←列名
d1  1 30  5          ←1 行目
d2  2  4  6          ←2 行目
```

行の指定を省略した添字を付けると指定した列がベクトルの形で得られる.

例. 列ベクトルへのアクセス（先の例の続き）

```
> D[,"c1"]  Enter     ←列 "c1" の参照
[1] 1 2                   ←ベクトルの形で得られる
```

また，括弧 '[…]' に添字を 1 つだけ指定する形式では，列にリストの形でアクセスできる.

例. リストの形で列にアクセス（先の例の続き）

```
> D["c1"]  Enter     ←列 "c1" の参照
   c1          ←リストの形で得られる
d1  1
d2  2
```

参考）

データフレームはリストを応用して実現されており，括弧 '[[…]]' の添字で列にアクセスすることができる.

例. 列ベクトルへのアクセス（先の例の続き）

```
> D[["c1"]]  Enter     ←列 "c1" の参照
[1] 1 2                   ←ベクトルの形で得られる
```

括弧 '[…]' のよる添字を付ける際に列指定を省略すると，指定した行がデータフレームの形で得られる.

例. 行を取り出す（先の例の続き）

```
> D2 <- D["d1",]  Enter     ←行 "d1" の参照
> D2  Enter          ←内容確認
   c1 c2 c3          ←列名
d1  1 30  5          ←1 行目
```

また，この形で得られた行は unlist 関数でベクトルに変換することができる.

例. 得られた行をベクトルに変換（先の例の続き）

```
> unlist(D2)  Enter     ←ベクトルに変換
c1 c2 c3                  ←要素が名前を持つベクトル
 1 30  5
```

2.2 制御構造

2.2.1 反復 (1)： for

for 文を用いると，文の実行や式の評価の反復ができる．

書き方： for (変数 in ベクトル) 実行部

「ベクトル」の中から要素を順番に 1 つずつ取り出して「変数」に割り当て，それを用いた「実行部」を実行する．この処理を「ベクトル」の先頭の要素から最後の要素にかけて繰り返し実行する．例として，値を出力する print 関数の評価を繰り返し行う処理を次に示す．

例. print 関数の評価の反復

```
> for ( x in 1:3 ) print(x)   Enter        ←反復処理
[1] 1                    ← 1 周目の処理の結果
[1] 2                    ← 2 周目の処理の結果
[1] 3                    ← 3 周目の処理の結果
```

print 関数は引数に与えられた値を出力するものであり，これがベクトル 1:3 の全要素に対して順番に実行されたことがわかる．

for 文の反復対象とする実行部は，複数の行にまたがる**ブロック**の形を取ることもできる．

> **《ブロック》**
> 括弧 {…} を用いると複数の行に渡る複数の文や式を 1 つの**ブロック**としてまとめることができる．ブロックを実行すると，内部に記述された文や式が 1 つずつ順番に実行される．この際，ブロック内に記述された最後の式の値がそのブロック自体の戻り値となる．また，ブロック内では余計な空白文字（半角）や TAB は無視されるので，それらを用いてインデントを表現することができる．

例. for 文でブロックを実行する

```
> for ( x in 1:3 ) {   Enter   ←ブロックの記述（ここから）
+     y <- x*2   Enter
+     print(y)   Enter
+ }   Enter                    ←ブロックの記述（ここまで）
[1] 2                    ← 1 周目の処理の結果
[1] 4                    ← 2 周目の処理の結果
[1] 6                    ← 3 周目の処理の結果
```

この例では y <- x*2 と print(y) の 2 つの式から成るブロックを for 文で実行している．またこの例のように，ターミナル環境でブロックを記述する際，ブロック内で Enter を押すと次の行の先頭に「+」が自動的に表示される．

2.2.1.1 様々なデータ構造に対する for

for 文はベクトル以外のデータ構造に対しても使用することができ，

for (変数 in データ構想) 実行部

として実行するすることが可能である．

例. リストに対する for 文

```
> L <- list("a","b","c")   Enter   ←リスト L
> for (x in L ) {   Enter     ←リスト L に対する for 文
+     print(x)   Enter
+ }   Enter     ← for 文の終了
[1] "a"     ←出力
[1] "b"
[1] "c"
```

リストの要素が順番に取り出されていることがわかる.

例. 行列に対する for 文

```
> m <- matrix(1:4,nrow=2,ncol=2)  Enter  ←行列 m
> m  Enter    ←内容確認
     [,1] [,2]       ←行列の内容
[1,]    1    3
[2,]    2    4
> for (x in m) {  Enter    ←行列 m に対する for 文
+     print(x)  Enter
+ }  Enter    ← for 文の終了
[1] 1    ←出力
[1] 2
[1] 3
[1] 4
```

カラム毎の要素が順番に要素が取り出されていることがわかる.

例. データフレームに対する for 文

```
> D <- data.frame(m)  Enter  ←データフレーム D
> D  Enter    ←内容確認
  X1 X2       ←データフレームの内容
1  1  3
2  2  4
> for (x in D) {  Enter    ←データフレーム D に対する for 文
+     print(x)  Enter
+ }  Enter    ← for 文の終了
[1] 1 2    ←出力
[1] 3 4
```

各列がベクトルとして順番に取り出されていることがわかる.

2.2.2 反復 (2)： while

while 文は，与えた条件式が TRUE である間，文の実行や式の評価を反復する.

書き方： while (条件式) 実行部

「条件式」が TRUE である間「実行部」の実行を繰り返す.

次に示す例は「x の値が正である間処理を繰り返す」というものである.

例. while 文による反復処理

```
> x <- 3  Enter  ←変数 x に値を設定する
> while ( x > 0 ) {  Enter  ←ブロックの記述（ここから）
+    print(x)  Enter
+    x <- x-1  Enter
+ }  Enter                    ←ブロックの記述（ここまで）
[1] 3                ← 1 周目の処理の結果
[1] 2                ← 2 周目の処理の結果
[1] 1                ← 3 周目の処理の結果
```

2.2.3 反復 (3)： repeat

repeat 文は，際限なく文の実行や式の評価を反復する.

書き方： repeat 実行部

「実行部」を際限なく繰り返す．これはいわゆる**無限ループ**であるが「実行部」の中で break が実行されるとその時点で反復処理を終了して repeat 文の次に処理が移行する．

次に示す例は，先に示した例と同じく「x の値が正である間処理を繰り返す」というものである．

例. repeat 文による反復処理

```
> x <- 3   Enter   ←変数 x に値を設定する
> repeat {   Enter              ←ブロックの記述（ここから）
+     print(x)   Enter
+     x <- x-1   Enter
+     if ( x==0 ) break   Enter   ←変数 x の値が 0 になると反復を終了する
+ }   Enter                       ←ブロックの記述（ここまで）
 [1] 3                  ← 1 周目の処理の結果
 [1] 2                  ← 2 周目の処理の結果
 [1] 1                  ← 3 周目の処理の結果
```

この例では変数 x の値が 0 かどうかを if 文で判定している．if 文については次に説明する．

2.2.4 条件分岐 (1)： if

if 文を用いると，条件式の真偽（TRUE/FALSE）によって処理の選択ができる．

書き方： if (条件式) 実行部 1 else 実行部 2

「条件式」が真（TRUE）の場合に「実行部 1」を，偽（FALSE）の場合に「実行部 2」を実行する．条件式が偽の場合の処理を省略することもでき，その場合は else 以降を省略する．

例. if 文による値の正負の判定

```
> x <- 3   Enter   ←変数 x に値を設定する
> if ( x>0 ) {   Enter                ←「x が正かどうか？」の判定
+     print('x は正です．')   Enter     ←「x が正」の場合の処理
+ } else {   Enter                      ←「x が正でない」
+     print('x は負です．')   Enter          場合の処理
+ }   Enter                             ←ブロックの終了
 [1] "x は正です．"                     ←実行結果
```

課題. 上の例において，変数 x の値が負である場合についても試みよ．

2.2.4.1 複数の条件による分岐

if 文の else の後には更に別の if 文を記述することができるので，これを応用すると複数の条件による分岐が実現できる．

《複数の条件による分岐》

```
if （条件式 1）{
    実行部 1
} else if （条件式 2）{
    実行部 2
      ⋮
} else {
    実行部 n
}
```

「条件式 1」を満たす場合は「実行部 1」を実行し，「条件式 2」を満たす場合は「実行部 2」を実行し，… という様に複数の条件分岐を実現する．またこの場合，全ての条件式を満たさなかった場合に「実行部 n」を実行する．

次に示す例は，体重と身長からボディマス指数を算出し，その値から複数の条件を判定するものである．

例. 体重 (kg) と身長 (m) からボディマス指数を算出して状態を判定する

```
> w <- 65   Enter                    ←体重 65kg を変数 w に割り当てる
> h <- 1.7   Enter                   ←身長 1.7m を変数 h に割り当てる
> bmi <- w/h/h   Enter               ←ボディマス指数を算出して変数 bmi に割り当てる
> if ( bmi < 18.5 ) {   Enter                          ←「bmi が 18.5 未満か？」
+    sprintf("%5.2f:痩せています．",bmi)   Enter       ←その場合の出力
+ } else if ( bmi < 25.0 ) {   Enter                   ←「bmi が 25 未満か？」
+    sprintf("%5.2f:正常範囲です．",bmi)   Enter       ←その場合の出力
+ } else {   Enter                                     ←上記のどの条件も満たさない
+    sprintf("%5.2f:太っています．",bmi)   Enter          場合の出力
+ }   Enter                                 ←ブロックの終了
[1] "22.49:正常範囲です．"                  ←実行結果
```

この例では体重を変数 w に，身長を変数 h に与え，それらからボディマス指数を算出して変数 bmi に与えており，更にその値から体重の状態（痩せ／正常／肥満）を判定している．この例で使用している sprintf 関数は書式整形機能を持った出力関数[17] である．

課題. 上の例において，身長，体重を様々に変えて試みよ．

2.2.5 条件分岐 (2)： switch

switch 関数を用いると，与えられた値によって処理の選択ができる．

2.2.5.1 自然数による分岐

switch 関数は，与えられた自然数の値に応じて処理を選択することができる．

書き方： switch(値, 実行部 1, 実行部 2, … , 実行部 n)

自然数の「値」に対応する「実行部」を実行する．switch 関数は，実行部の戻り値をその戻り値とする．
また「値」に対応する実行部が存在しない場合は switch 関数は NULL を返す．

例. 自然数による分岐

```
> x <- 2   Enter   ←変数 x に値を設定する
> r <- switch( x,   Enter   ← x の値で処理を選択する
+     print("1 です．"),   Enter   ← x が 1 の場合の処理
+     print("2 です．"),   Enter   ← x が 2 の場合の処理
+     print("3 です．")   Enter   ← x が 3 の場合の処理
+ )   Enter                ← switch 関数の記述の終了
[1] "2 です．"               ←処理結果
> r   Enter                 ← switch 関数の戻り値を調べる
[1] "2 です．"               ←戻り値（実行した print 関数の戻り値）
```

この例では，変数 x の値が 2 であるので 2 番目の実行部である「print("2 です．")」を実行している．またその結果，当該 print 関数の戻り値「"2 です．"」が switch 関数の戻り値として変数 r に得られている．

課題. 上の例において，x の値を様々に変えて試みよ．

2.2.5.2 文字列の値による分岐

switch 関数は，与えられた文字列の値に応じて処理を選択することができる．

書き方： switch(値, 値 1 = 実行部 1, 値 2 = 実行部 2, … , 実行部 n)

[17] 詳しくは「2.3.1 sprintf 関数による書式整形」（p.44）で解説する．

与えられた文字列の「値」によって対応する実行部を実行する．この書き方の場合のイコール「=」は代入を意味するものではなく，文字列の「値」が「値 1」の場合に「実行部 1」を実行し，文字列の「値」が「値 2」の場合に「実行部 2」を実行し… という様に，値と実行部を対応させるものである.「値 1」「値 2」… のどれにも該当しない文字列が第 1 引数に与えられた場合は「実行部 n」が実行される．

例. 文字列の値による分岐

```
> x <- "orange"  Enter  ←変数 x に文字列を設定する
> r <- switch( x,  Enter  ← x の値で処理を選択する
+     "apple"  = print("「りんご」です．"),  Enter  ← x が ”apple” の場合の処理
+     "orange" = print("「みかん」です．"),  Enter  ← x が ”orange” の場合の処理
+     "grape"  = print("「ぶどう」です．"),  Enter  ← x が ”grape” の場合の処理
+     print("その他です．")  Enter  ←上記のどれにも該当しない場合の処理
+ )  Enter  ← switch 関数の記述の終了
[1] "「みかん」です．"              ←処理結果
```

switch 関数の引数に記述する文字列のパターンは引用符「"」を省略することができる.（次の例）

例. 文字列の値による分岐（引用符省略，先の例と同様の処理）

```
> x <- "orange"  Enter  ←変数 x に文字列を設定する
> r <- switch( x,  Enter  ← x の値で処理を選択する
+     apple  = print("「りんご」です．"),  Enter  ← x が ”apple” の場合の処理
+     orange = print("「みかん」です．"),  Enter  ← x が ”orange” の場合の処理
+     grape  = print("「ぶどう」です．"),  Enter  ← x が ”grape” の場合の処理
+     print("その他です．")  Enter  ←上記のどれにも該当しない場合の処理
+ )  Enter  ← switch 関数の記述の終了
[1] "「みかん」です．"              ←処理結果
```

課題. 上の例において，x の値を様々に変えて試みよ．

2.3 入出力

print 関数を使用すると，オブジェクトの値をターミナル環境に出力することができる．

書き方：　print(値)

出力対象の「値」は 1 つだけであり，この「値」が print 関数の戻り値である．

例. print 関数の戻り値

```
> r <- print( "出力内容" )  Enter      ← print 関数を評価することで
[1] "出力内容"              ←値がターミナルウィンドウに出力される．
> r   Enter                 ← print 関数の戻り値は
[1] "出力内容"              ←引数に与えたものである．
```

2.3.1 sprintf 関数による書式整形

複数のオブジェクトの値をまとめて出力するには sprintf 関数[18] を応用すると良い．この関数は，複数のオブジェクトの値を与えられた書式に従って整形して 1 つの文字列を作成するものである．

書き方：　sprintf(書式文字列, 値 1, 値 2, …, 値 n)

「書式文字列」に従って「値 1」〜「値 n」を整形した文字列を返す．

例. sprintf 関数による書式整形

```
> v1 <- 1; v2 <- 2; v3 <- 3.14; v4 <- "abc"  Enter     ←変数 v1〜v4 に様々な値を与える
> s <- sprintf( "%2d:%03d:%5.2f:%5s", v1, v2, v3, v4 )  Enter     ←それらを書式整形
> s   Enter                          ←書式整形結果の確認
[1] " 1:002:  3.14:   abc"           ←書式整形されている
```

この例では v1 の値「1」を 2 桁の形式で，v2 の値「2」を前にゼロを埋める形の 3 桁の形式で，v3 の値「3.14」を小数部 2 桁かつ全体 5 桁の形式で，v4 の値「"abc"」を 5 桁の形式で整形し，それを文字列として変数 s に与えている．

書式文字列はパーセント記号「%」で始まる**書式指定**（後で解説する）を含むものであり，その部分に対応する値を整形して埋め込む形で処理結果を得る．

2.3.1.1 書式指定について

sprintf 関数で使用する書式文字列はパーセント記号「%」で始まる**書式指定**を含む．書式指定は図 5 に示す様に**フラグ**，**フィールド幅**，**変換指定子**から成る．

% フラグ フィールド幅 変換指定子

図 5: 書式指定の基本的な構成

書式指定の中で最も重要なものが**変換指定子**であり，これによってデータの型（値の種類）が決まる．また**フィールド幅**には書式整形の桁数の最小値を指定する．書式整形は基本的に右揃えであるが**フラグ**にマイナス記号「-」を与えると左揃えとなる．またフラグにゼロ「0」を与えると，値の整形結果がフィールド幅に満たない場合に左側をゼロで満たす．書式指定の例を表 7〜10 に示す．

表 7: 書式指定の例（整数）

変換指定子	例			
d	sprintf(":%5d:", 123)	→	": 123:"	5 桁（右揃え）
	sprintf(":%-5d:",123)	→	":123 :"	5 桁（左揃え）
	sprintf(":%05d:",123)	→	":00123:"	5 桁（右揃え, 前ゼロ）

[18] C 言語の sprintf 関数を応用したものである．

表 8: 書式指定の例（浮動小数点数）

変換指定子	例			
f	`sprintf(":%7.2f:", 3.141592)`	→	`":   3.14:"`	小数部 2 桁, 全体 7 桁（右揃え）
	`sprintf(":%7.3f:", 3.141592)`	→	`":  3.142:"`	小数部 3 桁, 全体 7 桁（右揃え）
	`sprintf(":%-7.3f:",3.141592)`	→	`":3.142  :"`	小数部 3 桁, 全体 7 桁（左揃え）
	`sprintf(":%07.3f:",3.141592)`	→	`":003.142:"`	小数部 3 桁, 全体 7 桁（右揃え, 前ゼロ）

表 9: 書式指定の例（浮動小数点数：指数表現）

変換指定子	例			
e	`sprintf(":%10.2e:",0.000001)`	→	`":  1.00e-06:"`	小数部 2 桁, 全体 10 桁
g	`sprintf(":%7.2g:",0.000001)`	→	`":  1e-06:"`	小数部 2 桁, 全体 7 桁
	`sprintf(":%7.2g:",0.01)`	→	`":   0.01:"`	

注）変換指定子「e」「g」の場合もフラグ「-」「0」などが使用できる.
変換指定子「g」の場合は，整形対象の値によって指数表現にするかどうかが自動的に判断される.

表 10: 書式指定の例（文字列，16 進数，8 進数）

変換指定子	例			
s	`sprintf(":%5s:", "abc")`	→	`":  abc:"`	5 桁の文字列（右揃え）
	`sprintf(":%-5s:", "abc")`	→	`":abc  :"`	5 桁の文字列（左揃え）
x	`sprintf(":%4x:",255)`	→	`":  ff:"`	4 桁の 16 進数（右揃え）
	`sprintf(":%-4x:",255)`	→	`":ff  :"`	4 桁の 16 進数（左揃え）
	`sprintf(":%04x:",255)`	→	`":00ff:"`	4 桁の 16 進数（右揃え, 前ゼロ）
o（オー）	`sprintf(":%4o:",255)`	→	`": 377:"`	4 桁の 8 進数（右揃え）
	`sprintf(":%-4o:",255)`	→	`":377 :"`	4 桁の 8 進数（左揃え）
	`sprintf(":%04o:",255)`	→	`":0377:"`	4 桁の 8 進数（右揃え, 前ゼロ）

2.3.2 CSV ファイルへの出力 (1)： write.table

CSV は，各種の値をコンマ「,」で区切る形で記述するテキストデータの形式であり，RFC 4180 で標準化されている．CSV 形式のデータは行，列から成る 2 次元のデータを表現する際に多く用いられており，多くののアプリケーションソフトウェアが CSV 形式のテキストファイルの入出力に対応している.

write.table を使用することでデータフレームの内容をファイルに出力できる．write.table は後で説明する write.csv よりも汎用性が高く，コンマ以外の区切り文字を持つファイルを作成することもできる．まずはじめに，この write.table を用いてデータフレームの内容を CSV ファイルに出力する方法について解説する.

処理の事例を挙げて解説する，まず，サンプルとして次のようなデータフレームを用意する.

例. サンプルのデータフレーム D の作成

```
> D <- data.frame(linear=c(1,2,3),square=c(1,4,9),cubic=c(1,8,27))  Enter  ←サンプル
> rownames(D) <- c("d1","d2","d3")  Enter  ←各行に名前を与える
> D  Enter              ←内容確認
   linear square cubic      ←データフレームの内容
d1      1      1     1
d2      2      4     8
d3      3      9    27
```

この例で作成したデータフレーム D を write.table でファイル csvtest11.csv に出力する例を次に示す.

例. データフレーム D をファイル csvtest11.csv に出力（先の例の続き）

```
> write.table(D,"csvtest11.csv")  Enter        ←ファイルに出力
```

この例のように

write.table(データフレーム, 出力ファイルのパス)

とすることで「データフレーム」の内容を「出力ファイルのパス」で指定したファイルに出力する．先の処理の結果，次に示すような内容のファイル csvtest11.csv がカレントディレクトリ作成される．

ファイル：csvtest11.csv

```
1 "linear" "square" "cubic"
2 "d1" 1 1 1
3 "d2" 2 4 8
4 "d3" 3 9 27
```

このファイルはデータ項目の区切りが空白文字となっており，厳密には CSV ファイルの形式とは異なる．区切り文字をコンマ「,」にして出力するには write.table に引数「sep=区切り文字」を与える．(次の例)

例. 出力ファイルのデータ項目の区切り文字をコンマにする（先の例の続き）

```
> write.table(D,"csvtest12.csv",sep=",")  Enter      ←区切り文字を指定してファイルに出力
```

この処理の結果，次に示すような内容のファイル csvtest12.csv が作成される．

ファイル：csvtest12.csv

```
1 "linear","square","cubic"
2 "d1",1,1,1
3 "d2",2,4,8
4 "d3",3,9,27
```

2.3.2.1 出力ファイルの行見出し，列見出しの有無の設定

先の例で作成したファイル csvtest12.csv には各行に見出し "d1"〜"d2" が付けられており，ファイル中にはこれに対応する列の見出しが無い．このようなファイルを R 以外のシステム（他のアプリケーションソフトウェア）で入力に使用すると，列の見出しと列のデータ項目の対応が意図しないもの（列見出しがずれたもの）になる[19] ．従って，汎用性のある（他のアプリケーションソフトウェアで使用する）CSV データファイルを作成するには行の見出しを出力しない方が良い．このためには，write.table の引数に「row.names=FALSE」を与える．(次の例)

例. 行見出しを出力せずにファイルを作成する（先の例の続き）

```
> write.table(D,"csvtest13.csv",sep=",",row.names=FALSE)  Enter
```

この処理の結果，次に示すような内容のファイル csvtest13.csv が作成される．

ファイル：csvtest13.csv

```
1 "linear","square","cubic"
2 1,1,1
3 2,4,8
4 3,9,27
```

列の見出しを付けない形のファイルを出力することも可能である．その場合は write.table の引数に「col.names=FALSE」を与える．(次の例)

例. 行見出し，列見出し共に付けない形の出力（先の例の続き）

```
> write.table(D,"csvtest14.csv",sep=",",row.names=FALSE,col.names=FALSE)  Enter
```

この処理の結果，次に示すような内容のファイル csvtest14.csv が作成される．

ファイル：csvtest14.csv

```
1 1,1,1
2 2,4,8
3 3,9,27
```

[19] この問題を回避するには，後で説明する write.csv を使用する方が有利である．

2.3.2.2　出力内容の引用符の有無の設定

先の例からもわかるように，ファイルへの出力内容の文字列の部分には二重引用符「"…"」がある．引用符を使わない形でデータを出力するには write.table に引数「quote=FALSE」を与える．(次の例)

例. 引用符を付けずにデータを出力（先の例の続き）

```
> write.table(D,"csvtest15.csv",sep=",",row.names=FALSE,quote=FALSE)  Enter
```

この処理の結果，次に示すような内容のファイル csvtest15.csv が作成される．

ファイル：csvtest15.csv

```
1 linear,square,cubic
2 1,1,1
3 2,4,8
4 3,9,27
```

2.3.3　CSV ファイルへの出力 (2)：　write.csv

先に解説した write.table は重要であるが，特に CSV 形式ファイルの出力においてはここで説明する write.csv が便利である．

書き方：　write.csv(データフレーム, 出力ファイルのパス)

この記述により「データフレーム」の内容を「出力ファイルのパス」で指定したファイルに出力する．

ここでも処理の事例を挙げて解説する，まず，サンプルとして次のようなデータフレームを用意する．

例. サンプルのデータフレーム D2 の作成

```
> D2 <- data.frame(linear=c(4,5,6),square=c(16,25,36),cubic=c(64,125,216))  Enter  ←サンプル
> rownames(D2) <- c("d4","d5","d6")  Enter  ←各行に名前を与える
> D2  Enter  ←内容確認
   linear square cubic      ←データフレームの内容
d4      4     16    64
d5      5     25   125
d6      6     36   216
```

このデータフレーム D2 の内容をファイル csvtest21.csv に出力する例を示す．

例. データフレーム D2 をファイル csvtest21.csv に出力（先の例の続き）

```
> write.csv(D2,"csvtest21.csv")  Enter
```

この処理の結果，次に示すような内容のファイル csvtest21.csv が作成される．

ファイル：csvtest21.csv

```
1 "","linear","square","cubic"
2 "d4",4,16,64
3 "d5",5,25,125
4 "d6",6,36,216
```

データ項目がコンマで区切られており，列の見出しと列の項目が正しく対応していることがわかる．

2.3.3.1　引用符，行見出しの有無の設定

write.table の場合と同様に，write.csv には「quote=」，「row.names=」といった引数を与えることができる．

例. 引数「quote=FALSE」を与えて出力（先の例の続き）

```
> write.csv(D2,"csvtest22.csv",quote=FALSE)  Enter
```

この処理の結果，次に示すような内容のファイル csvtest22.csv が作成される．

ファイル：csvtest22.csv

```
1 ,linear,square,cubic
2 d4,4,16,64
3 d5,5,25,125
4 d6,6,36,216
```

例. 引数「row.names=FALSE」を与えて出力（先の例の続き）

```
> write.csv(D2,"csvtest23.csv",row.names=FALSE)  Enter
```

この処理の結果，次に示すような内容のファイル csvtest23.csv が作成される．

ファイル：csvtest23.csv

```
1 "linear","square","cubic"
2 4,16,64
3 5,25,125
4 6,36,216
```

2.3.4 CSV ファイルからの入力 (1)： read.table

read.table を使用すると write.table で出力したファイルの内容を読み込んでデータフレームにすることができる．p.45 の例でデータフレームの内容をファイル csvtest11.csv を作成した．

ファイル：csvtest11.csv

```
1 "linear" "square" "cubic"
2 "d1" 1 1 1
3 "d2" 2 4 8
4 "d3" 3 9 27
```

このファイルはデータ項目の区切りが空白文字であり，列の見出しと列の項目の対応が独特のもの（ずれた形）である．このファイルから，元のデータフレームの形式をそのまま再現する形で読み込むには次のような処理を行う．

例. p.45 の処理で作成した csvtest11.csv を読み込む

```
> D <- read.table("csvtest11.csv")  Enter     ←ファイルの読み込み
> D  Enter                                    ←内容確認
   linear square cubic      ←データフレームの内容
d1      1      1     1
d2      2      4     8
d3      3      9    27
```

ファイル作成時のデータフレームの内容が同じ形式で得られていることがわかる．同様に，write.talbe の引数に「sep=","」を指定して区切り文字をコンマにして作成したファイル csvtest12.csv（p.46）を読み込むには次のような処理を行う．

例. p.46 のファイル csvtest12.csv を読み込む

```
> D <- read.table("csvtest12.csv",sep=",")  Enter     ←ファイルの読み込み
> D  Enter                                            ←内容確認
   linear square cubic      ←データフレームの内容
d1      1      1     1
d2      2      4     8
d3      3      9    27
```

2.3.5 CSV ファイルからの入力 (2)： read.csv

先に解説した read.table は重要であるが，特に CSV 形式ファイルの入力においてはここで説明する read.csv が便利である．

書き方：　read.csv(入力ファイルのパス)

この記述により「入力ファイルのパス」で指定したファイルの内容を読み込み，データフレームとして返す．

p.46 に例示した処理で作成したファイル csvtest13.csv を読み込む処理を示す．

ファイル：csvtest13.csv

```
1 "linear","square","cubic"
2 1,1,1
3 2,4,8
4 3,9,27
```

このファイルは先頭行が見出し（項目名）であり，行の見出しは無い．このファイルは次のようにして読み込むことができる．

例. ファイル csvtest13.csv を読み込む

```
> D <- read.csv("csvtest13.csv")  Enter     ←読み込み処理
> D  Enter                ←内容確認
  linear square cubic       ←データフレームの内容
1      1      1     1
2      2      4     8
3      3      9    27
```

ファイルには行見出しが無いので，得られたデータフレームの行には自然数の見出し（行の名前）が付けられている．

2.3.5.1 見出し行の無い CSV ファイルの読み込み

p.46 に例示した処理で作成したファイル csvtest14.csv を読み込む処理を示す．

ファイル：csvtest14.csv

```
1 1,1,1
2 2,4,8
3 3,9,27
```

このように行見出し，列見出し共に無いファイルを読み込むには read.csv の引数に「`header=FALSE`」を与える．

例. ファイル csvtest14.csv を読み込む

```
> D <- read.csv("csvtest14.csv",header=FALSE)  Enter     ←読み込み処理
> D  Enter                ←内容確認
  V1 V2 V3       ←データフレームの内容
1  1  1  1
2  2  4  8
3  3  9 27
```

得られたデータフレームに列の見出し V1，V2，V3 が自動的に与えられている．

2.3.6 テキストファイルの入力

テキストファイルは**文字コード**のバイト値[20] から成るデータファイルであり，可読な文字から成るデータ[21] を保存する目的で広く用いられている．文字コードを表現する体系（**エンコーディング**）として有名なものに **ISO-2022-JP**（いわゆる **JIS コード**）や **JIS X 0208**（いわゆる**シフト JIS**，**Shift_JIS**），**UTF-8** などが有名である．エンコーディングに関する詳細については他の文献や資料を参照されたい．

scan 関数を使用するとテキストファイルの内容を文字列のベクトルとして読み取ることができる．

書き方：　scan(ファイルのパス, what=データの種類)

「ファイルのパス」で示したファイルからデータを読み取った内容を返す．「データの種類」には，値を読み取る際のデータの種類を指定する．ファイルの内容を文字列のベクトルとして読み取る場合は `what=character()` とする．こ

[20] 正確には「文字コードを符号化したバイト値」である．

[21] いわゆる**テキストデータ**．

の方法でテキストファイルを読み取ると，ファイルの内容を空白文字や改行文字で区切った単語のベクトルとして返す．

次に示すテキストファイル R_wiki_e.txt の内容を読み取る例を挙げて説明する．

テキストファイル：R_wiki_e.txt

```
1 R is a programming language for statistical computing and graphics
2 supported by the R Core Team and the R Foundation for Statistical
3 Computing. Created by statisticians Ross Ihaka and Robert Gentleman,
4 R is used among data miners and statisticians for data analysis and
5 developing statistical software. Users have created packages to
6 augment the functions of the R language.
```

このファイルの内容を読み込む例を次に示す．

例. テキストファイルの内容を単語のベクトルとして読み取る

```
> tx <- scan( "R_wiki_e.txt", what=character() )  Enter    ←読み込み処理
Read 58 items                ←読み込んだ項目の数（単語数）が報告される
> tx   Enter                 ←内容確認
 [1] "R"            "is"           "a"            "programming"      ←単語のベクトル
 [5] "language"     "for"          "statistical"  "computing"
                          ;
                    （途中省略）
                          ;
[53] "the"          "functions"    "of"           "the"
[57] "R"            "language."
```

テキストファイルの内容を空白文字や改行で区切った単語として読み取り，それらを文字列ベクトル tx として得ている．

scan 関数でテキストファイルを読み込む際，空白文字以外の区切り文字を指定するには引数「sep=**区切り文字**」を与える．(区切り文字には 1 バイトで表現される文字を与えること)　これを応用すると，テキストファイルの内容を行単位の文字列のベクトルとして読み取ることができる．(次の例)

例. テキストファイルの行単位の読取り

```
> tx2 <- scan( "R_wiki_e.txt", what=character(), sep="\n" )  Enter    ←読み込み処理
Read 6 items                 ←読み込んだ項目の数（行数）が報告される
> tx2   Enter                ←内容確認
[1] "R is a programming language for statistical computing and graphics"    ←行毎のデータ
[2] "supported by the R Core Team and the R Foundation for Statistical"
[3] "Computing.  Created by statisticians Ross Ihaka and Robert Gentleman,"
[4] "R is used among data miners and statisticians for data analysis and"
[5] "developing statistical software.  Users have created packages to"
[6] "augment the functions of the R language.
```

引数「sep=」を省略した場合は空白文字と改行が区切り文字となるが，この例の様に「sep="\n"」とすると改行のみが区切り文字となる．

2.3.6.1　日本語テキストの入力

scan 関数で日本語のテキストファイルの内容を読み込むことができる．次に示すテキストファイル R_wiki_j_utf8.txt の内容を読み取る例を挙げて説明する．

テキストファイル：R_wiki_j_utf8.txt

```
1 R言語はオープンソース・フリーソフトウェアの統計解析向けのプログラミング
2 言語及びその開発実行環境である。
3 R言語はR Development Core Teamによりメンテナンスと拡張がなされている。
4 R言語のソースコードは主にC言語、FORTRAN、そしてRによって開発された。
```

このファイルの内容を読み込む例を次に示す.

例. テキストファイルの内容を単語のベクトルとして読み取る

```
> tx3 <- scan( "R_wiki_j_utf8.txt", what=character(), sep="\n" )  Enter    ←読み込み処理
Read 4 items                    ←読み込んだ項目の数（行数）が報告される
> tx3  Enter                    ←内容確認
[1] "R 言語はオープンソース・フリーソフトウェアの統計解析向けのプログラミング"    ←行毎のデータ
[2] "言語及びその開発実行環境である。"
[3] "R 言語は R Development Core Team によりメンテナンスと拡張がなされている。"
[4] "R 言語のソースコードは主に C 言語、FORTRAN、そして R によって開発された。"
```

日本語のデータが行毎の文字列ベクトルとして得られていることがわかる．これは Windows 版の R による実行例であり，入力対象のテキストファイルのエンコーディングは暗黙で **UTF-8** である[22] とみなされている．この例では，使用したテキストファイル R_wiki_j_utf8.txt のエンコーディングが UTF-8 であるとしているが，他のエンコーディングによるテキストファイルを同じ方法で扱うと読み取り結果が不適切なものとなる．例えば先の R_wiki_j_utf8.txt と同じ内容を**シフト JIS** のエンコーディングで作成したテキストファイル R_wiki_j_sjis.txt を先と同じ方法で読み取ると次のようになる.

例. シフト JIS のテキストファイルを読み込む試み

```
> tx4 <- scan( "R_wiki_j_sjis.txt", what=character(), sep="\n" )  Enter    ←読み込み処理
Read 4 items                    ←読み込んだ項目の数（行数）が報告される
> tx4  Enter                    ←内容確認
[1] "R¥x8c¥xbe¥x8c¥xea¥x82 ヘオ¥x81[¥x83v¥x83¥x93 ¥x83¥¥¥x81[¥x83X¥x81E¥x83t
¥x83¥x8a¥x81[¥x83¥¥¥x83t¥x83g¥x83E¥x83F¥x83A¥x82
            …（以下省略）…
```

作成された内容と異なる形で得られているように見える[23] ．シフト JIS のエンコーディングで作成されたテキストファイルを正しい形で読み取るには，scan 関数に引数「`fileEncoding="shift-jis"`」を与える.（次の例）

例. シフト JIS のテキストファイルを正しく読み込む

```
> tx4 <- scan("R_wiki_j_sjis.txt",what=character(),sep="\n",fileEncoding="shift-jis") Enter
Read 4 items                    ←読み込んだ項目の数（行数）が報告される
> tx4  Enter                    ←内容確認
[1] "R 言語はオープンソース・フリーソフトウェアの統計解析向けのプログラミング"    ←行毎のデータ
[2] "言語及びその開発実行環境である。"
[3] "R 言語は R Development Core Team によりメンテナンスと拡張がなされている。"
[4] "R 言語のソースコードは主に C 言語、FORTRAN、そして R によって開発された。"
```

2.3.6.2 テキストファイルから数値を読み込む方法

scan 関数は，テキストデータとしてファイルに記述されている数値を読み込むことができる.

書き方： scan(ファイルのパス, what=数値の型, sep=区切り文字)

「数値の型」として `integer()` を指定すると整数として，`double()` を指定すると浮動小数点数として読み込む．数値の各々の値は空白文字と改行で区切られたものとして読み込まれるが，空白文字以外の文字で値が区切られている場合はそれを「区切り文字」の部分に指定する．読み取った値はベクトルとして返される.

次のようなテキストファイル intText01.txt から整数として値を読み込む例を示す.

テキストファイル： intText01.txt

```
1 1 2 3 4 5
```

[22] R の版や実装されているプラットフォームによって異なることがあるので注意されたい.

[23] いわゆる「文字化け」.

例. 空白文字で区切られた数値データを読み込む

```
> v <- scan( "intText01.txt", what=integer() )  Enter      ←整数値の読み込み
Read 5 items                    ←読み込んだ項目の数が報告される
> v  Enter                      ←内容確認
[1] 1 2 3 4 5                   ←読み取った値
```

次に，コンマで区切られたテキストファイル intText02.txt から整数として値を読み込む例を示す.

テキストファイル：intText02.txt

```
1 1,2,3,4,5
```

例. コンマで区切られた数値データを読み込む

```
> v <- scan( "intText02.txt", what=integer(), sep="," )  Enter      ←整数値の読み込み
Read 5 items                    ←読み込んだ項目の数が報告される
> v  Enter                      ←内容確認
[1] 1 2 3 4 5                   ←読み取った値
```

2.3.7 テキストファイルの出力

write 関数を使用すると，値をテキストファイルに出力することができる.

書き方： write(値, 出力ファイルのパス, append=追加出力指定)

「値」を「出力ファイルのパス」で指定したファイルに出力する．引数「append=」を省略すると出力ファイルを新規作成（既存ファイルの場合は上書き）する.「追加出力指定」として TRUE を与えると既存のファイルの末尾に追加出力する.

例. 文字列をファイルに出力する

```
> write( "1 行目", "datText01.txt" )  Enter                ←ファイルを新規作成して出力
> write( "二行目", "datText01.txt", append=TRUE )  Enter      ←上のファイルに追加出力
> write( "さんぎょうめ", "datText01.txt", append=TRUE )  Enter      ←上のファイルに追加出力
```

この処理の結果，次のようなファイル datText01.txt が出来上がる.

出力されたファイル：datText01.txt

```
1 1行 目
2 二 行 目
3 さ ん ぎ ょ う め
```

この例のように，write 関数を複数回実行してファイルの内容を追加した場合，改行によって出力内容が区切られる.

参考）Windows 用の R では出力ファイルのエンコーディングは **UTF-8** である.

複数の要素から成るベクトルを write で出力することもできる．その際に出力項目の区切り文字を指定するには write 関数の引数に「sep=区切り文字」を与える.（引数「sep=」を省略すると出力項目は空白で区切られる）

例. 複数要素から成るベクトルの出力

```
> v <- c(1,2,3,4,5)  Enter                ← 5 個の要素から成るベクトル
> write( v, "datText02.txt" )  Enter        ←区切り文字指定なしで出力
> write( v, "datText03.txt", sep="," )  Enter      ←コンマ区切りで出力
```

この結果，ファイルへの出力結果は次のようになる.

出力されたファイル：datText02.txt（空白区切り）

```
1 1 2 3 4 5
```

出力されたファイル：datText03.txt（コンマ区切り）

```
1 1,2,3,4,5
```

2.3.8　ファイル，ディレクトリに関する処理

システムがファイルの入出力の対象とする**カレントディレクトリ**を確認，あるいは移動する方法について，先の「1.2.2 カレントディレクトリの確認と変更」（p.2）でも少し解説した．ここではファイル，ディレクトリの扱いに関して更に詳細に解説する．サンプルプログラムの実行例は Windows 環境におけるものを示す．

2.3.8.1　カレントディレクトリの移動と確認

カレントディレクトリは setwd で移動（変更）することができる．カレントディレクトリの確認は getwd で行う．

例. カレントディレクトリの確認

```
> getwd()  Enter          ←カレントディレクトリを調べる
[1] "C:/Users/katsu"      ←結果
```

例. カレントディレクトリの移動

```
> setwd("R")  Enter     ←カレントディレクトリを R に移動
> getwd()  Enter        ←確認
[1] "C:/Users/katsu/R"      ←結果
```

例. カレントディレクトリを 1 つ上に移動

```
> setwd("..")  Enter    ←カレントディレクトリを 1 つ上に移動
> getwd()  Enter        ←確認
[1] "C:/Users/katsu"      ←結果
```

setwd には移動先のディレクトリのパスを文字列の形式で与える．この場合**絶対パス**，**相対パス**のどちらも使用可能である．

存在しないディレクトリに移動しようとするとエラーとなる．例えば，カレントディレクトリに存在しないディレクトリ S に移動しようとすると次のようになる．

例. 存在しないディレクトリに移動しようとした例

```
> setwd("S")  Enter     ←移動を試みると…
setwd("S") でエラー:  作業ディレクトリを変更できません          ←エラーとなる
```

参考）setwd の引数に，ディレクトリではなくファイルのパスを与えた場合もエラーとなる．

2.3.8.2　ファイル，ディレクトリの一覧

ディレクトリの一覧を取得するには list.dirs を使用する．

書き方：　list.dirs(path=対象のパス, recursive=再帰指定, full.names=表記指定)

「対象のパス」配下のディレクトリの一覧を文字列のベクトルとして取得する．「再帰指定」にはサブディレクトリを再帰的に検索するかどうかを論理型で指定する．この引数のデフォルトは TRUE となっているので「対象のパス」の直下のディレクトリのみの一覧を取得する場合は FALSE を指定する．「表記指定」に FALSE を指定するとディレクトリの名前のみの一覧が得られるが，TRUE を指定すると「対象のパス」を含めた名前の一覧が得られる．

ファイルの一覧を取得するには list.files を使用する．

書き方：　list.files(path=対象のパス)

「対象のパス」配下のファイルの一覧を文字列のベクトルとして取得する．

2.3.8.3　ファイル，ディレクトリの作成

ディレクトリの作成には dir.create を使用する．

書き方：　dir.create(対象のパス)

「対象のパス」に指定したディレクトリを作成する．「対象のパス」のディレクトリが既に存在する場合は警告メッセージを出力して作成処理を行わない．

ファイルの作成には file.create を使用する．

書き方：　file.create(対象のパス)

「対象のパス」に指定したファイルを作成する．この方法で作成した直後はファイルの内容は空である.「対象のパス」のファイルが既に存在する場合は，既存のファイルを上書きして空のファイルを作成するので注意すること．

2.3.8.4　ファイル，ディレクトリの存在の確認

ディレクトリの存在検査には dir.exists を使用する．

書き方：　dir.exists(対象のパス)

「対象のパス」に指定したディレクトリが存在する場合は TRUE を，存在しない場合は FALSE を返す．

ファイルの存在検査には file.exists を使用する．

書き方：　file.exists(対象のパス)

「対象のパス」に指定したファイルが存在する場合は TRUE を，存在しない場合は FALSE を返す．

2.3.8.5　ファイルの削除

ファイルの削除には file.remove を使用する．

書き方：　file.remove(対象のパス)

「対象のパス」に指定したファイルを削除する．削除の処理が正常に実行されると TRUE を返す．削除しようとするファイルが存在しない場合は警告メッセージを出力して FALSE を返す．

2.4 関数の定義

R における関数は

関数名 (実引数 1, 実引数 2, …, 実引数 n)

の形式で評価され,「実引数 1」〜「実引数 n」の値に対する値（**戻り値**）を返す．また「関数名」の記号自体もその関数の定義内容のオブジェクトを保持する．例えば，与えられたベクトルの要素の平均値を求める関数 mean が R には予め提供されており，次のようにして評価することができる．

例. 関数 mean の評価

```
> v <- c(1,2,3,4,5)   Enter   ←ベクトルを v に与える
> mean( v )   Enter   ←ベクトル v の要素の平均値を求める
[1] 3                 ←得られた値
```

この関数の名前 mean も定義内容のオブジェクトを保持しており，それを参照することができる.（次の例）

例. 関数名 mean の参照（先の例の続き）

```
> mean   Enter   ←関数名 mean の内容を参照する
function (x, ...)                    ←以下，関数名 mean に定義されている内容
UseMethod("mean")
<bytecode:  0x0000000017f03f30>
<environment:  namespace:base>
```

この関数名 mean に定義されている内容も 1 つのオブジェクトとして扱うことできる[24] .（次の例）

例. 関数 mean の定義内容を別の変数記号に割り当てる（先の例の続き）

```
> mean2 <- mean   Enter   ←関数 mean の定義内容を変数 mean2 に割り当てる
> mean2( v )   Enter   ←上記の mean2 を関数として評価する
[1] 3                  ←値が得られている
```

R では利用者が独自の関数を定義することができる．

《関数の定義》

書き方： function(仮引数 1, 仮引数 2, …, 仮引数 n) 定義内容

「仮引数 1」〜「仮引数 n」の値に対する関数を「定義内容」で定義する.「定義内容」の部分は値を算出するための式，あるいはブロックとして記述する．ブロックを記述した場合はブロックの最後の式の値が関数の戻り値となる．この記述は 1 つのオブジェクトであり，これを関数名となる変数に割り当てて使用することができる．

例. 与えられた 2 つの引数の加算結果を返す関数の定義（その 1）

```
> tasu <- function( x, y ) x + y   Enter   ←加算の関数定義を関数名 tasu に与えている
```

このように定義された関数 tasu の評価と，定義内容を確認する例を次に示す．

例. 関数 tasu の評価と定義内容の確認（先の例の続き）

```
> tasu( 2, 3 )   Enter   ←定義した関数の評価
[1] 5                    ←得られた値
> tasu   Enter   ←関数名 tasu の内容を確認する
function( x, y ) x + y          ←定義内容が得られている
```

次に，関数の定義内容をブロックの形式で与える例を示す．

[24] 関数の定義内容も**第一級のオブジェクト**である．

例. 与えられた 2 つの引数の加算結果を返す関数の定義（その 2）

```
> tasu <- function( x, y ) {   Enter   ←関数の定義内容をブロックの形で与える
+     r <- x + y   Enter          ←引数の加算結果を変数 r に与えている
+     r   Enter          ←変数 r の値を戻り値とする
+ }   Enter          ←ブロックの記述の終了
> tasu( 2, 3 )   Enter     ←定義した関数の評価
[1] 5                  ←得られた値
```

この例では，関数定義のブロック内で変数 r に計算結果を得ている．ブロック自体は最終行に記述したものをその値として返すため，変数 r の値がこの関数の戻り値となる．

2.4.1 関数の引数について

関数を実際に評価する際に与える引数（具体的な値を持ったもの）を**実引数**（actual argument）と呼ぶ．また，関数の定義を記述する際に function の後ろに記述する引数（まだ具体的な値を持っていないもの）は**仮引数**（formal argument）と呼ぶ．基本的には，関数の評価時（関数を実際に呼び出す際）に記述した実引数の順番と，関数定義の際に記述した仮引数の順番が対応する形で引数の値が渡される．

関数を評価する際，その関数を定義した際の仮引数の名前を明に指定して（下記参照）値を渡すことができる．

書き方： 関数名 (仮引数名 1=値 1, 仮引数名 2=値 2, …)

例えば次のように定義された関数について考える．

例. サンプルの関数 f

```
> f <- function(x,y) {   Enter          ←関数定義の開始
+     print( paste("x:",as.character(x)) )   Enter
+     print( paste("y:",as.character(y)) )   Enter
+ }   Enter          ←関数定義の終了
```

これは 2 つの引数を取る関数 f を定義するものであり，この関数は値を受け取った仮引数の名前と値を対にして出力するものである．

例. 上記関数 f の評価（先の例の続き）

```
> f(1,2)   Enter          ←関数の評価
[1] "x:  1"            ←出力
[1] "y:  2"
```

次に仮引数を明に指定する関数呼び出しの例を示す．

例. 仮引数の名前を明に指定した関数評価（先の例の続き）

```
> f(x=1,y=2)   Enter          ←関数の評価
[1] "x:  1"            ←出力
[1] "y:  2"
```

先の例と同様の評価結果が得られていることがわかる．この形式による関数評価では，評価時の引数の記述の順番を変更することができる．(次の例)

例. 引数の記述の順番を変えて評価（先の例の続き）

```
> f(y=2,x=1)   Enter          ←関数の評価
[1] "x:  1"            ←出力
[1] "y:  2"
```

関数定義を記述した際の仮引数の順番とは異なる順番で引数を与えても，目的の仮引数に正しく値が渡されていることがわかる．

2.4.2 関数の内外で異なる変数の扱い（変数のスコープ）

関数定義の内部と外部では変数の扱いが異なる．先の例では，関数 tasu の内部で変数 r に値を割り当てていたが，それは関数定義の内部でのみ有効な**ローカル変数**[25]（local variable）であり，関数定義の外部の変数とは別のものとして扱われる．このことは次の例で確認できる．

例. 関数内部でのみ有効なローカル変数

```
> r <- 3.14   Enter      ←関数定義の外部の変数 r に値を与える
> tasu <- function( x, y ) {   Enter      ←関数 tasu の定義の開始
+      r <- x + y   Enter      ←関数定義の内部の変数 r（ローカル変数）に値を与える
+      print( sprintf("関数 tasu 内部の変数 r:  %f",r) )   Enter      ←ローカル変数 r の値の確認
+      r   Enter      ←関数 tasu の戻り値
+ }   Enter      ←関数 tasu の定義の終了
> tasu( 4, 5 )   Enter      ←関数 tasu の評価
[1] "関数 tasu 内部の変数 r:  9.000000"          ←関数 tasu のローカル変数 r の値は 9 である
[1] 9          ←関数 tasu の評価結果
> r   Enter      ←関数定義の外部の変数 r の値の確認
[1] 3.14          ←はじめに与えた値のままである
```

この例から，関数定義の外部の変数 r と関数定義内部のローカル変数 r が別のものであることがわかる．ただし，関数内部で関数外部の変数を参照することは可能である．（次の例）

例. 関数の内部から外部の変数を参照する

```
> r <- 3.14   Enter      ←関数定義の外部の変数 r に値を与える
> f1 <- function(n) {   Enter      ←関数 f1 の定義の開始
+      x <- r * n   Enter      ←関数外部の変数 r を参照している
+      x   Enter      ←関数 f1 の戻り値（変数 x はローカル変数）
+ }   Enter      ←関数 f1 の定義の終了
> f1(2)   Enter      ←関数 f1 の評価
[1] 6.28          ←関数 f1 の評価結果
```

この例では，関数 f1 内で外部の変数 r の値を参照している．すなわち，`x <- r * n` のように代入の右辺にあることでその値を参照するのみであり，この場合は変数 r は関数外部の変数として扱われる．

上の例のように関数定義の内部から外部の変数を参照することは，プログラムの可読性の面からも避けた方が良い．例えば次に示す例について考える．

例. あまり良くないプログラム

```
> r <- 3.14   Enter      ←関数定義の外部の変数 r に値を与える
> f2 <- function(n) {   Enter      ←関数 f2 の定義の開始
+      r <- r * n   Enter      ←この部分に注意！
+      r   Enter      ←関数 f2 の戻り値（変数 r はローカル変数）
+ }   Enter      ←関数 f2 の定義の終了
> f2(2)   Enter      ←関数 f2 の評価
[1] 6.28          ←関数 f2 の評価結果
> r   Enter      ←関数定義の外部の変数 r の値の確認
[1] 3.14          ←はじめに与えた値のままである
```

この例の関数 f2 の定義内部に `r <- r * n` という部分があるが，代入の右辺にある変数 r は関数定義の外部のものであり，左辺の変数 r は関数 f2 のローカル変数である．このような記述は誤解や混乱を招く恐れがあるので注意す

[25] **局所変数**と呼ぶこともある．

る必要がある.（あまり推奨されない記述方法である）関数の外部から関数定義の内部に値を与える場合は引数を介するべきである.

2.4.3　引数の暗黙値の設定

関数定義の記述において，引数に値が与えられなかった場合の**暗黙値**を設定することができる．これに関して例を挙げて説明する.

次の例で定義する関数 mag は，第 1 引数の値と第 2 引数の値の積を返すものである.

例. 2 つの引数の値の積を返す関数 mag

```
> mag <- function(x,m) {  Enter      ←関数 mag の定義の開始
+      x * m   Enter
+ }  Enter                  ←関数 mag の定義の終了
> mag(3,2)   Enter          ←関数 mag の評価
[1] 6                       ←定義通りの動作をしている
```

この関数 mag は 2 つの引数を取ることが前提となっており，第 2 引数が省略されると当然であるがエラーが発生する.

例. 上記関数の評価において第 2 引数を省略した場合（先の例の続き）

```
> mag(3)   Enter                                          ←評価を試みると…
  mag(3) でエラー:  引数 "m" がありませんし、省略時既定値もありません      ←エラーとなる
```

これに対して，関数 mag を次のような形で定義することができる.

例. 第 2 引数の暗黙値を設定された関数 mag の定義

```
> mag <- function(x,m=2) {  Enter      ←関数 mag の定義の開始
+      x * m   Enter
+ }  Enter                  ←関数 mag の定義の終了
> mag(3)   Enter            ←第 2 引数を省略して評価
[1] 6                       ←第 2 引数 m が 2 であるとして処理されている
```

この例のように，関数定義の記述において

仮引数=暗黙値

と記述することで，評価時に当該引数が省略された場合の値（**暗黙値**）を設定することができる．当然のことではあるが，その引数が明に与えられた場合はその値が取られる.

例. 上記関数 mag の評価（先の例の続き）

```
> mag(3,4)   Enter          ←第 2 引数を明に与えて評価
[1] 12                      ←評価結果
```

2.4.4　不定個数の引数を取る関数の定義

関数定義の記述の際，仮引数に 3 つのドット「...」を記述すると，不定の個数の引数を受け取ることができる．これに関して例を挙げて説明する.

次のような関数 f を考える.

例. 仮引数に「...」を記述する関数定義

```
> f <- function( x, y, ... ) { Enter      ←関数 f の定義の開始
+     s <- sprintf("第 1 引数:%d, 第 2 引数:%d",x,y) Enter
+     print(s) Enter
+     print("それ以降の引数:") Enter
+     for ( a in list(...) ) { Enter
+         print(a) Enter
+     } Enter
+ } Enter                    ←関数 f の定義の終了
```

この例で定義している関数 f には仮引数 x, y に続いて「...」が記述されている．このような記述により，関数の評価時に 2 個よりも多い個数の実引数が与えられた場合に 3 個目以降の引数が「...」の部分に受け取られる．そして，それらの引数の並びを

```
list(...)
```

とすることでリストの形にすることができる．

この関数 f を評価する例を示す．

例. 上記関数の評価（先の例の続き）

```
> f(1,2,3,4,5) Enter      ←関数 f にたくさんの引数を与えて評価
[1] "第 1 引数:1, 第 2 引数:2"          ←出力
[1] "それ以降の引数:"
[1] 3
[1] 4
[1] 5
```

例. 上記関数の評価（先の例の続き）

```
> f(1,2) Enter      ←関数 f に引数を 2 つだけ与えて評価
[1] "第 1 引数:1, 第 2 引数:2"          ←出力
[1] "それ以降の引数:"
```

関数定義の記述において，仮引数に「...」のみを記述することも可能であり，任意の個数の引数を受け取る関数を定義することができる．(次の例)

例. 関数 g(...) の定義

```
> g <- function( ... ) { Enter      ←任意の個数の引数を取る関数の定義
+     a <- list(...) Enter      ←引数の並びをリストに変換
+     print( sprintf("%d 個の引数が与えられました. ",length(a)) ) Enter
+     for ( x in a ) { Enter
+         print(x) Enter
+     } Enter
+ } Enter                    ←関数定義の終了
```

この関数 g を評価する例を示す．

例. 関数 g の評価（先の例の続き）

```
> g("x","y","z") Enter      ← 3 個の引数を与えて評価
[1] "3 個の引数が与えられました. "      ←出力
[1] "x"
[1] "y"
[1] "z"
```

例. 関数 g の評価（先の例の続き）

```
> g() Enter      ←引数無しで（0 個の引数を与えて）評価
[1] "0 個の引数が与えられました. "      ←出力
```

2.4.5　名前付きの引数を実現する方法

仮引数に暗黙値を設定する関数定義の記述

関数名 (仮引数 1=値 1, 仮引数 2=値 2, … , 仮引数 n=値 n)

において任意の仮引数の名前を扱えるようにする方法を示す．仮引数に「...」を記述して関数の内部で `list(...)` を実行すると，受け取った引数の並びをリストとして受け取ることができる．この際，得られた引数のリストの要素の名前を names 関数で参照する方法について考える．

例. 受け取った引数の名前を取り出す関数

```
> h1 <- function( ... ) { Enter      ←関数定義の開始
+     a <- list(...) Enter          ←受け取った引数をリスト a にする
+     print(a) Enter                ←それを表示して確認
+     print("引数の名前:") Enter
+     print( names(a) ) Enter       ←仮引数の名前のみ表示
+ } Enter      ←関数定義の終了
```

この例で定義している関数 h1 は任意の個数の引数を受け取るものであるが，受け取った引数をリストにして表示する．また，そのリストの要素の名前も文字列のベクトルとして出力する．この関数の実行例を次に示す．

例. 関数 h1 の評価（先の例の続き）

```
> h1(a=1,b=2,c=3) Enter      ←関数の評価
$a                 ←関数 h1 が受け取った引数をリストとして出力している
[1] 1
$b
[1] 2
$c
[1] 3
[1] "引数の名前:"
[1] "a" "b" "c"          ←引数のリストの名前のみ表示
```

これを応用すると，任意の名前を持つ引数を順不同で受け取る関数を実現する[26] ことができる．(次の例)

例. 任意の名前の引数を順不同で受け付ける関数 h2

```
> h2 <- function( ... ) { Enter      ←関数定義の開始
+     a <- list(...) Enter          ←受け取った引数をリスト a にする
+     nm <- names(a) Enter          ←引数の名前をベクトル nm にする
+     for (e in nm) { Enter          ←各引数名に対応する値を出力するループ
+         v <- a[[e]] Enter
+         print( paste(e,":",as.character(v)) ) Enter
+     } Enter
+ } Enter      ←関数定義の終了
```

この関数を評価する例を次に示す．

例. 関数 h2 の評価（先の例の続き）

```
> h2(a=1,b=2,c=3) Enter      ←関数の評価
[1] "a :  1"                ←受け取った引数を名前と値の対にして出力
[1] "b :  2"
[1] "c :  3"
```

引数の名前とその値が取得できていることがわかる．また，与える引数の順序を変えても引数の名前とその値の対応が取れている例を次に示す．

[26] **参考**）Python 言語における関数は**キーワード引数**を受け付けるが，それと同等のことを R で実現する手段となる．

例. 引数の順序を変えて関数 h2 を評価（先の例の続き）

```
> h2(c=3,b=2,a=1)   Enter      ←関数の評価
[1] "c :  3"                    ←受け取った引数を名前と値の対にして出力
[1] "b :  2"
[1] "a :  1"
```

引数の名前とその値の対応が先の例と同様になっていることがわかる.

2.5 オブジェクト指向プログラミング

R のオブジェクト指向プログラミング（以下 **OOP** と略すことがある）のための機能は段階的に発展してきており，OOP を支える機能のうち初期の（基本的な）ものとして **S3 オブジェクトシステム**（以下 S3 と略すことがある）がある．この S3 は R 以外の言語処理系が提供する OOP の仕組みとはかなり異なるものである．また，S3 とは別の **S4 オブジェクトシステム**（以下 S4 と略すことがある）も後に追加され，クラス定義の方法が R 以外の言語処理系のものに少し近づいた．S3 と S4 では OOP の記述の作法が異なるが，R 上で同時に使用することができる．S3，S4 以外にも OOP を実現するための機能[27] が後に実装されているが，S3 と S4 が R における基本的な OOP の機能である．

2.5.1 S3 オブジェクトシステム

要素に名前を持つリスト（list）の要素には「**リスト$名前**」と記述してアクセスすることができるが，S3 では基本的にこの機能を応用して OOP を実現している．また，S3 には**クラス定義**を明に宣言するための構文はなく，作成したリストなどのオブジェクトに **class 属性**を与えるという形で，そのオブジェクトが属するクラスを決定する．

2.5.1.1 関数定義のポリモーフィズムによるメソッドの実装

R の関数定義の機能には，処理対象のデータの型に基づく形の**多態性**[28]（ポリモーフィズム：polymorphism）がある．これを応用することで，特定のクラスのオブジェクトに対して有効となる**メソッド**を実装することができる．

処理対象のデータの型によって処理内容を異にする関数を定義するには次の 2 つの手順を踏む．

1. 関数を**総称関数**として定義する．
2. 型に応じた処理を定義する．

総称関数（generic function）とは，具体的な計算処理の定義を持つものではなく，その関数が多態性を持つ関数であることを宣言するものである．上記 1 は，具体的な関数定義に先立って 1 度だけ行う．上記 2 の関数定義は必要に応じて複数行う．以下に，処理対象のデータによって異なる加算処理を行う関数 tasu を定義して実行する例を示す．

例. 総称関数の定義

```
> tasu <- function(x,y) {   Enter      ←総称関数の定義の開始
+      UseMethod("tasu")    Enter      ←"tasu" を総称関数とする
+ }   Enter      ←総称関数の定義の終了
```

これで関数 "tasu" が総称関数として定義された．ただしこの段階では，関数 tasu には未だ処理内容は具体的には定義されていないので，これを実行しようとするとエラーが発生する．（次の例）

例. 総称関数の定義のみでは具体的な処理はできない（先の例の続き）

```
> tasu(2,3)   Enter      ←関数 tasu を実行しようとすると下のようなエラーが発生する
UseMethod("tasu") でエラー:
  'tasu' をクラス "c('double', 'numeric')" のオブジェクトに適用できるようなメソッドがありません
```

次に，引数に数値が与えられた場合の処理を実装して実行する例を示す．

例. 数値（numeric）に対する処理を実装する（先の例の続き）

```
> tasu.numeric <- function(x,y) {   Enter      ←数値に対する処理の定義の開始
+      x + y   Enter      ←具体的な処理（戻り値）の記述
+ }   Enter      ←関数の定義の終了
```

この例のように，「**関数名. 対象データの型**」に function の記述を与えて処理を実装する．この際の「対象データの型」は第 1 引数に与えるデータの型である．この例による関数定義の後，数値に対する tasu の処理が可能となる．（次の例）

[27] R5（RC，リファレンスクラス），R6 などがある．

[28] **多相性**ともいう．

例. 数値に対する関数 tasu の処理（先の例の続き）

```
> tasu(2,3)  Enter      ←関数 tasu の実行
[1] 5                   ←処理結果（戻り値）
```

数値に対する関数 tasu の戻り値が得られている．しかし，この段階では数値以外のデータ（例えば文字列など）に対する処理は定義されておらず，数値以外のデータを与えてこの関数を実行しようとするとエラーが発生する．(次の例)

例. 文字列に対する処理はまだ実装されていない（先の例の続き）

```
> tasu("abc","def")  Enter      ←関数 tasu に文字列を与えると下のようなエラーが発生する
UseMethod("tasu") でエラー:
  'tasu' をクラス "character" のオブジェクトに適用できるようなメソッドがありません
```

次に，引数に文字列（character）が与えられた場合の処理を実装して実行する例を示す．

例. 文字列（character）に対する処理を実装する（先の例の続き）

```
> tasu.character <- function(x,y) {  Enter      ←文字列に対する処理の定義の開始
+     paste(c(x,y),collapse="")  Enter      ←具体的な処理（単純な連結処理）の記述
+ }  Enter      ←関数の定義の終了
> tasu("abc","def")  Enter      ←文字列を与えて関数 tasu を実行
[1] "abcdef"                    ←処理結果が得られている
> tasu(2,3)  Enter      ←先に定義した数値に対する処理も
[1] 5                     もちろん可能
```

以上のように「**関数名. 型** `<- function` の記述 」の形式の関数定義により，オブジェクトのクラス毎に異なる処理を実行するメソッドを実装することができる．すなわち，関数をメソッドと見なして，第 1 引数に処理対象のオブジェクトを与えるという作法で「オブジェクトに対するメソッドの実行」を実現することができる．

2.5.1.2 オブジェクトのクラスの定義

R のオブジェクトのクラスは class 関数で調べることができるが，これは対象のオブジェクトの **class 属性**にアクセスする関数である．(次の例参照)

例. リストオブジェクトのクラスを調べる

```
> h0 <- list(name="John",weight=70,height=1.73)  Enter      ←リストオブジェクト h0 の作成
> class(h0)  Enter      ← h0 のクラスを調べる
[1] "list"          ←クラスが "list" であることがわかる
```

この class 属性は自由に設定することができ，S3 ではこれを応用して OOP を実現している．例えば，上の例では "name"，"weight"，"height" の 3 つの要素から成るリストオブジェクト h0 を作成しているが，このオブジェクトのクラスを "Human" に設定することで，"Human" クラスのオブジェクト h0 ができたことになる．(次の例参照)

例. class 属性の変更処理（先の例の続き）

```
> class(h0) <- "Human"  Enter      ← h0 のクラス属性を "Human" に変更
> class(h0)  Enter      ←クラスを確認
[1] "Human"          ←クラスが "Human" になっている
```

このように，R の S3 にはクラスの定義を明に記述するための構文は存在せず，作成したリストオブジェクトの class 属性を設定することでクラスの概念を実現している．また，先に説明した関数定義の多態性（ポリモーフィズム）を応用すると，特定のクラスに対して働くメソッドを実現することができる．次の例は，"Human" クラスのオブジェクトに対するメソッド bmi を実装するものである．

例. ”Human” クラスのオブジェクトに対する ”bmi” メソッドの実装（先の例の続き）

```
> bmi <- function(x) {  Enter      ← bmi を総称関数として定義
+     UseMethod("bmi")  Enter
+ }  Enter      ←総称関数の定義の終了
> bmi.Human <- function(x) {  Enter      ←”Human” クラスに対するメソッドとして bmi を定義
+     b <- x$weight / x$height ^2  Enter
+     b  Enter
+ }  Enter      ←メソッド定義の終了
```

この例で定義した bmi は「身長／体重 ^{2}」で定義される **ボディマス指数**を算出するものである．h0 に対して bmi を実行する例を次に示す．

例. h0 に対する bmi の実行（先の例の続き）

```
> bmi(h0)  Enter      ← h0 に対して bmi を実行
[1] 23.38869          ←計算結果（戻り値）
```

2.5.1.3 派生クラスの定義とメソッドの継承

OOP では，既存のクラス（**基底クラス**[29] ）の定義を継承した**派生クラス**[30] を新たに定義することができる．S3 では class 属性をベクトルの形で複数設定することでこれを実現する．すなわち

c(派生クラス, 基底クラス, 更にその基底クラス, …)

を class 属性に設定することで派生クラスを定義する．派生クラスは基底クラスに定義されたメソッドを自動的に継承する．次に示す例は，先に定義した ”Human” クラスの派生クラス ”Japanese” クラスを定義し，そのクラスのオブジェクトに対して ”Human” クラスのメソッドを実行するものである．

例. ”Human” クラスの派生クラス ”Japanese” の定義（先の例の続き）

```
> h1 <- list(name="Taro",weight=65,height=1.65)  Enter      ← list オブジェクト h1 の作成
> class(h1) <- c("Japanese","Human")  Enter      ←”Human”の派生クラス”Japanese”であるとの設定
> class(h1)  Enter      ←クラス設定の確認
[1] "Japanese" "Human"      ←クラス設定の状態
> bmi(h1)  Enter          ←基底クラス ”Human” のメソッド bmi が”Japanese” クラスの
[1] 23.87511                オブジェクトに対して実行可能である
```

もちろん bmi メソッドを ”Japanese” クラス用に定義することも可能である．この場合，派生クラス用のメソッドの中から NextMethod を使用して基底クラスの同名のメソッドを呼び出すことができる．(次の例)

例. bmi メソッドを ”Japanese” クラス用に定義する（先の例の続き）

```
> bmi.Japanese <- function(x) {  Enter      ←メソッド定義の開始
+     print(paste(c("日本人",x$name,"の BMI"),collapse=""))  Enter
+     NextMethod("bmi")  Enter          ←基底クラスの同名のメソッドを呼び出す
+ }  Enter      ←メソッド定義の終了
> bmi(h1)  Enter          ←”Japanese” クラス用の bmi メソッドを呼び出す
[1] "日本人 Taro の BMI"          ←処理結果
[1] 23.87511
```

■ 深い継承関係のあるクラスの定義

先に挙げた例では 2 つのクラス間（”Human”→”Japanese”の関係）のメソッドの継承を示したが，次に 3 つのクラス ”Human”→”Japanese”→”Tokyoite” の関係において，”Human” のメソッドを ”Tokyoite” のクラスに継承させることについて考える．

R 以外の言語処理系の OOP においては，”Tokyoite”クラスが”Japanese”クラスの派生クラスであるとの継承関係

[29] **スーパークラス**，**親クラス**，**上位クラス**などと呼ばれることもある．

[30] **サブクラス**，**子クラス**，**拡張クラス**，**下位クラス**などと呼ばれることもある．

（2 つのクラスの間の関係）を定義するだけで自動的に ”Human” クラスのメソッドが ”Tokyoite” クラスにも継承されるが，R の S3 では事情が異なる．このことを例を挙げて説明する．

次の例は，”Japanese” クラスの派生クラスである ”Tokyoite” クラスのオブジェクト h2 を作成し，”Tokyoite” クラスのためのメソッド bmi を定義するものである．

例. ”Japanese”の派生クラス”Tokyoite”のオブジェクト h2 とそれに対する bmi メソッド（先の例の続き）

```
> h2 <- list(name="Yuriko",weight=60,height=1.6)  Enter
> class(h2) <- c("Tokyoite","Japanese")  Enter    ←”Japanese”を継承する”Tokyoite”クラス
> bmi.Tokyoite <- function(x) {  Enter
+     print(paste(c("東京人",x$name,"の BMI"),collapse=""))  Enter
+     NextMethod("bmi")  Enter
+ }  Enter
```

”Tokyoite” クラスは ”Japanese” のメソッドを継承しているが，”Human”を継承する記述はないので，bmi を h2 に対して実行すると次のようにエラーとなる．

例. h2 に対して bmi を実行する試み（先の例の続き）

```
> bmi(h2)  Enter    ←実行を試みると…
[1] "東京人 Yuriko の BMI"        ←”Tokyoite”に対する処理
[1] "日本人 Yuriko の BMI"        ←”Japanese”の継承まではできているが
NextMethod("bmi") でエラー:  起動すべきメソッドはありません      ←”Human”は継承されていない
```

従って，”Human”→”Japanese”→”Tokyoite” と全ての派生関係を反映させるには h2 の class 属性の記述にそれら全てのクラスを含める必要がある．（次の例）

例. h2 の class 属性の変更（先の例の続き）

```
> class(h2) <- c("Tokyoite","Japanese","Human")  Enter    ← class 属性の再設定
> bmi(h2)  Enter    ←再度実行を試みる
[1] "東京人 Yuriko の BMI"        ←”Tokyoite”に対する処理
[1] "日本人 Yuriko の BMI"        ←”Japanese”に対する処理
[1] 23.4375                  ←”Human”に対する処理
```

2.5.2 S4 オブジェクトシステム

S4 オブジェクトシステムは OOP を実現するものであるが S3 とは異なるものである．S3 はリストを基本とするオブジェクトのシステムであるが，S4 ではリストとは異なる独特のオブジェクト（S4 オブジェクト）を用いる．また，S4 ではクラスの構造を明に定義した後，その定義に基づいた形のオブジェクトを生成するという手順[31] を踏む．

2.5.2.1 クラスの定義とインスタンスの生成

ここでは，S3 の解説で示したものに近い例を挙げて S4 について解説する．”name”，”weight”，”height” の 3 つの値を保持するクラス ”Human” を S4 の形式で宣言するには setClass を用いて次のように記述する．

例. S4 における ”Human” クラスの定義

```
> setClass("Human",  Enter    ←クラス宣言の開始
+     representation(  Enter    ←スロットの定義の開始
+         name = "character",  Enter    ←”name” の型は文字列である
+         weight = "numeric",  Enter    ←”weight” の型は数値である
+         height = "numeric"  Enter    ←”height” の型は数値である
+     )  Enter                ←スロットの定義の終了
+ )  Enter                    ←クラス宣言の終了
```

S3 の場合はオブジェクトは基本的にはリストなので，オブジェクト内で値を保持するものはそのリストの要素であっ

[31] R 以外の言語処理系の OOP では一般的な手順である．

た．それに対して S4 では，オブジェクト内部で値を保持するものは**スロット**と呼ばれる．上の例で示したように，setClass でクラス定義を行い，その内部に representation の記述によってスロットを定義する．また，スロットの定義の際には各スロットが保持するデータの型を指定する．

setClass で定義されたクラスの**インスタンス**[32]（S4 オブジェクト）を生成するには new を使用する．

書き方：　new(クラス名, スロット 1=値 1, スロット 2=値 2, …)

例. ”Human” クラスのインスタンス h0 を作成する（先の例の続き）

```
> h0 <- new("Human",name="John",weight=70,height=1.73)  Enter     ←インスタンスの生成
> h0  Enter     ←内容確認
An object of class "Human"
Slot "name":
[1] "John"
Slot "weight":
[1] 70
Slot "height":
[1] 1.73
```

S4 ではスロットの構成と型の定義が厳格であり，setClass で記述した定義に従わない形で new を実行するとエラーとなる．

例. 不正なデータ型の値を与える例（先の例の続き）

```
> h02 <- new("Human",name="John",weight="70",height=1.73)  Enter     ←不正な weight の値
validObject(.Object) でエラー:                                    ←エラーとなる．
   不正なクラス "Human" オブジェクト :  invalid object for slot "weight" in class "Human":
  got class "character", should be or extend class "numeric"
```

例. 存在しないスロット lang に値を与える例（先の例の続き）

```
> h02 <- new("Human",name="John",weight=70,height=1.73,lang="日本語")  Enter
initialize(value, ...)  でエラー:                                  ←エラーとなる．
   クラス "Human" のスロットに対しては不正な名前です:  lang
```

S4 オブジェクトのスロットにアクセス（値の参照や設定）するには「@」を用いて

S4 オブジェクト@スロット名

と記述する．（次の例）

例. S4 オブジェクトのスロットへのアクセス（先の例の続き）

```
> h0@name  Enter     ← h0 のスロット name の値の参照
[1] "John"            ←値
> h0@name <- "Joe"  Enter     ← h0 のスロット name に ”Joe” を設定
> h0@name  Enter     ←値の確認
[1] "Joe"             ←”Joe” が設定されている
```

これと同様のことを slot 関数を用いて行うこともできる．

書き方：　slot(S4 オブジェクト, スロット名)

「スロット名」は文字列形式で与える．

[32] クラス定義を反映した個別のオブジェクトのこと．

例. slot 関数によるスロットへのアクセス（先の例の続き）

```
> slot(h0,"name")  Enter      ← h0 のスロット name の値の参照
[1] "Joe"                ←値
> slot(h0,"name") <- "John"  Enter      ← h0 のスロット name に "John" を設定
> slot(h0,"name")  Enter      ←値の確認
[1] "John"               ←"John" が設定されている
```

2.5.2.2 メソッドの定義

S4 オブジェクトに対する**メソッド**の定義の手順は S3 のそれと似ており，**総称関数**を定義した後，対象クラスのための関数を定義するという手順を踏む．例えば，S4 の総称関数 bmi を定義するには setGeneric を用いて次のようにする．

例. 総称関数 bmi の定義（先の例の続き）

```
> setGeneric("bmi",  Enter      ←総称関数の定義の開始
+     function(x) {  Enter
+         standardGeneric("bmi")  Enter      ←総称関数であることを設定している
+     },  Enter
+     valueClass = "numeric"  Enter      ←戻り値の型の宣言
+ )  Enter      ←総称関数の定義の終了
[1] "bmi"      ←総称関数 bmi が定義された
```

このように，setGeneric の内部で function を記述し，更にその内部で standardGeneric を記述して総称関数であることを設定する．また setGeneric の内部には valueClass を記述して，当該関数の戻り値の型を指定する．（ベクトルの要素として複数指定可能）

上の例の処理で bmi が総称関数であることが宣言できたが，この段階ではまだ具体的な処理は実装されていない．（次の例）

例. 総称関数を定義しただけでは具体的な処理はできない（先の例の続き）

```
> bmi(h0)  Enter      ←実行を試みると…
(function (classes, fdef, mtable) でエラー:              ←エラーとなる
  関数 'bmi' (シグネチャ '"Human"' に対する) に対する継承メソッドを見付けることができません
```

特定のクラスのオブジェクトに対する具体的な処理は setMethod を使用して定義する．（次の例）

例. "Human" クラスに対する bmi メソッドの実装と実行（先の例の続き）

```
> setMethod("bmi",signature("Human"),  Enter      ←"Human"クラスに対するメソッド実装の開始
+     function(x) {  Enter      ←具体的な処理を表す function の記述の開始
+         x@weight / x@height ^ 2  Enter
+     }  Enter              ← function の記述の終了
+ )  Enter      ←メソッド実装の記述の終了
> bmi(h0)  Enter      ← h0 に対して bmi メソッドを実行
[1] 23.38869      ←実行結果（戻り値）
```

setMethod の第 1 引数にメソッド名を文字列の形式で与え，第 2 引数に対象オブジェクトのクラス名を signature(クラス名) の形式（クラス名は文字列の形式）で与える．そして第 3 引数に具体的な処理を表す function の記述を与える．この場合，function の第 1 番目の仮引数は処理対象のオブジェクト（signature に記述したクラスのオブジェクト）を受け取るものである．

2.5.2.3 派生クラスの定義とメソッドの継承

"Human" クラスの派生クラスである "Japanese" クラスの定義を例に挙げて派生クラスの定義とメソッドの継承について解説する．

例. ”Human” クラスを継承する ”Japanese” クラスの定義（先の例の続き）

```
> setClass("Japanese",  Enter     ←クラス宣言の開始
+     representation(  Enter     ←スロットの定義の開始
+         lang = "character"  Enter     ←このクラス固有の ”lang” スロット
+     ),  Enter                  ←スロットの定義の終了
+     prototype(  Enter          ←スロットの初期値の記述の開始
+         lang = "日本語"  Enter       ←スロット ”lang” は初期値 ”日本語” を持つ
+     ),  Enter                  ←スロットの初期値の記述の終了
+     contains = "Human"  Enter       ←このクラスは ”Human” の派生クラスである
+ )  Enter                       ←クラス宣言の終了
```

先に ”Human” クラスを定義した際の setClass と基本的には同じであるが，contains の記述により，この ”Japanese” クラスが ”Human” クラスの派生クラスとして定義される．またこの定義の中の representation の記述で lang というスロットが設置されているが，これは ”Human” クラスの全てのスロットを継承した上で追加される．このクラス定義に含まれる prototype の記述は，当該クラスのインスタンスを new によって生成した際の初期値の設定である．prototype は派生クラスのみならず，基底クラスの定義の際にも記述できる．

”Japanese” クラスには ”Human” クラスのメソッドも自動的に継承される．”Japanese” クラスのインスタンスを生成して bmi メソッドを実行する例を次に示す．

例. ”Japanese” クラスのインスタンスを生成して bmi メソッドを実行する（先の例の続き）

```
> h1 <- new("Japanese",name="Taro",weight=65,height=1.65)  Enter
> bmi(h1)  Enter     ←基底クラス ”Human” に定義されたメソッド bmi を実行する
[1] 23.87511     ←実行結果（戻り値）
```

もちろん ”Japanese” クラスに対する bmi メソッドを改めて定義することも可能である．（次の例）

例. ”Japanese” クラスに対する bmi メソッドの実装と実行（先の例の続き）

```
> setMethod("bmi",signature("Japanese"),  Enter  ←”Japanese”クラスに対するメソッド実装の開始
+     function(x) {  Enter
+         print(paste(c("日本人",x@name,"の BMI"),collapse=""))  Enter
+         callNextMethod(x)  Enter          ←基底クラスに対する ”bmi” を呼び出す
+     }  Enter
+ )  Enter     ←メソッド実装の記述の終了
> bmi(h1)  Enter     ← h1 に対して bmi メソッドを実行
[1] "日本人 Taro の BMI"       ←”Japanese” クラスの bmi メソッドによる出力
[1] 23.87511       ←基底クラス ”Human” の bmi メソッドによる出力
```

この例のように，callNextMethod で基底クラスの同名のメソッドを実行することができる．

S3 では ”Human”→”Japanese”→”Tokyoite” の継承において，これら 3 つのクラス全てを ”Tokyoite” クラスのオブジェクトの class 属性に与える必要があることを先に示した．S4 では 2 つのクラス間（基底クラスと派生クラスの 2 者間）の継承関係を setClass で記述するだけで，”Human”→”Japanese”→”Tokyoite” のような深い継承関係においてもメソッドは自動的に継承される．（次の例）

例. ”Japanese” クラスを継承する ”Tokyoite” クラスの定義（先の例の続き）

```
> setClass("Tokyoite",  Enter     ←クラス宣言の開始
+     contains = "Japanese"  Enter       ←このクラスは ”Japanese” の派生クラスである
+ )  Enter                       ←クラス宣言の終了
```

この ”Tokyoite” クラスのインスタンス h2 を生成して bmi メソッドを実行する例を次に示す．

例. ”Tokyoite” クラスのインスタンスを生成して bmi メソッドを実行する（先の例の続き）

```
> h2 <- new("Tokyoite",name="Yuriko",weight=60,height=1.6)  Enter
> bmi(h2)  Enter      ←上位のクラスに定義されたメソッド bmi を実行する
[1] "日本人 Yuriko の BMI"        ←基底クラス ”Japanese” クラスの bmi メソッドによる出力
[1] 23.4375        ←更に上位の ”Human” クラスの bmi メソッドによる出力
```

■ S4 のまとめ

《クラスの定義》

書き方： **setClass(クラス名,**
representation(スロット 1=型 1, スロット 2=型 2, …),
prototype(スロット 1=値 1, スロット 2=値 2, …),
contains = 基底クラス
)

クラス名は文字列形式で，representation 中の型も文字列形式で記述する．

《インスタンスの生成とスロットへのアクセス》

書き方： **new(クラス名, スロット 1=値 1, スロット 2=値 2, …)**

文字列形式で指定したクラス名のクラスのオブジェクトを生成する．この際にスロットの初期値を与えることができる．オブジェクトのスロットにアクセスするには「@」を用いて「**オブジェクト名@スロット名**」と記述する．また,「**slot(オブジェクト名, スロット名)**」と記述してもスロットにアクセスすることができる．この場合は文字列形式でスロット名を記述する．

《メソッドの定義》

総称関数： **setGeneric(メソッド名,**
function(仮引数) { standardGeneric(メソッド名) },
valueClass = 戻り値の型
)

メソッド名は文字列形式で記述する．戻り値の型も文字列形式で記述する．個別のクラスのオブジェクトに対する具体的な処理の実装は次のような形式で記述する．

書き方： **setMethod(メソッド名, signature(クラス名),**
function(仮引数) { 具体的な処理 }
)

この記述で signature に指定したクラスのオブジェクトに対するメソッドを実装する．この場合のクラス名は文字列の形式で記述する．function の第 1 番目の仮引数には signature に記述したクラスのオブジェクトを受け取る．基底クラスの同名のメソッドを呼び出すには **callNextMethod** を呼び出す．

2.6　日付，時刻の扱い

日付，時刻は**タイムスタンプ**を扱うための基本的な情報であり，R には「ある時点の日付，時刻」を 1 つの値として取り扱うためのクラスやメソッドなどが提供されている．

2.6.1　基本的なクラス

日付の情報を取り扱うためのクラスとして Date クラスがある．このクラスのオブジェクトは年月日の情報を保持するもので，オブジェクト内部に 1970 年 1 月 1 日からの経過日数を保持している．文字列として記述された日付を Date オブジェクトに変換するには

as.Date(日付の記述)

を実行する．また，使用している計算機環境の現在の日付（ローカルタイム）を取得するには

Sys.Date()

を実行する．

例. 1970 年 1 月 1 日の日付を持つ Date オブジェクトの作成

```
> d0 <- as.Date("1970-01-01")  Enter     ← Date オブジェクトの作成
> d0   Enter                  ←内容確認
[1] "1970-01-01"              ←得られた Date オブジェクトの日付
> class(d0)   Enter           ←クラスを調べる
[1] "Date"                    ← Date クラスである
```

日付が Date オブジェクト d0 として得られている．

例. 現在の日付の取得（先の例の続き）

```
> d1 <- Sys.Date()   Enter     ←現在時刻の取得
> d1   Enter                  ←内容確認
[1] "2022-08-07"              ←得られた Date オブジェクトの日付（実行時）
```

Date クラスのオブジェクトの値は差を求めることができる．すなわち 2 つの日付の間の差（日数）を求めることができる．(次の例)

例. 上の例で取得した Date オブジェクトの差（先の例の続き）

```
> dd <- d1 - d0   Enter     ← Date オブジェクトの差を dd に取得
> dd   Enter                ←内容確認
Time difference of 19211 days       ←差（日数）が得られている
> class(dd)   Enter         ← dd のクラスを調べる
[1] "difftime"                ← difftime クラスである
```

Date オブジェクトの差は difftime クラスのオブジェクトである．上の例では 19,211 日の差が dd に得られている．difftime オブジェクトは Date オブジェクトに加算することが可能で，その場合は difftime オブジェクトが意味する日数だけ経過した日付が得られる．(次の例)

例. Date オブジェクト d1 に difftime オブジェクト dd を加算する（先の例の続き）

```
> d2 <- d1 + dd   Enter     ← Date オブジェクト d1 に dd を加算
> d2   Enter                ←内容確認
[1] "2075-03-13"              ←得られた Date オブジェクトの日付
```

時間軸上の 1 点（**タイムスタンプ**）を正確に表現するには日付と時刻の両方を表現する必要があり，そのためのクラスとして POSIXct と POSIXlt がある[33] .

使用している計算機環境の現在の日付と時刻（ローカルタイム）を POSIXct オブジェクトとして取得するには

[33] タイムスタンプの表現には POSIXct クラスがより基本的である．

Sys.time()

を実行する．

例. 現在の日付と時刻の取得

```
> t1 <- Sys.time()  Enter      ←現在の日付と時刻を取得
> t1  Enter                    ←内容確認
[1] "2022-08-07 18:39:16 JST"       ←得られた POSIXct オブジェクトの内容
> class(t1)  Enter             ←クラスを調べる
[1] "POSIXct" "POSIXt"          ← POSIXct クラスである
```

この実行例からわかるように，POSIXct クラスは POSIXt クラスの派生クラスである．

文字列として記述された日付，時刻を POSIXct オブジェクトに変換するには

as.POSIXct(日付時刻の記述)

を実行する．例えば上記の実行例と同じ日付，時刻の文字列表現を POSIXct オブジェクトに変換するには次のように実行する．

例. 日付，時刻の文字列を POSIXct に変換する

```
> t1 <- as.POSIXct("2022-08-07 18:39:16")  Enter
> t1  Enter                    ←内容確認
[1] "2022-08-07 18:39:16 JST"
```

POSIXct クラスのオブジェクトの値は差を求めることができる．すなわち 2 つのタイムスタンプの間の時間差を求めることができる．(次の例)

例. タイムスタンプの差を求める（先の例の続き）

```
> t0 <- as.POSIXct("1970-01-01 00:00:00")  Enter     ←別のタイムスタンプ
> tt <- t1 - t0  Enter     ←タイムスタンプ t1 と t0 の差を求める
> tt  Enter                ←内容確認
Time difference of 19211.78 days       ←時間の差が得られている
> class(tt)  Enter         ← tt のクラスを調べる
[1] "difftime"             ← difftime クラスである
```

POSIXct オブジェクトの差も difftime クラスのオブジェクトである．difftime オブジェクトは POSIXct オブジェクトに加算することが可能で，その場合は difftime オブジェクトが意味する時間だけ経過した後のタイムスタンプが得られる．(次の例)

例. POSIXct オブジェクト t1 に difftime オブジェクト tt を加算する（先の例の続き）

```
> t2 <- t1 + tt  Enter     ← t1 に tt を加算
> t2  Enter                ←内容確認
[1] "2075-03-14 13:18:32 JST"          ←得られたタイムスタンプ
```

2.6.1.1 difftime クラス

difftime クラスのオブジェクトは時間の差（時間の長さ，間隔）を表現するためのもので，単独で作成することができる．(次の例)

例. difftime オブジェクトの作成（時，分，秒）

```
> as.difftime(3,units="secs")  Enter   ← 3 秒間
Time difference of 3 secs
> as.difftime(3,units="mins")  Enter   ← 3 分間
Time difference of 3 mins
> as.difftime(3,units="hours")  Enter  ← 3 時間
Time difference of 3 hours
```

例. difftime オブジェクトの作成（日，週）

```
> as.difftime(3,units="days")  Enter   ← 3 日間
Time difference of 3 days
> as.difftime(3,units="weeks")  Enter   ← 3 週間
Time difference of 3 weeks
```

2.6.2 タイムゾーン

世界の地域には時差があり，タイムスタンプには日付と時刻の他に，その地域を意味する**タイムゾーン**の情報が含まれる．as.POSIXct でタイムスタンプを作成する際に，引数「tz=タイムゾーン」を与えることでタイムゾーンの情報を付加することができる．先に示した例では，時刻の表示に ”**JST** ” という表記が見られたが，これは日本のローカルタイムを意味する．ただし as.POSIXct で日本のローカルタイムを作成する場合は引数に「tz="Asia/Tokyo"」を与える．日本のローカルタイムを意味する ”Asia/Tokyo” は IANA [34] が管理する tz database におけるタイムゾーンの名前[35] である．タイムゾーンを指定してタイムスタンプを作成する例を次に示す．

例. 協定世界時（UTC）と日本のローカルタイムの違い

```
> tj <- as.POSIXct("1970-01-01 00:00:00",tz="Asia/Tokyo")  Enter    ←日本時間
> tu <- as.POSIXct("1970-01-01 00:00:00",tz="UTC")  Enter    ←協定世界時
> tu - tj  Enter    ←時差を調べる
Time difference of 9 hours    ← 9 時間の時差がある
```

この例からもわかるように，POSIXct オブジェクトは地域の情報を含んでおり，同じ「1970 年 1 月 1 日 0 時 0 分 0 秒」でも日本時間と協定世界時では 9 時間の時差がある．

2.6.3 日付，時刻の書式整形

format 関数を使用すると POSIXct オブジェクトの書式を整形した文字列を取得する[36] ことができる．(次の例)

例. POSIXct オブジェクトの書式整形

```
> t3 <- as.POSIXct("2022-10-01 13:15:21",tz="Asia/Tokyo")  Enter    ←日付時刻の作成
> s <- format(t3,"%Y 年%m 月%d 日%H 時%M 分%S 秒 (%Z)")  Enter    ←書式整形
> s  Enter                                          ←内容確認
[1] "2022 年 10 月 01 日 13 時 15 分 21 秒 (JST)"          ←結果
```

書式を指定する記号については代表的なものを表 11 に挙げる．

表 11: 日付，時刻に関する書式の指定（一部）

書式指定	解　　説	書式指定	解　　説
%Y	西暦年（4 桁）	%y	西暦年（下 2 桁）
%m	月	%d	日
%H	時（24 時間制）	%l	時（12 時間制）
%M	分	%S	秒
%p	”午前” or ”午後”　*	%Z	タイムゾーン
%z	UTC との時差	%w	曜日番号（日が 0, 土が 6）
%a	曜日（省略形）　*	%A	曜日　*

*ローカルの表現（日本の場合は日本語）

Date クラスのオブジェクトに対しても同様に format による書式整形が可能である．(次の例)

[34] **I**nternet **A**ssigned **N**umbers **A**uthority（https://www.iana.org/）

[35] List of tz database time zones（https://en.wikipedia.org/wiki/List_of_tz_database_time_zones）

[36] 考え方は「2.3.1 sprintf 関数による書式整形」（p.44）で解説した sprintf に似ている．

例. Date オブジェクトの書式整形

```
> d1 <- as.Date("2022-10-01")  Enter    ←日付の作成
> format(d1,"%Y 年%m 月%d 日")  Enter    ←書式整形
[1] "2022 年 10 月 01 日"          ←結果
```

2.6.4 日付，時刻から部分を取り出す方法

POSIXct オブジェクトを POSIXlt に変換すると日付，時刻の各部の情報を取り出すことができる．変換処理には as.POSIXlt を用いる．(次の例)

例. POSIXct オブジェクトを POSIXlt オブジェクトに変換する

```
> t3 <- as.POSIXct("2022-10-01 13:15:21",tz="Asia/Tokyo") Enter ← POSIXct オブジェクトの作成
> t32 <- as.POSIXlt(t3)  Enter    ←変換処理（t3 → t32）
> t32  Enter                       ← t32 の内容確認
[1] "2022-10-01 13:15:21 JST"        ←結果
> class(t32)  Enter                  ← t32 のクラスを確認
[1] "POSIXlt" "POSIXt"              ←結果
```

この例からわかるように，POSIXlt クラスは POSIXt の派生クラスである．POSIXlt オブジェクトはリストを応用したものであり，内部の各要素の名前を指定して値を参照することができる．(次の例)

例. POSIXlt オブジェクトから時, 分, 秒を取り出す（先の例の続き）

```
> t32$hour  Enter
[1] 13      ←「時」
> t32$min  Enter
[1] 15      ←「分」
> t32$sec  Enter
[1] 21      ←「秒」
```

POSIXlt オブジェクトの要素の名前の代表的なものを表 12 に挙げる．

表 12: POSIXlt オブジェクトの要素の名前（一部）

名前	解　　説	名前	解　　説
year	年（1900 年からの差）	mon	月番号（0〜11）
mday	日	wday	曜日番号（日が 0, 土が 6）
hour	時	min	分
sec	秒	zone	タイムゾーン
gmtoff	世界標準時からの差（秒）		

2.6.5 日付，時刻の系列を作成する方法

関数 seq は Date や POSIXct のオブジェクトの系列をベクトルの形で生成することができる．

例. Date オブジェクトの系列作成

```
> d1 <- as.Date("2022-01-01")  Enter    ← 2022/01/01
> d2 <- as.Date("2023-01-01")  Enter    ← 2023/01/01（1 年後）
> ds <- seq(d1,d2,by="month")  Enter    ← 1 月間隔の日付の系列を作成
> ds  Enter                          ←内容確認
 [1] "2022-01-01" "2022-02-01" "2022-03-01" "2022-04-01" "2022-05-01"
 [6] "2022-06-01" "2022-07-01" "2022-08-01" "2022-09-01" "2022-10-01"
[11] "2022-11-01" "2022-12-01" "2023-01-01"
```

例. POSIXct オブジェクトの系列作成

```
> t3 <- as.POSIXct("2022-10-01 13:15:21",tz="Asia/Tokyo")  Enter   ←タイムスタンプ 1
> t4 <- as.POSIXct("2023-03-31 23:59:59",tz="Asia/Tokyo")  Enter   ←タイムスタンプ 2
> ts <- seq(t3,t4,by="month")  Enter   ← 1 月間隔のタイムスタンプの系列を作成
> ts  Enter                          ←内容確認
 [1] "2022-10-01 13:15:21 JST" "2022-11-01 13:15:21 JST"
 [3] "2022-12-01 13:15:21 JST" "2023-01-01 13:15:21 JST"
 [5] "2023-02-01 13:15:21 JST" "2023-03-01 13:15:21 JST"
```

seq の引数「by=」には

```
"sec", "min", "hour", "day", "DSTday", "week", "month", "quarter", "year"
```

といったものを与えることができる．またこれ以外にも，時間の差を表す difftime オブジェクトを与えることもでき，時間間隔をより具体的に指定することができる．(次の例)

例. 6 時間毎のタイムスタンプの系列

```
> tt <- as.difftime(6,units="hours")  Enter   ←「6 時間」の長さ
> t3 <- as.POSIXct("2022-10-01 13:15:21",tz="Asia/Tokyo")  Enter   ←タイムスタンプ 1
> t4 <- as.POSIXct("2022-10-02 23:59:59",tz="Asia/Tokyo")  Enter   ←タイムスタンプ 2
> ts <- seq(t3,t4,by=tt)  Enter        ←系列を作成
> ts  Enter                          ←内容確認
 [1] "2022-10-01 13:15:21 JST" "2022-10-01 19:15:21 JST"
 [3] "2022-10-02 01:15:21 JST" "2022-10-02 07:15:21 JST"
 [5] "2022-10-02 13:15:21 JST" "2022-10-02 19:15:21 JST"
```

3　統計処理のための基本的な機能

3.1　乱数データの作成

各種のサンプルデータを作成する際に乱数が応用される．特に**一様乱数**はよく用いられるものであり，関数 runif を用いて生成することができる．

書き方：　runif(個数, min=下限, max=上限)

「下限」〜「上限」の範囲の乱数を「個数」に指定した長さのベクトルの要素として作成する．下限を省略した際のデフォルトは 0，上限を省略した際のデフォルトは 1.0 である．

例. -1〜1 の範囲の乱数を 10,000 個作成する

```
> r <- runif(10000,min=-1,max=1)  Enter      ←乱数を r に作成
> r[1:12]  Enter      ←先頭 12 個を表示
 [1] -0.66391695 0.61503280 -0.23011530 -0.34453137 0.20420135 0.20878811
 [7] -0.75073311 -0.41079815 0.15521984 0.26195855 0.02403180 0.01004783
```

一様乱数は偏りのない値を生成する．このことを次の例の処理で確認する．

例. 一様乱数のヒストグラム（先の例の続き）

```
> hist(r,breaks=10,main="")  Enter      ←乱数を r のヒストグラムを作成
```

これはヒストグラムを作図する機能である．この処理によって図 6 のようなグラフが表示され，データにあまり偏りがない様子がわかる．

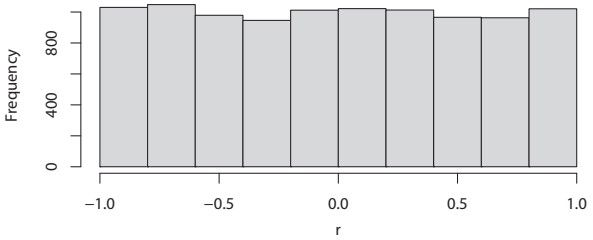


図 6: 一様乱数の散らばり方

各種グラフの作成に関しては後の「4 データの可視化」（p.85）で詳しく解説する．

一様乱数と並んでよく用いられる乱数に**正規乱数**があり，関数 rnorm で生成することができる．

書き方：　rnorm(個数, mean=平均, sd=標準偏差)

与えられた「平均」と「標準偏差」に従う正規乱数を「個数」に指定した長さのベクトルの要素として作成する．平均を省略した際のデフォルトは 0.0，標準偏差を省略した際のデフォルトは 1.0 である．

例. $\mu = 0, \sigma = 1$ の正規乱数を 10,000 個作成する

```
> r <- rnorm(10000,mean=0,sd=1)  Enter      ←乱数を r に作成
> r[1:12]  Enter      ←先頭 12 個を表示
 [1] -0.96193342 -0.29252572 0.25878822 -1.15213189 0.19578283 0.03012394
 [7] 0.08541773 1.11661021 -1.21885742 1.26736872 -0.74478160 -1.13121857
```

正規乱数は平均の周辺に多く偏った値を生成する．このことを次の例の処理で確認する．

例. 正規乱数のヒストグラム（先の例の続き）

```
> hist(r,breaks=20,main="")  Enter      ←乱数を r のヒストグラムを作成
```

この処理によって図 7 のようなグラフが表示される．

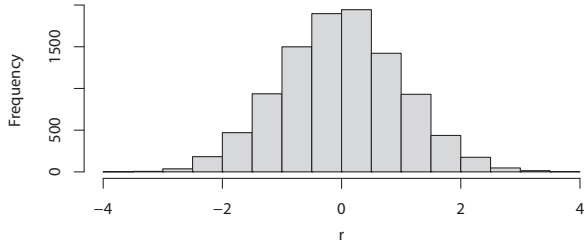


図 7: 正規乱数のヒストグラム

3.1.1 乱数の seed

乱数は生成する度に再現性の無い数値が得られる.（次の例）

例. 再現性の無い乱数生成

```
> runif(6)  Enter      ← 1 回目の乱数生成
[1] 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819 0.8983897
> runif(6)  Enter      ← 2 回目の乱数生成
[1] 0.94467527 0.66079779 0.62911404 0.06178627 0.20597457 0.17655675
> runif(6)  Enter      ← 3 回目の乱数生成
[1] 0.6870228 0.3841037 0.7698414 0.4976992 0.7176185 0.9919061
```

これは当然のことであるが，乱数をサンプルデータ作成に応用する際などには，生成する乱数に再現性が求められる場合がある．乱数生成[37] には seed（**乱数の種**）という概念があり，同一の seed からは同じ系列の乱数が生成される．この seed の設定には set.seed を使用する．

書き方： set.seed(種)

乱数生成に先立って「種」を設定すると，以後の乱数生成の系列は確定的なものとなる.（次の例）

例. seed 設定後の乱数生成

```
> set.seed(3); runif(6)  Enter      ← 1 回目の乱数生成
[1] 0.1680415 0.8075164 0.3849424 0.3277343 0.6021007 0.6043941
> set.seed(3); runif(6)  Enter      ← 2 回目の乱数生成
[1] 0.1680415 0.8075164 0.3849424 0.3277343 0.6021007 0.6043941
> set.seed(3); runif(6)  Enter      ← 3 回目の乱数生成
[1] 0.1680415 0.8075164 0.3849424 0.3277343 0.6021007 0.6043941
```

異なる seed を設定すると生成する乱数の系列は異なったものになるが，同一の seed から生成される乱数系列のパターンは同じものとなる.（次の例）

例. 先の例とは異なる seed の設定

```
> set.seed(5); runif(6)  Enter      ← 1 回目の乱数生成
[1] 0.2002145 0.6852186 0.9168758 0.2843995 0.1046501 0.7010575
> set.seed(5); runif(6)  Enter      ← 2 回目の乱数生成
[1] 0.2002145 0.6852186 0.9168758 0.2843995 0.1046501 0.7010575
```

3.2 ランダムサンプリング

関数 sample を使用するとデータセットの中からランダムにデータを取り出すことができる.

書き方： sample(データセット, 個数)

「データセット」の中から指定した「個数」の要素を取り出す.「個数」を省略するとデータセットの全ての要素をシャッフルしたものを返す．例を挙げてこの関数の使用方法について解説する.

[37] 厳密には**疑似乱数**の生成である.

次の例はアルファベット小文字の ”a”〜”z” を要素として持つベクトルを作成するものである.

例. サンプルデータの作成

```
> d1 <- unlist( strsplit("abcdefghijklmnopqrstuvwxyz","") )  Enter  ←サンプルデータの作成
> d1  Enter      ←内容確認
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
```

このようにしてできた文字列型のベクトル d1 からランダムに要素を取り出す例を示す.

例. ランダムサンプリング（先の例の続き）

```
> sample(d1,1)  Enter      ←ランダムに 1 個取り出す
[1] "e"                    ←得られたデータ
> sample(d1,3)  Enter      ←ランダムに 3 個取り出す
[1] "z" "l" "g"             ←得られたデータ
> sample(d1)  Enter      ←データのシャッフル
 [1] "d" "h" "k" "y" "t" "j" "w" "p" "v" "r" "e" "b" "m" "l" "s" "q" "n" "i" "a"
[20] "f" "o" "x" "z" "g" "c" "u"
```

sample 関数によるサンプリングも set.seed によって再現性を持たせることができる.

3.3 確率分布

R には各種の**確率分布**に関する関数群（表 13）が提供されている．ここでいう確率分布とは，**確率質量関数**[38] と**確率密度関数**[39] の両方を指すので，今後の解説においてどちらを意味するかは文脈によって判断されたい.

表 13: R で使用できる確率分布（一部）

確率分布	表記	確率分布	表記	確率分布	表記
一様分布	unif	正規分布	norm	対数正規分布	lnorm
ガンマ分布	gamma	ベータ分布	beta	χ 2 乗分布	chisq
t 分布	t	F 分布	f	指数分布	exp
ロジスティック分布	logis	二項分布	binom	ポアソン分布	pois
幾何分布	geom	超幾何分布	hyper	多項分布	multinom

R では表 13 に挙げる確率分布の「表記」に基づく各種の関数（表 14）が使用できる．例えば「表記」が「xxx」の確率分布に関して,「**pxxx**」は確率密度関数（もしくは確率質量関数）を意味する.

表 14: 確率分布に関する関数

関数	説明	関数	説明
dxxx	確率密度関数（もしくは確率質量関数）	**pxxx**	累積分布関数
qxxx	クォンタイル関数（分位数関数）	**rxxx**	乱数作成関数

正規分布に関連する各種の関数の実行例を以下に示す.

[38]離散的な確率変数に対する確率を与える関数.

[39]連続的な確率変数に対する確率密度を与える関数.

例. 正規分布の確率分布関数

```
> x <- seq(-5,5,by=0.1) Enter ←定義域
> y <- dnorm(x,mean=0,sd=1) Enter ←値域
> plot(x,y,type="l") Enter ←プロット
```

これで図 8 の (a) が作図される.

例. 正規分布の累積分布関数（先の例の続き））

```
> y <- pnorm(x,mean=0,sd=1) Enter ←値域
> plot(x,y,type="l") Enter ←プロット
```

これで図 8 の (b) が作図される.

例. 正規分布のクォンタイル関数（先の例の続き）

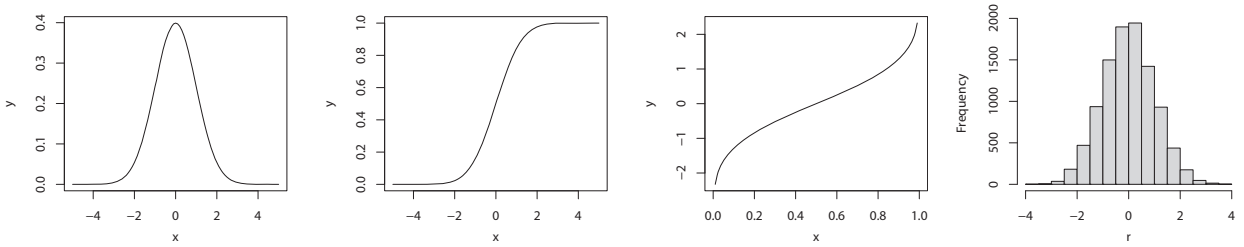
```
> x <- seq(0,1,by=0.01) Enter ←定義域
> y <- qnorm(x,mean=0,sd=1) Enter ←値域
> plot(x,y,type="l") Enter ←プロット
```

これで図 8 の (c) が作図される.

例. 正規乱数

```
> r <- rnorm(10000,mean=0,sd=1) Enter
> hist(r,main="") Enter ←ヒストグラム
```

これで図 8 の (d) が作図される.



(a) 確率分布関数　(b) 累積分布関数　(c) クォンタイル関数　(d) 乱数生成

図 8: 正規分布に関する各種の関数

対数正規分布に関連する各種の関数の実行例を以下に示す.

例. 対数正規分布の確率分布関数

```
> x <- seq(0.01,5,by=0.05) Enter
> y <- dlnorm(x,meanlog=0,sdlog=1) Enter
> plot(x,y,type="l") Enter ←プロット
```

これで図 9 の (a) が作図される.

例. 対数正規分布の累積分布関数（先の例の続き））

```
> y <- plnorm(x,meanlog=0,sdlog=1) Enter
> plot(x,y,type="l") Enter ←プロット
```

これで図 9 の (b) が作図される.

例. 対数正規分布のクォンタイル関数（先の例の続き）

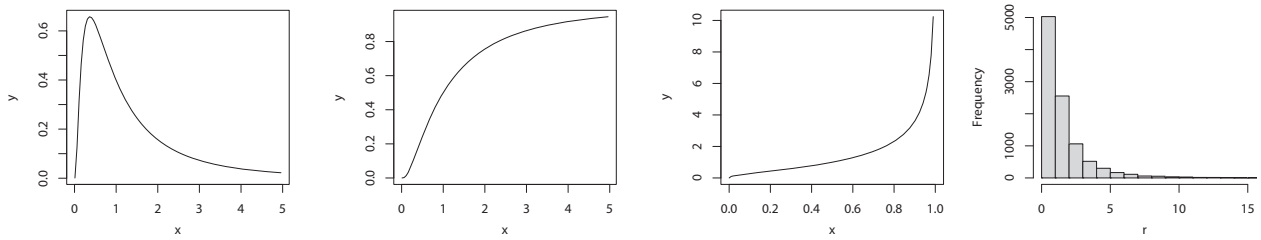
```
> x <- seq(0,1,by=0.01) Enter
> y <- qlnorm(x,meanlog=0,sdlog=1) Enter
> plot(x,y,type="l") Enter ←プロット
```

これで図 9 の (c) が作図される.

例. 対数正規乱数

```
> r <- rlnorm(10000,meanlog=0,sdlog=1) Enter
> hist(r,xlim=c(0,15),breaks=40,main="") Enter
```

これで図 9 の (d) が作図される.



(a) 確率分布関数　(b) 累積分布関数　(c) クォンタイル関数　(d) 乱数生成

図 9: 対数正規分布に関する各種の関数

3.4 要約統計量

ここでは例を挙げながら**要約統計量**を求める方法について解説する.

次のようにしてサンプルデータとなる正規乱数を作成する.

例. サンプルデータの作成

```
> v1 <- rnorm(10000,mean=0,sd=1)  Enter      ← μ = 0, σ = 1 の正規乱数を 10,000 個作成
```

以後，このベクトル v1 を対象にして解説する．

3.4.1 データの個数，最小値，最大値，値の範囲

データの個数は length，最小値は min，最大値は max，値の範囲は range の各関数で求めることができる．

例. データの個数，最小値，最大値，値の範囲を求める（先の例の続き）

```
> n <- length(v1)  Enter      ←データ個数の取得
> n  Enter                    ←確認
[1] 10000                     ←結果
> min(v1)  Enter              ←最小値の取得
[1] -3.627529                 ←結果
> max(v1)  Enter              ←最大値の取得
[1] 3.890746                  ←結果
> range(v1)  Enter            ←値の範囲の取得
[1] -3.627529 3.890746        ←結果
```

3.4.2 平均値，標準偏差，分散，中央値

平均値は mean，**不偏標準偏差**は sd，**不偏分散**は var，**中央値**は median の各関数で求めることができる．

例. 平均値，不偏標準偏差，不偏分散，中央値（先の例の続き）

```
> mean(v1)  Enter             ←平均値の取得
[1] -0.007247946              ←結果
> sd(v1)  Enter               ←不偏標準偏差の取得
[1] 1.00287                   ←結果
> var(v1)  Enter              ←不偏分散の取得
[1] 1.005748                  ←結果
> median(v1)  Enter           ←中央値の取得
[1] -0.007223977              ←結果
```

■ 注意

標本標準偏差，**標本分散**を求める組み込み関数は無いので，データ個数 n を用いて次のような計算によって求める．

$$\text{標本標準偏差} = \sqrt{\frac{n-1}{n}} \times \text{不偏標準偏差}$$

$$\text{標本標分散} = \frac{n-1}{n} \times \text{不偏分散}$$

例. 標本標準偏差，標本分散（先の例の続き）

```
> sd(v1) * sqrt((n-1)/n)  Enter      ←標本標準偏差の取得
[1] 1.00282                          ←結果
> var(v1) * (n-1)/n  Enter           ←標本分散の取得
[1] 1.005647                         ←結果
```

3.4.3 分位数（クォンタイル）

関数 quantile を使用すると，データセットの指定した確率の**分位数（クォンタイル）**を取得することができる．

書き方：　quantile(データセット, progs=確率)

「確率」で指定した「データセット」の分位数を求める．

例. 四分位数を求める（先の例の続き）

```
> quantile( v1, probs=c(0,0.25,0.5,0.75,1) )  Enter
          0%          25%          50%          75%         100%
-3.627528834 -0.694292916 -0.007223977 0.663339321 3.890745731
```

このように百分率[40] のラベルの付いたベクトルとして結果が得られる．

3.5 質的データの集計

地名，人名などを始めとする非数値の離散的なデータは**質的データ**[41] と呼ばれ，多くの場合，文字列型のデータもしくは整数値で表現される．ここでは，質的データを項目毎に集計する方法について例を挙げて解説する．

まず次のような処理によってサンプルデータを作成する．

例. サンプルデータの作成

```
> d0 <- c(rep("お好み焼き",80),rep("カレーライス",120),  Enter  ←各種文字列の繰り返しを
+          rep("おでん",60),rep("肉うどん",40))  Enter              連結する処理
> d1 <- sample(d0)  Enter    ←上で作成したデータ d0 をシャッフル
> d1[1:5]  Enter    ←先頭 5 要素を確認
[1] "カレーライス" "お好み焼き"   "おでん"       "肉うどん"     "カレーライス"
```

料理の各種メニューを表す文字列を繰り返したものを連結し，それをシャッフルすることでサンプルのデータセット d1 ができた．このデータセットには ”お好み焼き” が 80 個，”カレーライス” が 120 個，”おでん” が 60 個，”肉うどん” が 40 個乱雑な順番で含まれている．質的データの集合を各要素毎に件数を集計するには table 関数を使用する．

書き方： table(質的データ)

「質的データ」を各要素毎に集計して結果を table オブジェクトとして返す．

例. table 関数による集計処理（先の例の続き）

```
> t1 <- table(d1)  Enter    ←集計処理の実行
> t1  Enter    ←内容確認
d1                                                ←集計対象のデータセットの名前
     おでん  お好み焼き カレーライス     肉うどん     ←集計した要素
         60          80         120          40     ←件数
```

正しく集計できていることがわかる．得られた table オブジェクトの内容（集計した項目と件数）にアクセスする例を次に示す．

例. table オブジェクトの内容にアクセスする（先の例の続き）

```
> names(t1)  Enter    ← table オブジェクト t1 の名前の並びを確認
[1] "おでん" "お好み焼き" "カレーライス" "肉うどん"    集計した項目の並び
> t1["おでん"]  Enter    ←項目名を指定して集計結果を参照
おでん
   60
```

3.5.1 factor オブジェクト（因子オブジェクト）

質的データは factor **オブジェクト**にすることで集計の際の処理効率を高めることができる．

書き方： factor(質的データ)

「質的データ」を集計処理に有利な factor オブジェクトに変換する．

[40] 分位数を百分率の確率で求めた場合は**パーセンタイル**と呼ぶ．

[41] **カテゴリカルデータ**あるいは**カテゴリーデータ**とも呼ばれる．

例. 文字列を要素とするベクトルを factor オブジェクトに変換する（先の例の続き）

```
> f1 <- factor(d1)  Enter      ←文字列ベクトル d1 を factor オブジェクト f1 に変換
> f1[1:5]  Enter      ←先頭 5 要素を確認
[1] カレーライス お好み焼き  おでん  肉うどん  カレーライス
Levels:  おでん お好み焼き カレーライス 肉うどん      ← levels 属性が付加されている
```

得られた factor オブジェクトには，元の質的データ d1 の要素を整理して一意に並べた lebels 属性が付加されているが，f1 のデータ要素自体は元の文字列ベクトル d1 と同じものに見える．しかし factor オブジェクト f1 の要素の型を typeof 関数で調べると整数型であることがわかる．質的データをこのような形式に編成することで，各要素を単純な整数値として扱うことができ，データを主記憶に保持する際の格納効率を高めると同時に，集計処理の際の処理効率を高めることができる．

例. f1 の要素のデータ型と levels 属性を取得する（先の例の続き）

```
> typeof(f1)  Enter      ←要素の型を調べる
[1] "integer"                    ←整数型である
> levels(f1)  Enter      ← levels 属性を参照する
[1] "おでん" "お好み焼き" "カレーライス" "肉うどん"
```

levels 関数は factor オブジェクトの levels 属性にアクセスする．

factor オブジェクトは table 関数で集計することができる．

例. table 関数による集計処理（先の例の続き）

```
> t1 <- table(f1)  Enter      ←集計処理の実行
> t1  Enter      ←内容確認
f1                                          ←集計対象のデータセットの名前
      おでん  お好み焼き  カレーライス      肉うどん      ←集計した要素
        60          80          120           40      ←件数
```

3.5.2 ordered オブジェクト（順序付き因子オブジェクト）

factor クラスの派生クラスに ordered クラスがある．このクラスのオブジェクトは値に順序（大小関係）のある質的データを扱う．

書き方：　ordered(質的データ, level=順番指定のベクトル)

「質的データ」を「順番指定のベクトル」で順序付けられた ordered オブジェクトに変換する．以下に例を挙げて解説する．

例. サンプルデータの作成

```
> d1 <- sample(c(rep("秀",5),rep("優",15),rep("良",30),rep("可",20),rep("不",10)))  Enter
> d1[1:5]  Enter      ←先頭 5 要素を確認
[1] "秀" "可" "優" "良" "不"
```

これで ”秀”，”優”，”良”，”可”，”不” の 5 種類の質的データから成るサンプルデータ d1 ができた．これを ordered オブジェクトに変換する．

例. 文字列を要素とするベクトルを ordered オブジェクトに変換する（先の例の続き）

```
> f1 <- ordered(d1,level=c("不","可","良","優","秀"))  Enter
> f1[1:5]  Enter      ←先頭 5 要素を確認
[1] 秀 可 優 良 不
Levels:  不 < 可 < 良 < 優 < 秀
> class(f1)  Enter      ←クラスを確認
[1] "ordered" "factor"      ← factor クラスの派生クラス ordered であることがわかる
```

levels 属性を見ると質的データの間に「不 < 可 < 良 < 優 < 秀」という関係が確認できる．この質的データの順序は

大小を比較する関係演算子で扱うことができる.

例. ”優” より大きなデータを調べる（先の例の続き）

```
> which( f1>"優" )  Enter      ←”優” より大きなデータを抽出する
[1] 1 35 39 45 67              ←結果
```

これは ”優” よりも大きな項目である ”秀” のデータの位置（インデックス）が得られていることを示している.

ordered オブジェクトは factor オブジェクトと同じように table 関数で集計することができる.

例. table 関数による集計処理（先の例の続き）

```
> t1 <- table(f1)  Enter      ←集計処理の実行
> t1  Enter      ←内容確認
f1                             ←集計対象のデータセットの名前
不 可 良 優 秀        ←集計した要素（昇順に並んでいる）
10 20 30 15 5         ←件数
```

3.6 度数分布

数値のデータセットを区間で分類して度数分布を調査するには次のような手順を踏む.

手順 1. 数値データ集合の各要素を分類するための**区間情報**を作成する.

手順 2. 上記 1 で作成された区間情報を元に出現頻度（件数）を集計する.

$\mu = 0, \sigma = 1$ の 10,000 個の正規乱数を用いて度数分布を調査する作業を例に挙げて解説する.

例. サンプルデータの作成

```
> v1 <- rnorm(10000,mean=0,sd=1)  Enter
```

今回，-4〜4 の範囲を 0.4 刻みの区間（20 個の区間）に分け，各区間に存在するデータの件数を集計する作業を想定する.

手順 1.

関数 cut を用いて，数値データ集合の各要素が所属する区間の情報を作成する.

書き方：　cut(数値データセット, 区間の列)

「区間の列」から数値を分類するための区間情報を作り,「数値データセット」の各要素がどの区間に属するかの情報を作成する.「区間の列」には区間の区切りの位置の並びをベクトルで与える．cut の実行結果は **factor オブジェクト**の形で得られる.

例. cut で数値データを区間データにする（先の例の続き）

```
> c1 <- cut(v1,seq(-4,4,by=0.4))  Enter      ←データ集合 v1 の各データの区間情報を c1 に作成
> class(c1)  Enter      ←クラスを確認
[1] "factor"              ← factor クラスである
> c1[1:18]  Enter      ←先頭 18 個を表示
 [1] (-1.2,-0.8]   (-0.4,0]    (0,0.4]      (-1.2,-0.8]   (0,0.4]    (0,0.4]
 [7] (0,0.4]       (0.8,1.2]   (-1.6,-1.2] (1.2,1.6]    (-0.8,-0.4] (-1.2,-0.8]
[13] (-0.8,-0.4] (0,0.4]      (0,0.4]      (-0.4,0]      (-1.2,-0.8] (-0.8,-0.4]
20 Levels:  (-4,-3.6] (-3.6,-3.2] (-3.2,-2.8] (-2.8,-2.4] ...  (3.6,4]   ←区間の並び
```

得られた factor オブジェクト c1 の要素は，元のデータ v1 の要素が所属する区間を意味する．このことを次の例で確かめる.

例. v1 の要素と c1 の要素の対応を調べる（先の例の続き）

```
> v1[1]  Enter      ←元の数値データ集合の先頭要素を調べる
[1] -0.9619334      ←値
> c1[1]  Enter      ←得られた区間情報の先頭要素を調べる
[1] (-1.2,-0.8]     ←所属区間
20 Levels:  (-4,-3.6] (-3.6,-3.2] (-3.2,-2.8] (-2.8,-2.4] ...  (3.6,4]   ←区間の並び
```

この例から v1 の先頭要素の値 -0.9619334 は区間 (-1.2,-0.8] に属していることがわかる．c1 の要素の表記は数学におけるの区間の表記に沿っている．すなわち区間 (-1.2,-0.8] は「-1.2 より大きく-0.8 以下の区間」を意味している．（下限の -1.2 は含まず，上限の -0.8 は含む）

手順 2.

数値データの所属する区間を**質的データ**と見なしてその出現頻度を集計する．そのために table 関数を使用する．

書き方：　table(factor オブジェクト)

「factor オブジェクト」の要素の出現頻度を集計し，結果を **table オブジェクト**として返す．

例. factor オブジェクトの要素の集計（先の例の続き）

```
> t1 <- table(c1)  Enter      ← factor オブジェクト c1 の要素の出現頻度を集計
> class(t1)  Enter      ←クラスを確認
[1] "table"       ← table クラスである．
> t1  Enter      ←集計結果の確認
c1
(-4,-3.6] (-3.6,-3.2] (-3.2,-2.8] (-2.8,-2.4]   (-2.4,-2]   (-2,-1.6] (-1.6,-1.2] (-1.2,-0.8]
        1           2          14          46         163         357         604         965
(-0.8,-0.4]   (-0.4,0]     (0,0.4]   (0.4,0.8]   (0.8,1.2]   (1.2,1.6]     (1.6,2]     (2,2.4]
     1326        1550        1591        1276         961         597         308         155
(2.4,2.8]   (2.8,3.2]   (3.2,3.6]     (3.6,4]
       57          17           9           1
```

得られた table オブジェクトの要素は集計結果の整数値であり，各要素は factor の要素が示す区間の文字列を名前として持つ．例えば，上の例では最初の区間 (-4,-3.6] が値 1 を持つが，次のようにしてそれを参照することができる．

例. 区間の名前を指定して table オブジェクトにアクセスする（先の例の続き）

```
> t1["(-4,-3.6]"]  Enter      ←参照
(-4,-3.6]      ← table 要素の名前
        1      ← table 要素の値
```

3.6.1　最頻値

度数分布調査の結果として得られた table オブジェクトの最大値を max によって求める処理を応用すると元のデータセットの**最頻値**を取得することができる．すなわち，table オブジェクトの中の最大値となる要素の位置を which によって求めることで最頻値となる区間を求めることができる．

例. 最頻値を求める（先の例の続き）

```
> mv <- max(t1)  Enter      ←最大値を求める関数 max
> mv  Enter              ←データ件数の最大値を確認
[1] 1591                 ←最大値
```

例. table オブジェクト中の最大値の位置（最頻の区間）を求める（先の例の続き）

```
> mi <- which( t1==max(t1) )  Enter      ← t1 の最大値の位置を求める
> mi  Enter              ←最大値の位置を確認
(0,0.4]                  ←最頻の区間
     11                  ←最頻値のインデックス
```

例. 最頻の区間の名前を求める（先の例の続き）

```
> names(mi)  Enter      ←名前（区間を意味する）を求める
[1] "(0,0.4]"     ←最頻値の区間
```

最頻値の位置は which.max を用いて求めることもできる.

例. table オブジェクト中の最大値の位置を求める（先の例の続き）

```
> mi <- which.max( t1 )  Enter     ← t1 の最大値の位置を求める
> mi  Enter                ←最大値の位置を確認
(0,0.4]                  ←最頻の区間
     11                   ←最頻値のインデックス
```

4　データの可視化

4.1　最も基本的な可視化機能（Traditional）

plot 関数を使用することで**散布図**や**折れ線グラフ**を作成することができる．

書き方：　plot(X, Y)

ベクトル X，Y の各要素を対応させた座標位置をプロットする．以下に例を示しながら説明する．

例. サンプルデータの作成

```
> x <- seq( -2*pi, 2*pi, length=20 )  Enter
```
← -2π〜2π の範囲の 20 個の値（等間隔）

```
> y <- sin(x)  Enter
```
←上記のベクトルの各要素に対する正弦関数のベクトル

このようにして作成されたベクトル x に対する y のプロットを plot 関数によって作図する例を次に示す．

例. plot 関数による作図（先の例の続き）

```
> plot(x,y)  Enter
```
←作図処理の実行

この処理によって図 10 のようなウィンドウが表示される．

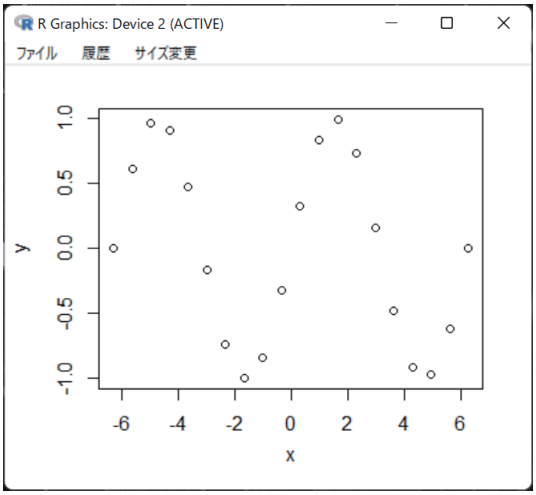


図 10: plot 関数による作図

4.1.1　グラフィクスデバイス

図 10 のウィンドウは**グラフィクスデバイス**（graphics devices）と呼ばれるものであり，R システムが作図の対象とする仮想的な表示装置である．R ではこれに対して作図に関する各種の操作を実行する．またこのウィンドウの「ファイル」メニューから「別名で保存」を選択することで，表示されているグラフを各種の画像フォーマットのファイルとして保存することができる．

dev.new を実行することで，グラフィクスデバイスのサイズを指定することができる．

書き方：　dev.new(width=横幅, height=高さ)

「横幅」,「高さ」にサイズの値（インチ単位）[42] を与える．

例. グラフィクスデバイスのサイズを指定した作図（先の例の続き）

```
> dev.new(width=6.4,height=3.6)  Enter
```
←サイズの指定

```
> plot(x,y)  Enter
```
←作図処理の実行

この処理によって図 11 のようなウィンドウが表示される．

[42] 使用するディスプレイによって，実際に表示されるウィンドウのサイズは異なる．

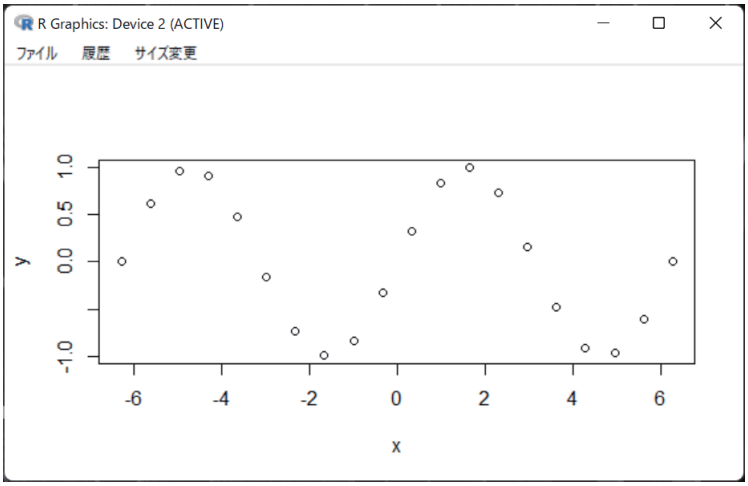


図 11: グラフィクスデバイスのサイズを指定した作図

4.1.2 グラフのタイトル，軸ラベルの表示

グラフ上部にタイトルを表示するには plot に引数「`main=`**タイトル**」を与える．また引数「`xlab=`**横軸ラベル**」，「`ylab=`**縦軸ラベル**」を与えることで，横軸ラベル，縦軸ラベルを表示することができる．

例. グラフ上部にタイトルを表示する（先の例の続き）

```
> plot(x,y,main="y = sin(x)", xlab="定義域 x",ylab="値域 y")
```
Enter

この処理によって図 12 のようなグラフが表示される．

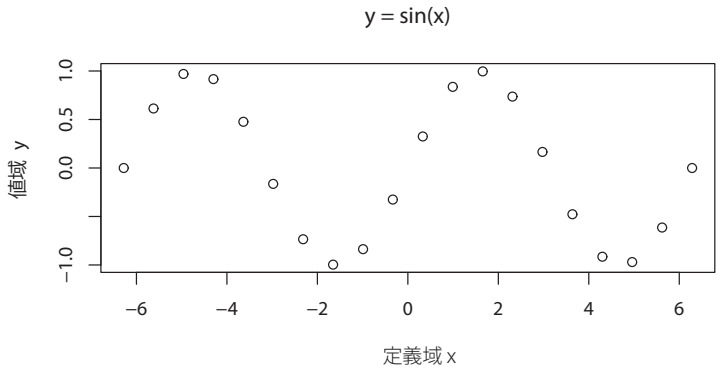


図 12: 図のタイトルと軸ラベルを付加した例

plot に引数「`sub=`**サブタイトル**」を与えて，グラフ下部にサブタイトルを表示することもできる．

4.1.3 プロットの点，線の指定

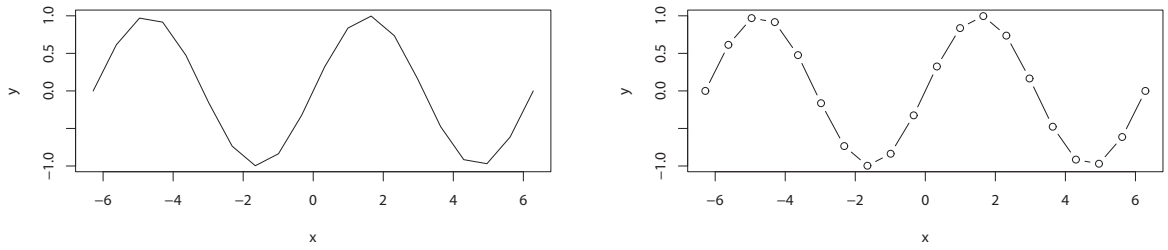
plot 関数の引数にオプション「type=」を与えることでプロットの点や線の描画を制御できる．先に作成したベクトル x，y を plot 関数にオプション「type=」を与えて作図した様子を図 13 に示す．

例. 折れ線グラフのプロット（先の例の続き）

```
plot(x,y,type="l")
```
Enter

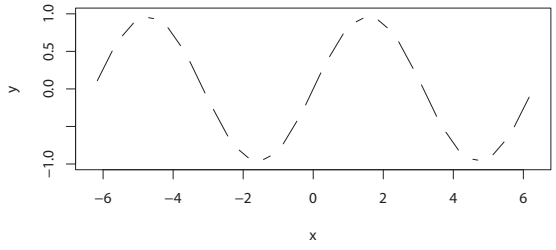
この結果，図 13 の (i) のようなグラフが表示される．

デフォルトでは「`type="p"`」であり，小さな「○」がプロットされる．これを応用すると**散布図**が作成できる．(次の例)

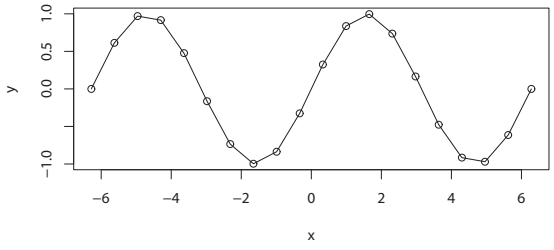


(i) type=”l”（小文字のエル）

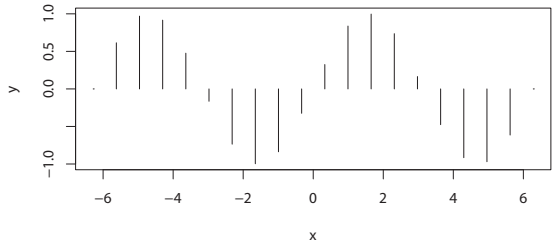
(ii) type=”b”



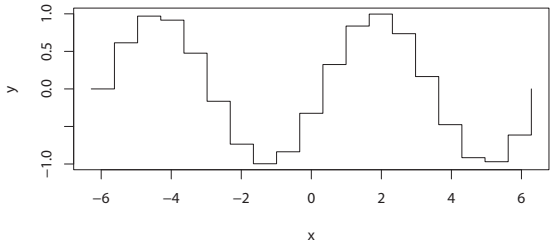
(iii) type=”c”



(iv) type=”o”（小文字のオー）



(v) type=”h”



(vi) type=”s”

図 13: オプション「type=」を与えた作図

例. 散布図

```
> r1 <- rnorm(1000,mean=0,sd=1)   Enter    ← μ = 0, σ = 1 の正規乱数（1000 個）
> r2 <- rnorm(1000,mean=0,sd=1)   Enter    ←上記と同様の別の乱数
> dev.new(width=3.6,height=4)     Enter    ←グラフィクスデバイスのサイズの設定
> plot(r1,r2)                     Enter    ←プロットの実行
```

この結果，図 14 のようなグラフが表示される．

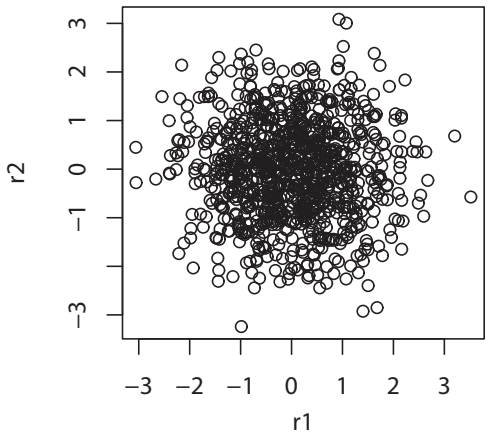


図 14: 散布図

プロットの線の太さを設定するには plot に引数「**lwd=太さ**」を与える．この場合の太さは基準となる線の太さに対する倍率である．プロットの線種を変更するには plot に引数「**lty=番号**」を与える．(図 15 参照))

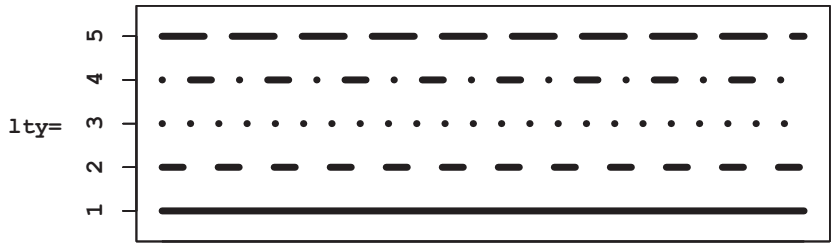


図 15: プロットの線種

プロット点の種類を変更するには plot に引数「pch=**番号**」を与える.（図 16 参照）また点の大きさは plot に引数「cex=**値**」を与える．この場合の値は基準となる点の大きさに対する倍率である.

図 16: プロット点の種類

4.1.4 プロットの色の指定

plot に引数「col=**番号**」を与えることでプロットの色を指定することができる.（図 17 参照）

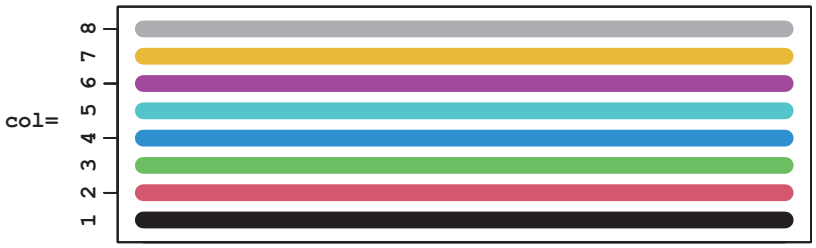


図 17: プロットの色

4.1.5 座標軸と格子の表示

abline を用いると水平，垂直の線を描画することができる．これを応用するとグラフ中に座標軸や格子を描くことができる.

書き方：　abline(h=水平線の位置, v=垂直線の位置)

「水平線の位置」,「垂直線の位置」はベクトルの要素として複数与えることができる．また，線種，線の太さ，色などの指定には先に解説した plot に対する引数が使用できる．以下に使用方法の例を示す.

例. サンプルデータの作成

```
> x <- seq(-1.17,1.17,length=80)   Enter
> y <- x^3 - x   Enter
```

これは $y = x^3 - x$ の $-1.17 \leq x \leq 1.17$ の範囲のデータ列を作成する処理である．このデータをプロットし，グラフ中に座標軸と格子を描く例を次に示す.

例. データをプロットして座標軸と格子を描く（先の例の続き）

```
> plot(x,y,type="l",lwd=2)   Enter     ←データのプロット
> abline(h=seq(-0.4,0.4,by=0.2),v=seq(-1,1,by=0.5),col=8,lty=3,lwd=1) Enter ←格子の描画
> abline(h=0,v=0,col=8,lwd=2)   Enter     ←座標軸の描画
```

この処理により図 18 のようなグラフが表示される.

4.1.6 複数のプロットを作成する方法

1 つのグラフィックデバイス上に複数のプロットを作成する方法を例を挙げて説明する．まず次のようなサンプルデータ（正弦関数，余弦関数）を作成する.

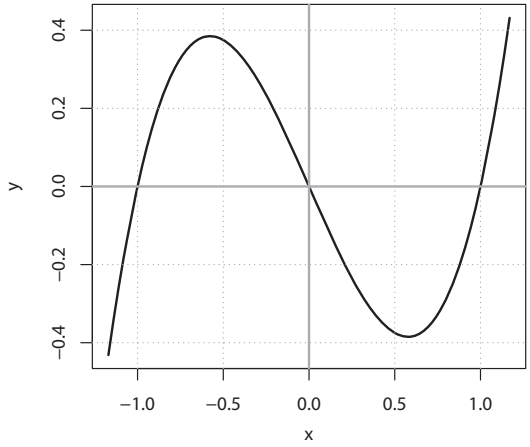


図 18: 座標軸と格子の表示

例. サンプルデータの作成

```
x <- seq( -2*pi, 2*pi, length=80 )  Enter    ←横軸データの作成
y_sin <- sin(x)  Enter    ←縦軸データ 1（正弦関数）
y_cos <- cos(x)  Enter    ←縦軸データ 2（余弦関数）
```

このようにして作成した 2 種類の関数のプロットを作成する.

■ 横方向にグラフを並べる（左から右の方向）

グラフィックデバイス上に横方向に並んだ複数のプロット領域を作成するには

par(mfrow=c(行数, 列数))

この処理によって複数のプロット領域が横方向に作成され，plot による描画を順番に表示する.（次の例）

例. 横方向に複数のグラフを並べる（先の例の続き）

```
> par( mfrow=c(1,2) )  Enter  ← 1 行 2 列の領域
> plot(x,y_sin,type="l")  Enter  ← 1 つ目の作図
> plot(x,y_cos,type="l")  Enter  ← 2 つ目の作図
```

この処理によって図 19 のようなグラフが作成される.

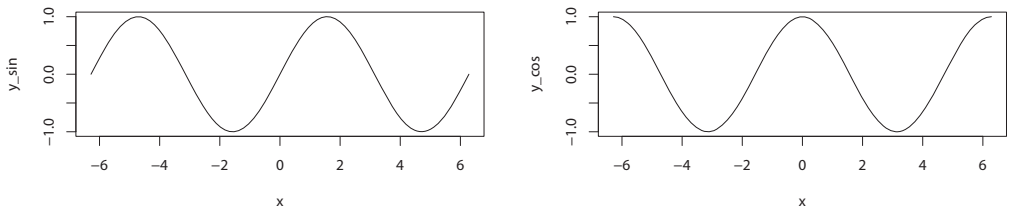


図 19: 横並びのプロット

■ 縦方向にグラフを並べる（上から下の方向）

グラフィックデバイス上に縦方向に並んだ複数のプロット領域を作成するには

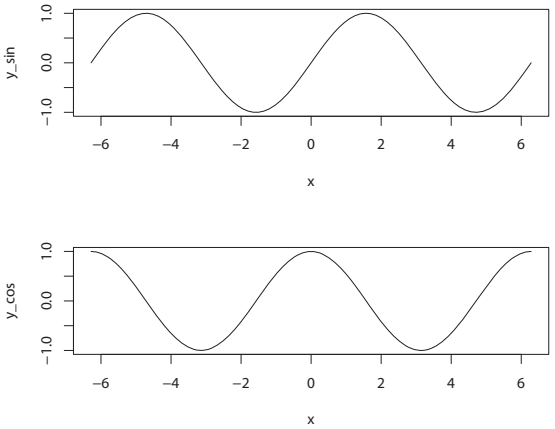
par(mfcol=c(行数, 列数))

この処理によって複数のプロット領域が縦方向に作成され，plot による描画を順番に表示する.（次の例）

例. 横方向に複数のグラフを並べる（先の例の続き）

```
> par( mfcol=c(2,1) )   Enter   ← 2 行 1 列の領域
> plot(x,y_sin,type="l")   Enter   ← 1 つ目の作図
> plot(x,y_cos,type="l")   Enter   ← 2 つ目の作図
```

この処理によって右のようなグラフが作成される．



縦並びのプロット

先の 2 つの作図例を比較すると，par 関数の引数 mfrow と mfcol は同じ効果をもたらすかのように見えるが次の作図例を見るとその違いがわかる．

例. サンプルデータの作成

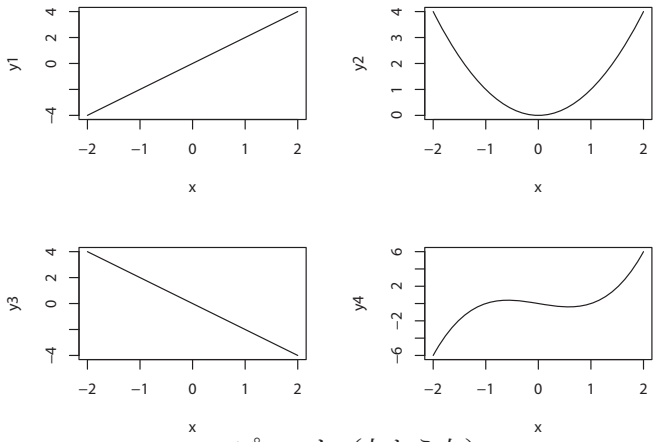
```
> x <- seq(-2,2,length=80)   Enter   ←横軸データの作成
> y1 <- 2*x; y2 <- x^2; y3 <- -2*x; y4 <- x^3-x   Enter   ← 4 つのデータの作成
```

このようにして作成したデータ y1〜y4 を 2 行 2 列の領域にプロットする例を次に示す．

例. 横方向に複数のグラフを並べる（先の例の続き）

```
> par( mfrow=c(2,2) )   Enter
> plot(x,y1,type="l")   Enter
> plot(x,y2,type="l")   Enter
> plot(x,y3,type="l")   Enter
> plot(x,y4,type="l")   Enter
```

この処理によって右のようなグラフが作成され，左から右にかけてグラフが並んでいることがわかる．

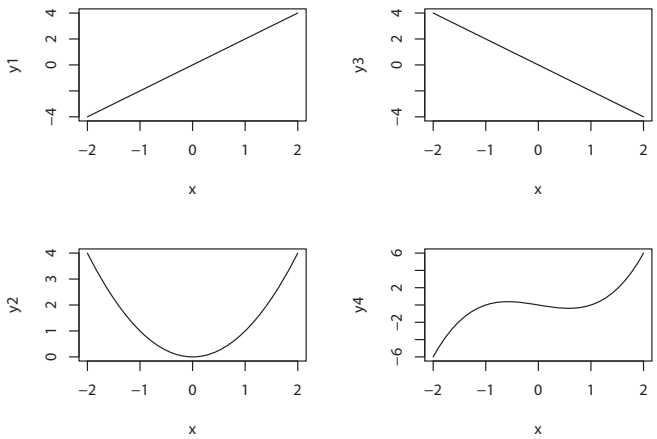


プロット（左から右）

例. 縦方向に複数のグラフを並べる（先の例の続き）

```
> par( mfcol=c(2,2) )   Enter
> plot(x,y1,type="l")   Enter
> plot(x,y2,type="l")   Enter
> plot(x,y3,type="l")   Enter
> plot(x,y4,type="l")   Enter
```

この処理によって右のようなグラフが作成され，上から下にかけてグラフが並んでいることがわかる．



プロット（上から下）

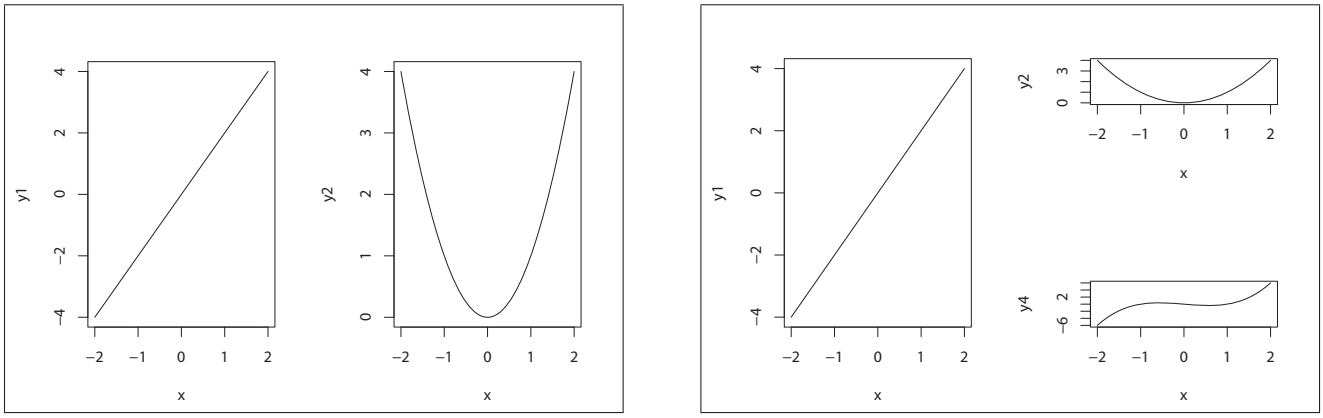
4.1.6.1 split.screen による方法

グラフの表示領域は split.screen で分割することができ，分割された表示領域（スクリーン）には番号が割り当てられる．そして，各表示領域の番号を指定してグラフを描画することができる．先に作成したベクトル x，y1，y2，y4 をプロットする例を挙げてスクリーンの分割について説明する．

例. スクリーンを 1 行 2 列に分割してプロット（先の例の続き）

```
> split.screen( figs=c(1,2) )  Enter     ←スクリーンを 1 行 2 列に分割
[1] 1 2                          ←上の処理の戻り値
> screen(1)  Enter                 ←スクリーン「1」に
> plot(x,y1,type="l")  Enter      ← x，y1 をプロット
> screen(2)  Enter                 ←スクリーン「2」に
> plot(x,y2,type="l")  Enter      ← x，y2 をプロット
```

これはスクリーンを 1 行 2 列に分割してプロットする例であり，この処理の結果，図 20 の (a) のように表示される．



(a) (b)

図 20: スクリーンを分割してプロット

書き方： split.screen(figs=c(行数, 列数))

これにより，指定した行数と列数にスクリーンが分割されて番号が割り当てられる．また，描画対象のスクリーンを指定するには次のように記述する．

書き方： screen(スクリーン番号)

上の実行例におけるスクリーンの分割の概略を図 21 の (a) と (b) に示す．

上の例のように分割したスクリーンを更に分割することもでき，スクリーン「2」を更に 2 行 1 列に分割してプロットする例を示す．

例. 先の実行例のスクリーン「2」を更に分割してプロット（先の例の続き）

```
> split.screen( figs=c(1,2) )  Enter     ←スクリーンを 1 行 2 列に分割
[1] 1 2                          ←上の処理の戻り値
> split.screen( figs=c(2,1), screen=2 )  Enter     ←スクリーン「2」を更に 2 行 1 列に分割
[1] 3 4                          ←上の処理の戻り値
> screen(1)  Enter                 ←スクリーン「1」に
> plot(x,y1,type="l")  Enter      ← x，y1 をプロット
> screen(3)  Enter                 ←スクリーン「3」に
> plot(x,y2,type="l")  Enter      ← x，y2 をプロット
> screen(4)  Enter                 ←スクリーン「4」に
> plot(x,y4,type="l")  Enter      ← x，y4 をプロット
```

この処理の結果，図 20 の (b) のように表示される．この処理によるスクリーンの分割の概略を図 21 の (b) と (c) に示す．

4.1.6.2 複数のグラフを重ねて描画する方法

plot による作図を複数回実行すると，最後に実行された plot によるグラフが表示される．このことを例を挙げて確かめる．

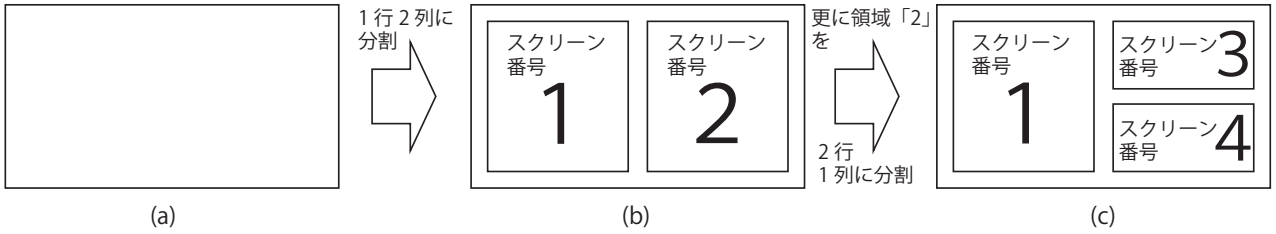


図 21: スクリーンの分割の概要

例. サンプルデータの作成

```
> x <- seq(-5,5,length=10)  Enter      ←横軸データ x の作成
> y1 <- 2*x - 5  Enter                 ←縦軸データ列 2x − 5 の作成
> y2 <- 2*x  Enter                     ←縦軸データ列 2x の作成
> y3 <- 2*x + 5  Enter                 ←縦軸データ列 2x + 5 の作成
```

このようにして作成したデータ x，y1，y2，y3 に対して plot を立て続けに実行する.（次の例）

例. 上記 y1，y2，y3 を立て続けにプロットする（先の例の続き）

```
> plot(x,y1,type="l")  Enter
> plot(x,y2,type="l")  Enter
> plot(x,y3,type="l")  Enter
```

この結果，図 22 のように最後に実行した plot のグラフ（x，y3）のみがグラフィクスデバイス上に残る.

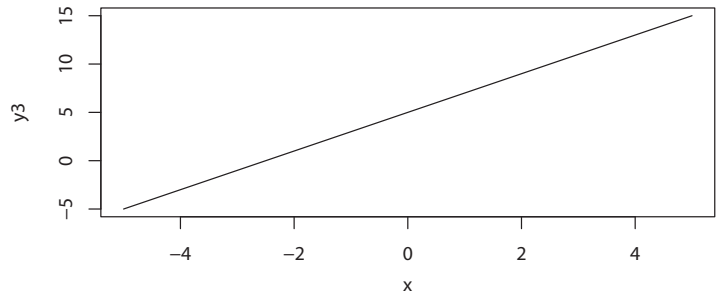


図 22: plot の連続実行の結果

先に作図したグラフを残す形でプロット処理を行うには次の plot の実行の前に「`par( new=TRUE )`」を実行する.

例. 先のプロットの上に重ねる形でプロットする（先の例の続き）

```
> plot(x,y1,type="l")  Enter     ←最初のプロット
> par(new=TRUE)  Enter           ←先のプロットを残す設定
> plot(x,y2,type="l")  Enter     ← 2 番目のプロット
> par(new=TRUE)  Enter           ←先のプロットを残す設定
> plot(x,y3,type="l")  Enter     ← 3 番目のプロット
```

この結果，図 23 のように表示される.（問題あり）

これは，各回の plot による作図がそのまま重ねられることによって得られたグラフであり，縦軸のラベルや目盛りの表示までもが重なっており，実用性が損なわれている．これを解決するための 1 つの方法として，各回の plot 処理において，表示する縦軸のプロット範囲を同一にするというものが考えられる.

プロット範囲を指定して plot を実行するには，plot に引数「**xlim=c(下限, 上限)**」（横軸の範囲の限定),「**ylim=c(下限, 上限)**」（縦軸の範囲の限定）を与える．これを応用した例を次に示す.

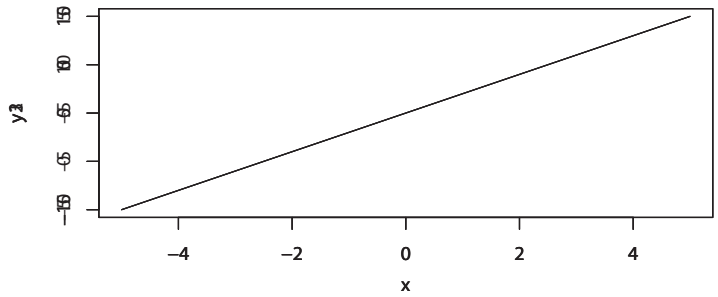


図 23: 重ねて plot で作図（問題あり）

例. 同一のプロット範囲で複数の plot を実行（先の例の続き）

```
> plot(x,y1,type="l",xlim=c(-5,5),ylim=c(-10,10),ylab="")  Enter    ←最初のプロット
> par(new=TRUE)  Enter
> plot(x,y2,type="l",xlim=c(-5,5),ylim=c(-10,10),ylab="")  Enter    ← 2 番目のプロット
> par(new=TRUE)  Enter
> plot(x,y3,type="l",xlim=c(-5,5),ylim=c(-10,10),ylab="y")  Enter    ← 3 番目のプロット
```

この例では，縦横のプロット範囲を明に指定し，更に縦軸ラベルも最後のプロットで表示している．この処理の結果図 24 のようなグラフが表示される．

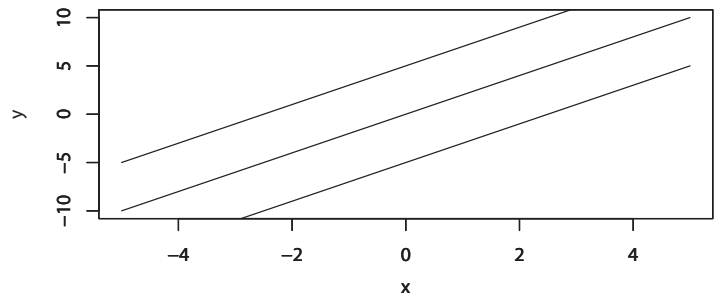


図 24: 同じプロット範囲を指定して複数のグラフをプロットした例

■ 更なる工夫

先に示した例のように，複数のグラフを同一の描画領域に重ねてプロットする場合は，各グラフを別々の色で表示したり，legend を使用して凡例を表示するなどの工夫をすると良い．

書き方：　legend(位置, legend=c(凡例の並び), lty=線種, col=c(色の並び), lwd=線の太さ)

「位置」は凡例を表示する図中の位置を指定するものであり，

`"bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right", "center"`

といったものが指定できる．これを応用した例を次に示す．

例. 色分けと凡例（先の例の続き）

```
> plot(x,y1,type="l",xlim=c(-5,5),ylim=c(-10,10),ylab="",col=2,lwd=2)  Enter
> par(new=TRUE)  Enter
> plot(x,y2,type="l",xlim=c(-5,5),ylim=c(-10,10),ylab="",col=3,lwd=2)  Enter
> par(new=TRUE)  Enter
> plot(x,y3,type="l",xlim=c(-5,5),ylim=c(-10,10),ylab="y",col=4,lwd=2)  Enter
> legend("bottomright",legend=c("2x-5","2x","2x+5"),lty=1,col=c(2,3,4),lwd=2)  Enter
```

この処理の結果，図 25 のようなグラフが表示される．

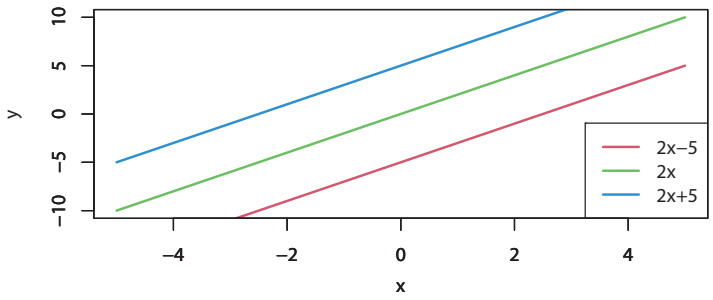


図 25: 色分けのプロットと凡例表示

4.1.7 対数グラフ

plot に引数「log="x"」もしくは「log="y"」を与えると**片対数グラフ**を作成することができる．また，引数「log="xy"」を与えると**両対数グラフ**を作成することができる．以下に例を挙げて説明する．

例. サンプルデータの作成

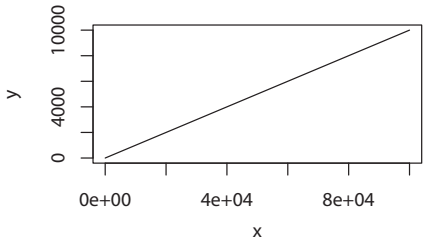
```
> x <- seq(1,1e5,length=100)  Enter      ←横軸データ
> y <- x/10  Enter      ←縦軸データ
```

これは $y = \dfrac{x}{10}$ の値を $1 \leq x \leq 10^5$ の範囲でデータ列にしたものである．これを可視化する例を次に示す．

例. 通常のスケールのプロット（先の例の続き）

```
> plot(x,y,type="l")  Enter
```

この処理によるプロットを右に示す．

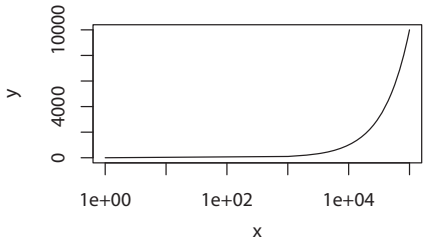


次に，横軸を対数軸にする例を示す．

例. 横軸を対数スケールにしてプロット（先の例の続き）

```
> plot(x,y,type="l",log="x")  Enter
```

この処理によるプロットを右に示す．

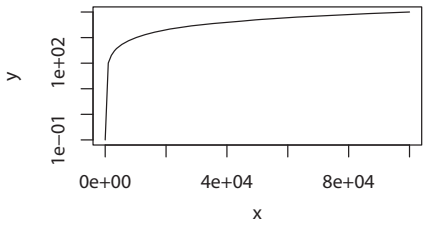


次に，縦軸を対数軸にする例を示す．

例. 縦軸を対数スケールにしてプロット（先の例の続き）

```
> plot(x,y,type="l",log="y")  Enter
```

この処理によるプロットを右に示す．



対数目盛りの格子を表示する例を次に示す．

次の例は垂直，水平のグリッド線の位置を表すベクトル gv，gh を作成するものである．

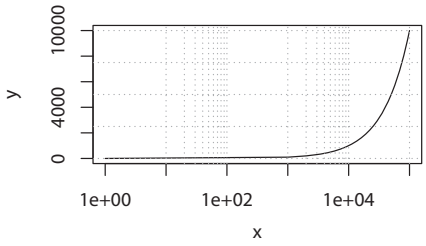
例. グリッド線の位置のデータ列を作成する（先の例の続き）

```
> gv <- c(seq(0,1e2,length=11),seq(0,1e4,length=11),seq(0,1e6,length=11))  Enter
> gh <- seq(0,1e4,length=5)  Enter
```

これらを abline の引数に与えてグラフにグリッド線を表示する．

例. 上で作成した位置にグリッド線を表示する（先の例の続き）

```
> plot(x,y,type="l",log="x")  Enter
> abline(v=gv,h=gh,col=8,lty=3,lwd=1)  Enter
```



この処理によるプロットを右に示す.

次に，縦横共に対数軸にする例を示す.

例. サンプルデータの作成

```
> x <- seq(10,10,length=51)^seq(0,50,length=51)  Enter   ←横軸データ
> y <- x/10  Enter   ←縦軸データ
```

これは先と同じく $y = \dfrac{x}{10}$ の値をデータ列にする例であるが，横軸の値の並びを $10^0, 10^1, 10^2, \cdots, 10^{50}$ としてデータの間隔が急激に大きくなる形にしている．これら x，y を可視化する例を示す.

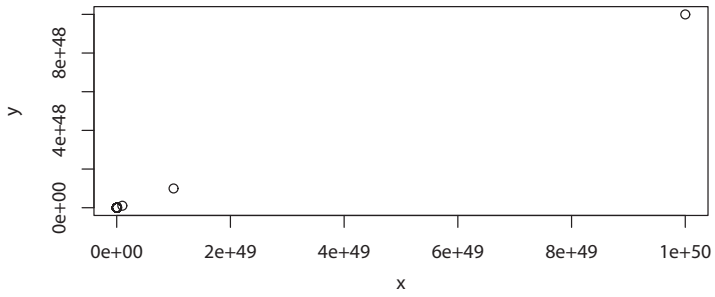
例. 通常のスケールのプロット（先の例の続き）

```
> plot(x,y,type="p")  Enter
```

この処理によるプロットを右に示す.

プロット点が 51 個あるが，値の小さい範囲に集中して点が表示されているので各点の確認が困難である.

次に，縦横両方の軸を対数軸にする例を示す.

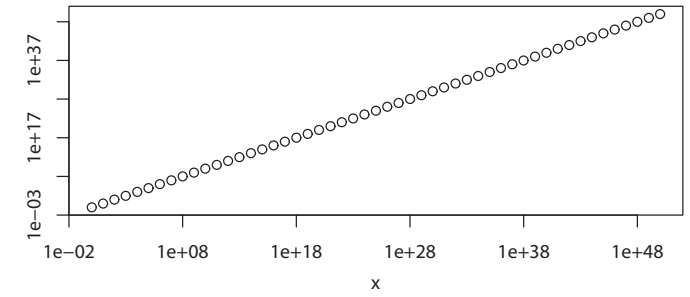


例. 縦横両方の軸を対数スケールにしてプロット（先の例の続き）

```
> plot(x,y,type="p",log="xy")  Enter
```

この処理によるプロットを右に示す.

値が小さい領域のプロット点も確認できる.



4.1.8 ヒストグラム（度数分布図）

関数 hist を使用することでデータ集合のヒストグラム（度数分布図）を作成することができる.

書き方： hist(データ, breaks=階級設定)

「データ」の要素を階級毎に集計して柱状グラフを作成する．引数「breaks=」は省略可能である．また plot に指定できる引数の多くが hist でも使用可能である．以下に例を示して hist の使用方法について説明する.

例. サンプルデータの作成

```
r <- rnorm(10000,mean=0,sd=1)  Enter   ← μ = 0, σ = 1 の正規乱数（10,000 個）
```

この処理によって $\mu = 0, \sigma = 1$ の正規乱数が 10,000 個作成され，それらを要素とするベクトルが r に得られている．これのヒストグラムを作成する例を次に示す.

例. ヒストグラムの作成（先の例の続き）

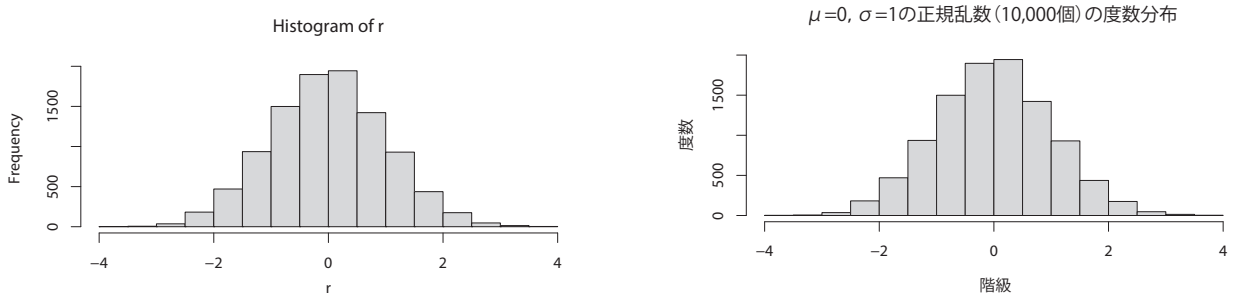
```
> hist(r)  Enter   ←ヒストグラムの作図
```

この処理により図 26 の (a) のようなヒストグラムが作成される.

例. タイトルと軸ラベルの指定（先の例の続き）

```
> hist(r,main="μ=0, σ=1 の正規乱数（10,000 個）の度数分布",xlab="階級",ylab="度数")  Enter
```

この処理により図 26 の (b) のようなヒストグラムが作成される.



(a) タイトルと軸ラベルの指定なし　　(b) タイトルと軸ラベルを指定して作図

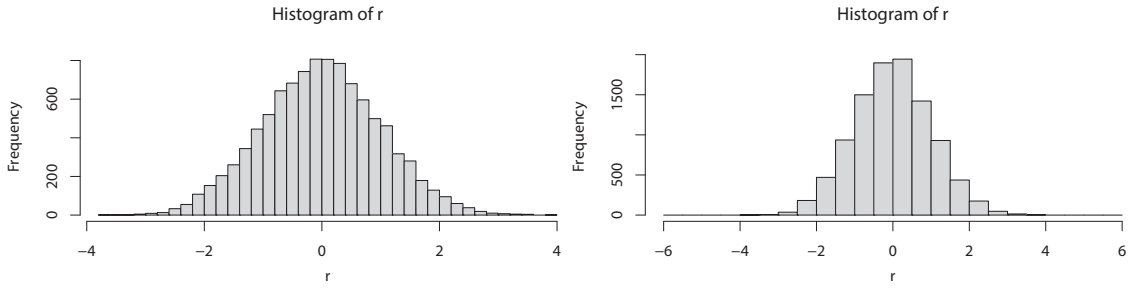
図 26: 関数 hist によるヒストグラム

hist に引数「breaks=**階級設定**」を与えることで階級の分割方法を設定できる．

例. 階級の分割数を指定（先の例の続き）

```
> hist(r,breaks=30)
```
Enter　←階級を 30 個に設定

この処理により図 27 の (a) のようなヒストグラムが作成される．



(a) breaks=30　　(b) breaks=seq(-6,6,by=0.5)

図 27: 関数 hist によるヒストグラム

例. 階級の分割位置の指定（先の例の続き）

```
> hist(r,breaks=seq(-6,6,by=0.5))
```
Enter　← -6〜6 の範囲，0.5 刻みの階級

この処理により図 27 の (b) のようなヒストグラムが作成される．

注）引数「breaks=」を指定してもシステムの自動設定が優先されることが多々ある．

4.1.9　棒グラフ

関数 barplot を使用することで棒グラフを作成することができる．

書き方：　barplot(データ, names.arg=各値の名称)

「データ」の要素の値を表す棒グラフを作成する．引数「names.arg=」には棒グラフの各値の名称（棒のラベル）の並びを与える．また plot に指定できる引数の多くが barplot でも使用可能である．以下に例を示して barplot の使用方法について説明する．

例. サンプルデータの作成と棒グラフの作図

```
> v <- c(1,2,3,4,3,2,1)
```
Enter　←サンプルデータ v

```
> barplot(v,names.arg=c("a","b","c","d","e","f","g"))
```
Enter　←棒グラフの作図

この処理により図 28 のような棒グラフが表示される．

barplot による作図対象のデータには配列などを与えることもできる．例えば R システムに標準的に添付されている配列データ VADeaths があり，これを棒グラフにする処理を例示する．

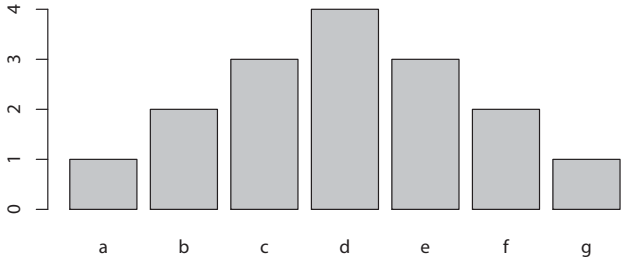


図 28: 関数 barplot による棒グラフ

例. 標準添付のデータ VADeaths

```
> data(VADeaths)  Enter      ←データセット VADeaths の読み込み[43]
> VADeaths  Enter      ←内容確認
      Rural Male Rural Female Urban Male Urban Female
50-54       11.7          8.7       15.4          8.4
55-59       18.1         11.7       24.3         13.6
60-64       26.9         20.3       37.0         19.3
65-69       41.0         30.9       54.6         35.1
70-74       66.0         54.3       71.1         50.0
```

このデータは 1940 年の米国ヴァージニア州における住民 1000 人当たりの死亡率のデータであり，年齢群（配列の行の名前），性別，居住地（農村，都市部）のクロス集計データである．例えばこのデータの第 1 列（Rural Male）は農村部の男性の年齢別の死亡率である．(次の例)

例. 上記データの第 1 列（Rural Male）(先の例の続き)

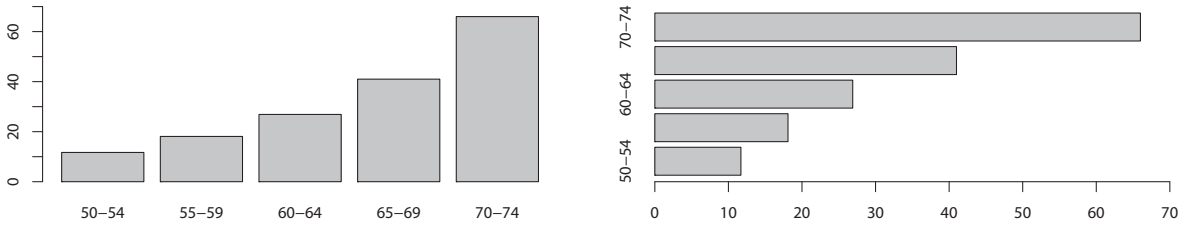
```
> VADeaths[,1]  Enter      ←第 1 列の取り出し
50-54 55-59 60-64 65-69 70-74
 11.7  18.1  26.9  41.0  66.0
```

これを barplot でプロットすると棒グラフが作成できる．この場合の各棒の名前には上記データの各値の名前が自動的に与えられる．(次の例)

例. 上記データの第 1 列（Rural Male）を棒グラフにする（先の例の続き）

```
> barplot(VADeaths[,1],ylim=c(0,70))  Enter      ←棒グラフの作図
```

この処理により図 29 の (a) のようなグラフが表示される．



(a) 縦向き（デフォルト）　　(b) 横向き（horiz=TRUE）

図 29: 配列データを棒グラフにする例

4.1.9.1 横向きの棒グラフ

barplot に引数「horiz=TRUE」を与えると図 29 の (b) のように横向きの棒グラフとなる．

例. 横向きの棒グラフ（先の例の続き）

```
> barplot(VADeaths[,1],xlim=c(0,70),horiz=TRUE)  Enter      ←横向きの棒グラフ
```

[43] この処理は省略可能な場合がある．

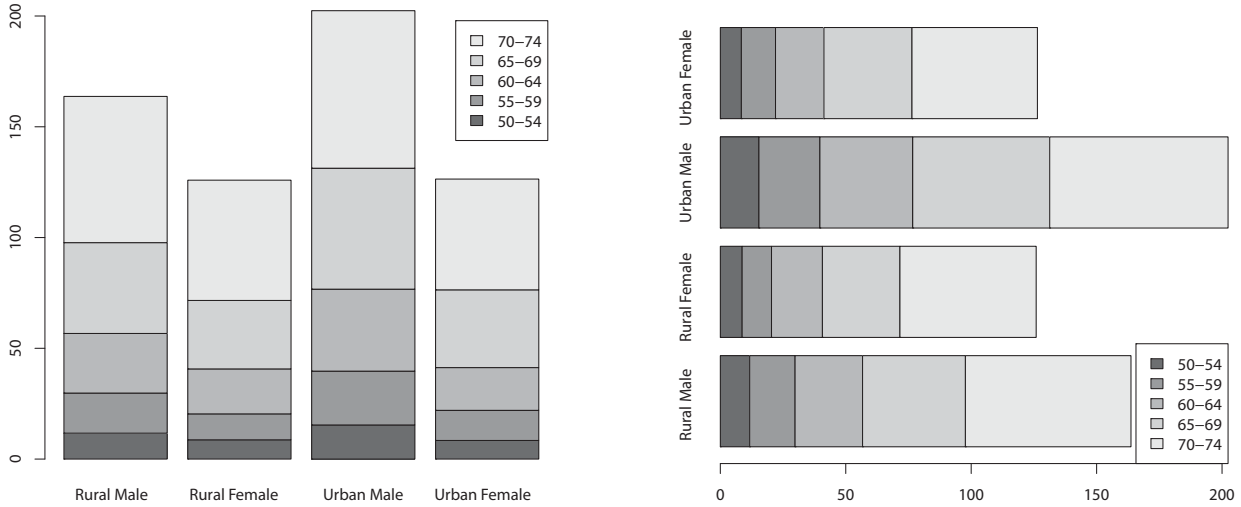
4.1.9.2　積み上げ形式の棒グラフ

barplot の第 1 引数に配列を与えると，各列毎の積み上げ形式の棒グラフが作成できる．(次の例)

例. 積み上げ形式の棒グラフ：縦向き（先の例の続き）

```
> barplot( VADeaths, legend=rownames(VADeaths) )  Enter
```

引数「legend=」には凡例の並びを与える．この例では各列毎の棒グラフを作成するので凡例は行の名前を rownames 関数で取得して与えている．この処理により図 30 の (a) のような積み上げ形式の棒グラフが表示される．



(a) 縦向き（デフォルト）　　(b) 横向き（horiz=TRUE）

図 30: 配列の各列を積み上げ形式の棒グラフにする例

barplot に引数「horiz=TRUE」を与えて横向きの積み上げ棒グラフを作成することができる．(次の例)

例. 積み上げ形式の棒グラフ：横向き（先の例の続き）

```
> barplot( VADeaths, legend=rownames(VADeaths),  Enter
+           horiz=TRUE, args.legend=list(x="bottomright") )  Enter
```

この例では引数「args.legend=list(x="bottomright")」を barplot に与えて凡例の表示位置を右下に設定している．この処理により図 30 の (b) のような横向きの積み上げ形式の棒グラフが表示される．

■ 各列のデータの内訳を横並びの棒で表示する方法

barplot に引数「beside=TRUE」を与えることで，各列の内訳（各行の値）を横並びの棒として表示することができる．(次の例)

例. 横並びの棒グラフ（先の例の続き）

```
> barplot( VADeaths, beside=TRUE, legend=rownames(VADeaths) )  Enter
```

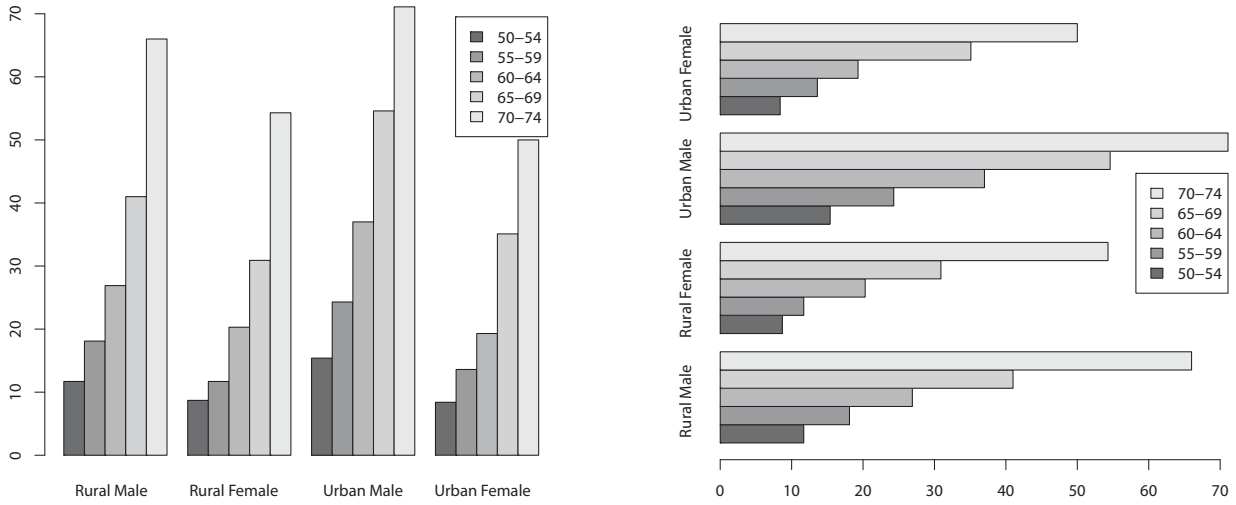
この処理により図 31 の (a) のようなグラフが表示される．

barplot に引数「horiz=TRUE」を与えてグラフを横向きにすることができる．(次の例)

例. 上の例と同じ内容のグラフを横向きに表示する（先の例の続き）

```
> barplot( VADeaths, beside=TRUE, legend=rownames(VADeaths),  Enter
+           horiz=TRUE, args.legend=list(x="right") )  Enter
```

この例では引数「args.legend=list(x="right")」を barplot に与えて凡例の表示位置を右に設定している．この処理により図 31 の (b) のような横向きの棒グラフが表示される．



(a) 縦向き（デフォルト）　　(b) 横向き（horiz=TRUE）

図 31: 各列の内訳（各行の値）を横並びにする棒グラフの例

4.1.10　円グラフ

関数 pie を使用することで円グラフを作成することができる．

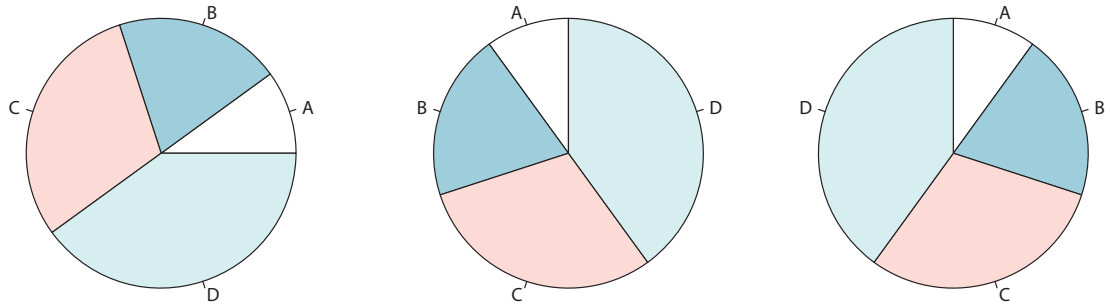
書き方：　pie(データ, labels=各値の名称)

「データ」の要素の値を表す円グラフを作成する．引数「labels=」には円グラフの各値の名称（扇形のラベル）の並びを与える．また plot に指定できる引数の多くが pie でも使用可能である．以下に例を示して pie の使用方法について説明する．

例. サンプルデータの作成と円グラフの作図

```
> d <- c(2,4,6,8)  Enter      ←サンプルデータ d
> pie(d,labels=c("A","B","C","D"))  Enter      ←円グラフの作図
```

この処理により図 32 の (a) のような円グラフが表示される．



(a) 縦向き（デフォルト）　(b) 真上から開始（init.angle=90）　(c) 時計回りに描画（clockwise=TRUE）

図 32: 円グラフの例

図 32 の (a) の円グラフからわかるように，扇形の描画の開始位置は右端で描画の方向は反時計回りである．描画の開始位置を真上にするには pie に引数「init.angle=90」を与える．(次の例)

例. 扇形の描画開始を真上にする（先の例の続き）

```
> pie(d,labels=c("A","B","C","D"),init.angle=90)  Enter
```

この処理により図 32 の (b) のような円グラフが表示される．扇形の描画の方向を時計回りにするには pie に引数「clockwise=TRUE」を与える．(次の例)

例. 時計回りの方向に描画（先の例の続き）

```
> pie(d,labels=c("A","B","C","D"),clockwise=TRUE)  Enter
```

この処理により図 32 の (c) のような円グラフが表示される．描画方向を時計回りにすると，描画開始位置が自動的に真上に設定される．

4.1.11　箱ひげ図

関数 boxplot を使用することで箱ひげ図を作成することができる．

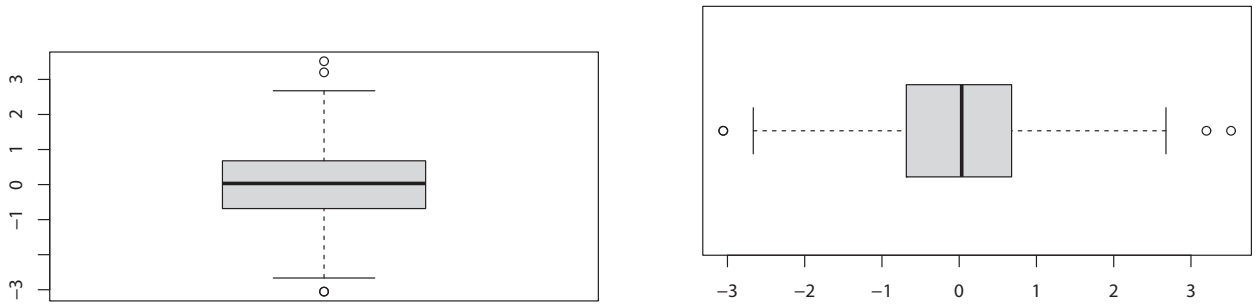
書き方：　boxplot(データ)

ベクトルや配列などの「データ」の要素の分布を表す箱ひげ図を作成する．また plot に指定できる引数のいくつかが boxplot でも使用可能である．以下に例を示して boxplot の使用方法について説明する．

例. サンプルデータ（ベクトル形式）の作成と箱ひげ図の作図

```
> d <- rnorm(1000,mean=0,sd=1)  Enter      ← μ = 0, σ = 1 の 1,000 個の正規乱数
> boxplot(d)  Enter      ←箱ひげ図の作図
```

この処理により図 33 の (a) のような箱ひげ図が表示される．



(a) 縦向き（デフォルト）　　(b) 横向き（horizontal=TRUE）

図 33: 箱ひげ図の例

boxplot に引数「horizontal=TRUE」を与えると横向きの箱ひげ図となる．(次の例)

例. 横向きの箱ひげ図（先の例の続き）

```
> boxplot(d,horizontal=TRUE)  Enter
```

この処理により図 33 の (b) のような箱ひげ図が表示される．

boxplot に配列形式のデータを与えると，各列の箱ひげ図が作成される．(次の例)

例. サンプルデータ（配列形式）の作成と箱ひげ図の作図

```
> m <- matrix(nrow=1000,ncol=5)  Enter                ← 1,000 行 5 列の配列
> colnames(m) <- c("d1","d2","d3","d4","d5")  Enter    ←列に名前を与える
> for (n in 1:5) {  Enter
+     m[,n] <- rnorm(1000,mean=(n-1)*2,sd=n/2)  Enter    ← μ, σ の異なる 5 列の正規乱数
+ }  Enter
> boxplot(m)  Enter      ←箱ひげ図の作図
```

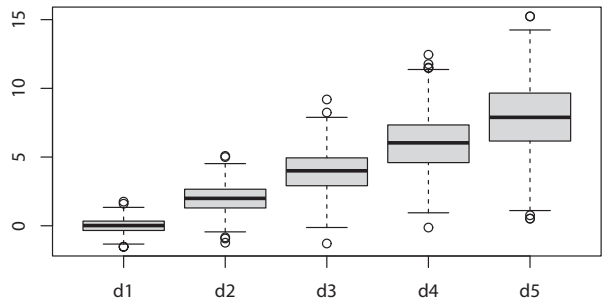
この処理により図 34 の (a) のような箱ひげ図が表示される．

グラフの横軸のラベルに配列の列名が自動的に設定されている．次に，boxplot に引数「horizontal=TRUE」を与えて横向きの箱ひげ図を作成する例を示す．

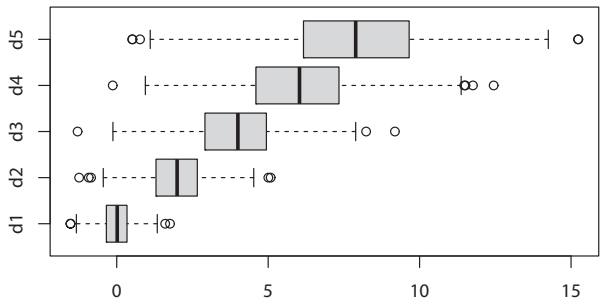
例. 横向きの箱ひげ図（先の例の続き）

```
> boxplot(m,horizontal=TRUE)  Enter
```

この処理により図 34 の (b) のような箱ひげ図が表示される．



(a) 縦向き（デフォルト）



(b) 横向き（horizontal=TRUE）

図 34: 箱ひげ図の例

4.1.12　多変数のプロット

ここでは 2 変数の値に対応する値のプロット（3 次元プロット）の方法について解説する．

4.1.12.1　ワイヤーフレーム

関数 persp を使用することで 3 次元のワイヤーフレームを作成することができる．

書き方：　persp(x 座標, y 座標, z の値, theta=回転角度 1, phi=回転角度 2)

x，y の 2 変数に対する z の値を 3 次元空間にワイヤーフレームの形でプロットする．「x 座標」と「y 座標」は 1 次元のデータの並びとして与え，それらの各要素の対に対する「z の値」を行列（matrix）の形で与える．また plot に指定できる引数のいくつかが persp でも使用可能である．以下に例を示して persp の使用方法について解説する．

次の例は $-4.5 < x < 4,5$, $-4.5 < y < 4.5$ の xy 平面の上の値 $z = \cos\left(\sqrt{x^2 + y^2}\right)$ を作成するものである．

例. サンプルデータの作成

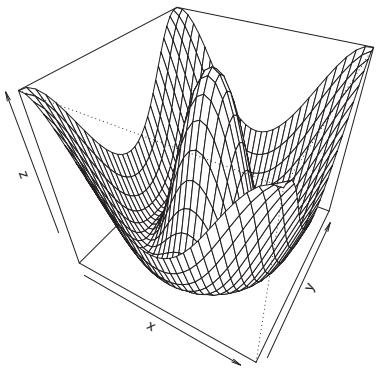
```
> x <- seq(-4.5,4.5,by=0.3); y <- seq(-4.5,4.5,by=0.3)  Enter     ← x,y の座標データ
> z <- matrix(ncol=length(x),nrow=length(y))  Enter     ← z の値を格納するための行列
> for ( yi in 1:length(y) ) {  Enter     ← z の値を算出する処理
+     for ( xi in 1:length(x) ) {  Enter
+         z[yi,xi] <- cos( sqrt(x[xi]^2+y[yi]^2) )  Enter
+     }  Enter
+ }  Enter
```

このようにして得られたデータをワイヤーフレームとして作図する例を次に示す．

例. ワイヤーフレームの作図（先の例の続き）

```
persp(x,y,z, theta=30, phi=40)  Enter
```

この処理の結果，右の図のようなワイヤーフレームが作成される．



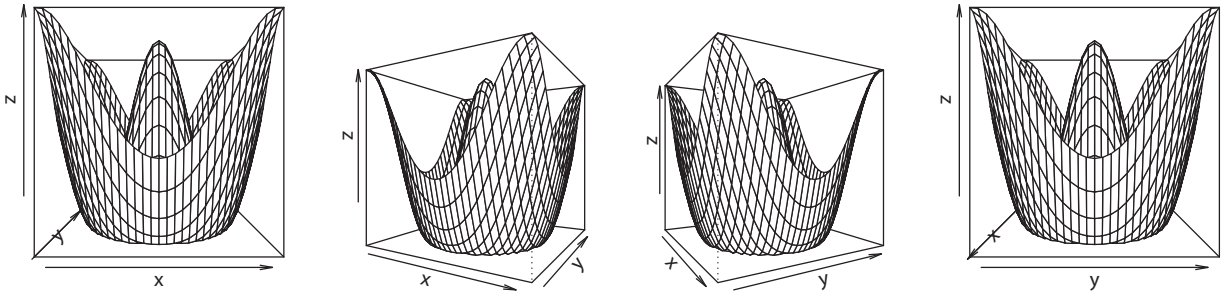
persp の引数 theta，phi の値を変化させて作図する例を示す．
図 35 は `phi=0` として `theta` を 0〜90 まで変化させて作図したもの，図 36 は `theta=0` として `phi` を 0〜90 まで変化させて作図したものである．

4.1.12.2　ヒートマップ

関数 image を使用することで 3 次元のデータから**ヒートマップ**を作成することができる．

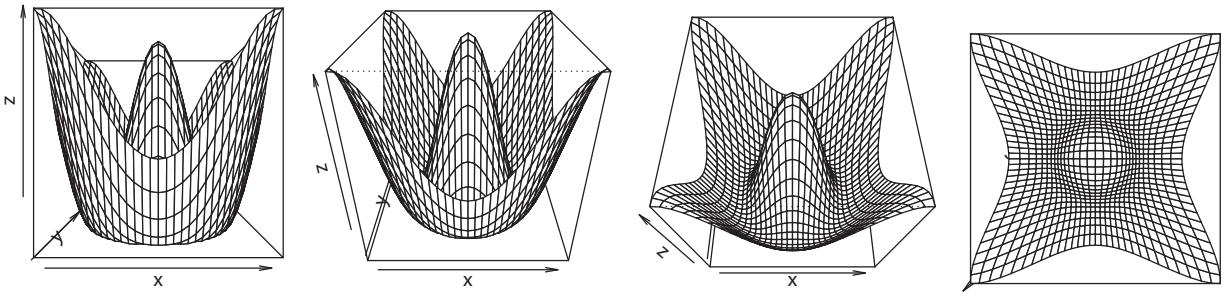
書き方：　image(x 座標, y 座標, z の値, col=カラー設定)

x，y の 2 変数に対する z の値をヒートマップとして作図する．「x 座標」と「y 座標」は 1 次元のデータの並びとして与え，それらの各要素の対に対する「z の値」を行列（matrix）の形で与える．作図対象のデータに関しては，ワイヤーフレームの場合と考え方は基本的に同じである．また plot に指定できる引数のいくつかが image でも使用可能である．以下に例を示して image の使用方法について解説する．



(a) theta=0　(b) theta=30　(c) theta=60　(d) theta=90

図 35: theta の値による回転角度の違い



(a) phi=0　(b) phi=30　(c) phi=60　(d) phi=90

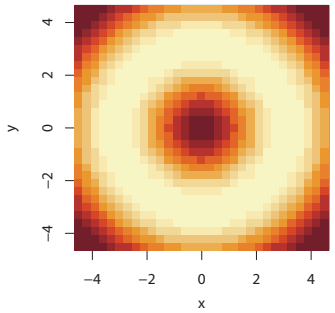
図 36: phi の値による回転角度の違い

先にワイヤーフレームの解説の際に作成したデータを用いてヒートマップ作成の例を示す.

例. ヒートマップの作成（先の例の続き）

```
> image(x,y,z)
```
Enter

この処理の結果，右の図のようなヒートマップが作成される.



この例では image に与えるカラー設定が省略されており，その場合は

```
image(x,y,z, col=hcl.colors(12,"YlOrRd",rev=TRUE))
```

を実行したのと同じ結果となる.

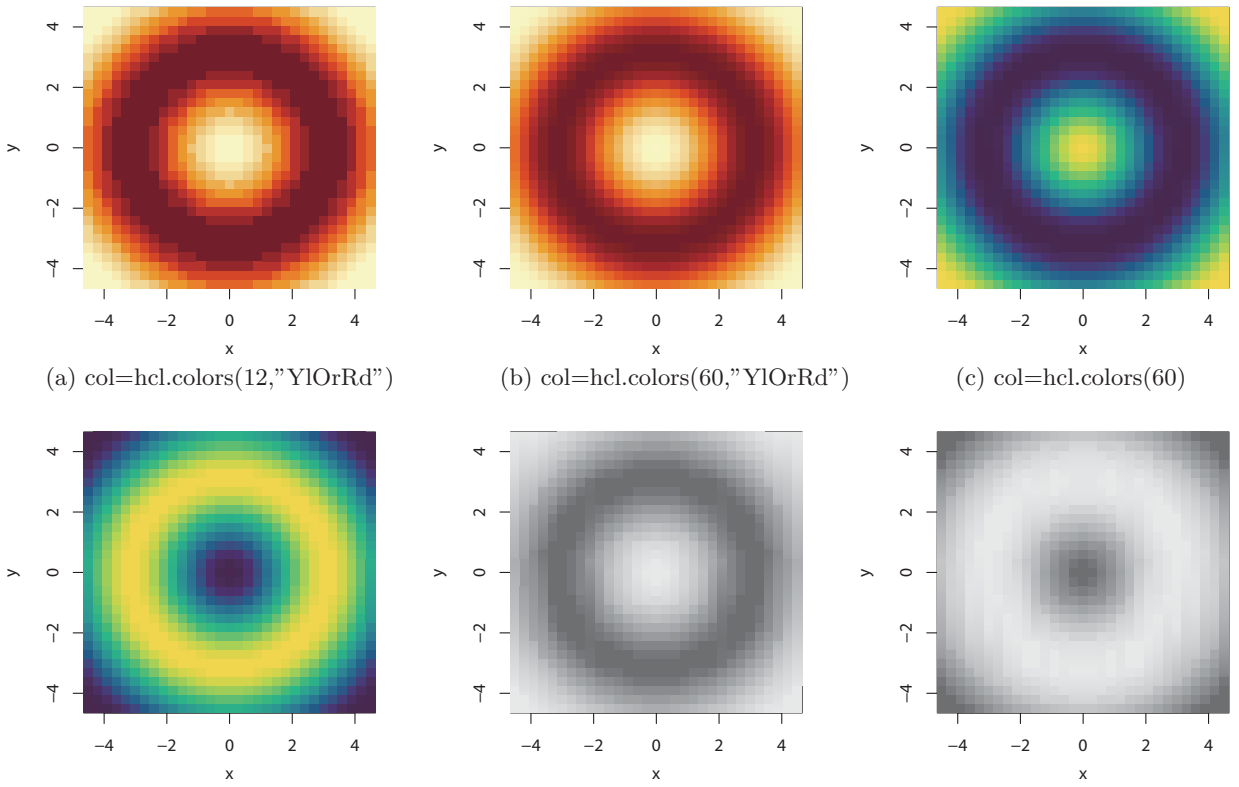
■ カラー設定（引数「col=」）

image の引数「col=」に与えるカラー設定はヒートマップを描画する際の色の表現を指定するものである．上に示したようにデフォルトでは

```
hcl.colors(12,"YlOrRd",rev=TRUE)
```

が指定される．この hcl.colors の第 1 引数には描画に使用する色数を指定し，第 2 引数にはデータの値を対応させる色表現（**パレット**：palette）を指定する．また引数「rev=」には色の順序を真理値で指定する．hcl.colors のパレット（第 2 引数）を省略するとパレットはデフォルトで "viridis" となる．引数「rev=」を省略するとデフォルトで FALSE となる

カラー設定には hcl.colors 以外にも gray.colors なども使用できる．カラー設定に様々なものを与えて image を実行する例を以下に示す.



(a) col=hcl.colors(12,”YlOrRd”)　(b) col=hcl.colors(60,”YlOrRd”)　(c) col=hcl.colors(60)

(d) col=hcl.colors(60,rev=TRUE)　(e) col=gray.colors(30)　(f) col=gray.colors(30,rev=TRUE)

図 37: 引数「col=」の設定によるヒートマップの表現の違い

付録

A　Rの入手先とインストール方法

R の処理系はソースコード，コンパイル済みインストーラ（図 38）共に公式インターネットサイト

https://www.r-project.org/

から入手することができる．

図 38: Windows 用 R 処理系のインストーラ

Windows 環境では図 38 に示すインストーラを起動することで R をインストールすることができる．図 39 に Windows 版 R のインストール作業の例を示す．

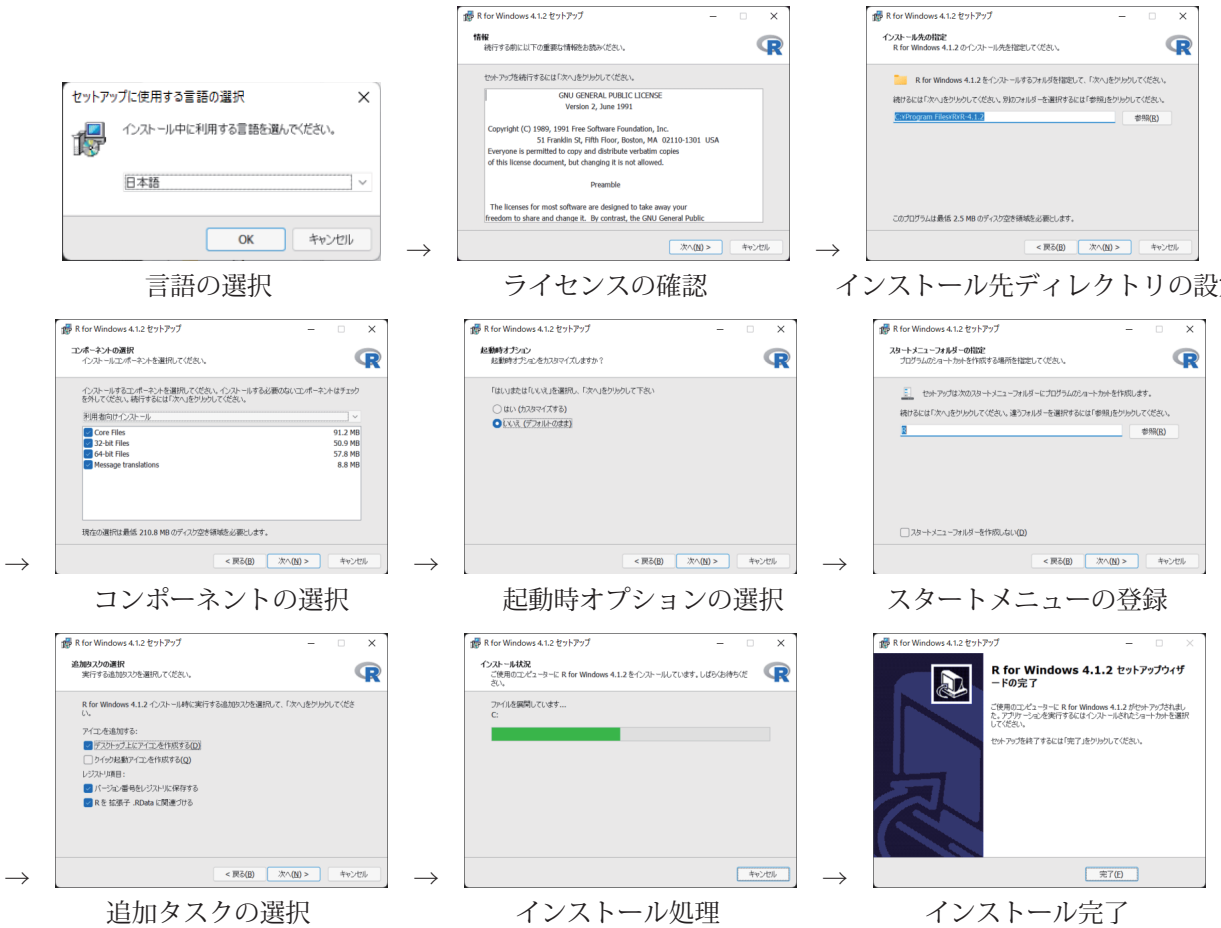


図 39: Windows 版 R のインストール作業の例

B　Rの起動に関する事柄

OS の**環境変数**「LANG」に値「ja_JP.UTF-8」を設定しておくと，ターミナル環境で表示されるメッセージが基本的に日本語となる．

参考文献

[1] R Core Team,
"R Language Definition", R Development Core Team, 2021

索 引

「GNU R プログラミング入門」

テキストの最新版と更新情報

本書の最新版と更新情報を，プログラミングに関する情報コミュニティ Qiita で配信しています.

→ https://qiita.com/KatsunoriNakamura/items/9d9fd58ca74cce73b663

上記 URL の QR コード

● 連絡先

nkatsu2012@gmail.com

中村勝則