|  |
| --- |
| 博士課程教育リーディングプログラム  実世界データ循環学リーダー人材育成プログラム |
| データツールファースト |
| R入門 |

|  |
| --- |
| 名古屋大学  2014/03/02 |

* 本テキストに掲載されている会社名、組織名、製品名は一般に各社の登録商標または商標です。

目次

[はじめに 1](#_Toc381545711)

[1. Rについて 2](#_Toc381545712)

[1.1 「R」とは 2](#_Toc381545713)

[オープンソースで提供されていること 2](#_Toc381545714)

[統計解析向けであること 2](#_Toc381545715)

[プログラミング言語およびその開発実行環境であること 3](#_Toc381545716)

[1.2 背景から見るR 4](#_Toc381545717)

[1.3 困ったときは 5](#_Toc381545718)

[help()関数 6](#_Toc381545719)

[インターネット上の情報 6](#_Toc381545720)

[掲示板やフォーラムで聞いてみる 8](#_Toc381545721)

[操作中に困ったら 8](#_Toc381545722)

[2. Rで扱うデータの姿 10](#_Toc381545723)

[2.1 データの型 10](#_Toc381545724)

[データの型とは 10](#_Toc381545725)

[データ型の種類 11](#_Toc381545726)

[データのmodeとclass 11](#_Toc381545727)

[データ型の変換 12](#_Toc381545728)

[特殊なデータ 12](#_Toc381545729)

[欠損値、ヌル値、非数値 12](#_Toc381545730)

[因子（factor） 14](#_Toc381545731)

[日付型データ 16](#_Toc381545732)

[2.2 データの構造 21](#_Toc381545733)

[ベクトル(vector) 21](#_Toc381545734)

[配列(array) 22](#_Toc381545735)

[行列(matrix) 23](#_Toc381545736)

[行列の基本操作 23](#_Toc381545737)

[行列内部での計算 24](#_Toc381545738)

[行列の計算（特殊な演算子） 24](#_Toc381545739)

[データの結合（rbind/cbind） 25](#_Toc381545740)

[データのソート・並び替え（order） 26](#_Toc381545741)

[リスト(list) 26](#_Toc381545742)

[データフレーム（data.frame） 27](#_Toc381545743)

[データフレームへのアクセス（データの指定） 28](#_Toc381545744)

[データの編集 29](#_Toc381545745)

[データの結合・併合の例（cbindとdata.frame） 30](#_Toc381545746)

[データフレームの併合（merge） 31](#_Toc381545747)

[列データを追加する（transform） 33](#_Toc381545748)

[データ構造の変換と検査 34](#_Toc381545749)

[～統計Tips 1～　変数 35](#_Toc381545750)

[～統計Tips 2～　記述統計 35](#_Toc381545751)

[3. データを整備する 39](#_Toc381545752)

[3.1 ワーキングディレクトリの設定 39](#_Toc381545753)

[3.2 データの読み込みと出力 39](#_Toc381545754)

[ベクトル・行列の読み込み（scan） 40](#_Toc381545755)

[データフレームの読み込み（read.table） 42](#_Toc381545756)

[データセットの出力 46](#_Toc381545757)

[3.3 データの概要を把握する 48](#_Toc381545758)

[読み込みデータの最初（最後）の一部を見る（head、tail） 49](#_Toc381545759)

[条件にあったデータを見る（subset、split） 50](#_Toc381545760)

[データの性質を見る（class、mode、sapply、lapply、str） 51](#_Toc381545761)

[基本統計量を見る（table、summary） 53](#_Toc381545762)

[3.4 その他のデータの整形 55](#_Toc381545763)

[データ形式の変更（table、dcast（reshapeパッケージ）） 55](#_Toc381545764)

[重複データの削除（unique、duplicated） 58](#_Toc381545765)

[欠損値の扱い方 59](#_Toc381545766)

[欠損データの除去 60](#_Toc381545767)

[欠損データの補完（replace） 62](#_Toc381545768)

[～統計Tips 3～　推測統計 63](#_Toc381545769)

[4. データ計算・処理の基礎 67](#_Toc381545770)

[4.1 数値計算と行列計算 67](#_Toc381545771)

[数値計算と演算子 67](#_Toc381545772)

[行列計算 70](#_Toc381545773)

[4.2 関数 71](#_Toc381545774)

[関数の利用 71](#_Toc381545775)

[関数について 71](#_Toc381545776)

[数学基本関数 72](#_Toc381545777)

[統計基本関数 73](#_Toc381545778)

[apply()関数群 74](#_Toc381545779)

[データの構造がベクトル、配列の場合：apply()関数 74](#_Toc381545780)

[データの構造がリストの場合：sapply()関数、lapply()関数 76](#_Toc381545781)

[データの構造がトランザクション 78](#_Toc381545782)

[パッケージの利用 79](#_Toc381545783)

[4.3 視覚化（グラフ）の基礎 80](#_Toc381545784)

[その前に（高水準グラフィックス/低水準グラフィックス） 80](#_Toc381545785)

[グラフを作成する（plot：高水準グラフィックス） 81](#_Toc381545786)

[回帰線を引いてみる（abline：低水準グラフィックス） 82](#_Toc381545787)

[グラフの種類 83](#_Toc381545788)

[タイトルをつける 84](#_Toc381545789)

[～統計Tips 4～　多変量解析 86](#_Toc381545790)

[5. プログラミングの基礎 88](#_Toc381545791)

[5.1 制御構文 88](#_Toc381545792)

[条件分岐構文（if） 88](#_Toc381545793)

[if文の構文 88](#_Toc381545794)

[ifelse()関数 90](#_Toc381545795)

[繰り返し構文 90](#_Toc381545796)

[for文 91](#_Toc381545797)

[while文、repeat文 92](#_Toc381545798)

[5.2 Rのコード編集 93](#_Toc381545799)

[Rのプログラムファイルの中身 93](#_Toc381545800)

[処理の自動化 93](#_Toc381545801)

[関数の作成 95](#_Toc381545802)

[～統計Tips 5～　機械学習とデータマイニング 97](#_Toc381545803)

[6. 参考文献 100](#_Toc381545804)

[7. 変更履歴 102](#_Toc381545805)

# はじめに

　本テキストは、Rを使った分析の基礎について、分析プロセスの各段階で必要最低限の知識を参照できることを目標として書かれています。そのため、内容としては網羅的に情報を盛り込むというよりも、特に分析者としての著者が、分析実務において最低限必要と思われる知識、手順、考え方を示すことに重点を置いています。

辞書的な意味でのRの各部各詳細の説明については、メジャーな統計解析ツールであるRだけあって、優れた他書・文献がすでに多く出ていることもあり、情報の入手方法の節（「第1章 Rについて 1.3 困ったときは」参照）にある情報ソース、および末尾の参考文献を紐解くことをお勧めします。

　本テキストが、これから分析実務に取り組む読者にとって分析プロセスの各段階で躓くことなく効率的に最終的な結果にたどり着くための一助になれば幸いです。

# Rについて

本章では、導入部として「R」および「R言語」とはどんなものかについての概要を説明しています。「R」の特徴、歴史、また問題解決の方法などについて触れています。

## 「R」とは

「R」および「R言語」（以下、単にRと表記します。）は、Wikipedia（http://ja.wikipedia.org/）によれば「R言語（あーるげんご）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言語及びその開発実行環境である。」と定義されています。

　この文章は、非常に端的に以下３点のRの特徴を表現しています。

* オープンソースで提供されていること
* 統計解析向けであること
* プログラミング言語およびその開発実行環境であること

### オープンソースで提供されていること

　オープンソースとは、アプリケーション本体そのもののプログラム（ソースコードと言います）が公開（オープンに）されているソフトウェア等を指します。オープンソースソフトウェアは、世界中の誰でもがそのソースコードをチェックすることができ、自由に利用・改変・再配布ができます。このことにより、商用ソフトウェアのような多額の投資を伴わずとも、ソフトウェアそのものの品質・信頼性が担保・向上されることになります。

　一方で、こうした品質の高いソフトウェアを「無償で自由に使用することができる」ことも重要な特徴のひとつです。ソフトウェアメーカーの恣意的な仕様変更などにより継続的な分析業務が中断してしまう、などのデメリットなどとも基本的には無縁です。

　なにより、無償でこれだけの高機能・高信頼性のあるソフトウェア・環境を利用できることは、分析者・研究者にとってはたいへんありがたいことと言えます。

### 統計解析向けであること

　統計解析やデータ分析の業務は、ある意味データとの格闘が作業の大半であると言えます。Rは、こうしたデータの処理・加工等を高速に行うことに特化したソフトウェアであると言えます。

Rでは後述するように、多様なオブジェクトを使用することができます。実数・整数・複素数・文字列・論理値などのようなデータ型に対応し、ベクトル・行列・リスト・データフレームといったデータ構造のオブジェクトを扱うことができます。

特に、Rでは独特な「ベクトル」単位でのデータ処理が可能であり、これにより複雑になりがちなデータの構造を非常に簡潔に扱うことができます。

そして、こうしたオブジェクトを高速に計算・処理するための関数群がライブラリ（必要に応じて取り込んで使うプログラムの部品）として充実している点も特筆すべきです。オープンソースとして鍛え上げられた結果の信頼性をベースに、それらの関数を簡便な記述でいつでも利用することができることがRの強みと言えるでしょう。

なお、現在Rパッケージの管理元であるCRANには、世界中のユーザーやプログラマーが開発したパッケージが約4,000件収録されています。この中には発表されたばかりの分析手法をRで利用できるものも含まれています。こうした最新の理論等を他の商用ソフトに先んじて、いち早く利用できるのもオープンソースであるRの強みです。

|  |
| --- |
| **「統計解析向け」という表現について**  Rは上述のとおり、データの処理に特化したソフトウェアであることは間違いありません。しかし近年ではその高い機能性により、用途は非常に広範に及んでいます。「統計解析」はもちろんですが、数学・物理学での「数値解析」の領域や、ビジネスで注目されている「データマイニング」「機械学習」などの領域でも、Rは活躍しています。  なお、各分野の代表的なソフトウェアには以下のものがあります。  統計解析・・・SPSS、SAS、Stata  数値解析・・・IMSL、MATLAB、GNU Octave  データマイニング・・・SPSS、SAS  それぞれに特徴がありますが、昨今は特に相互に領域横断する傾向が見られます。いずれにしても、Rは各分野においてこうしたソフトウェアと対等もしくはそれ以上の評価を受けています。 |

### プログラミング言語およびその開発実行環境であること

「統計解析向けソフト」と言われるソフトウェアには、大きく分けて「コマンドライン型」「プログラム言語型」「GUI型」の３つのタイプがあります。それぞれの特徴を以下の表にまとめました。（表1-1-1）

表1-1-1　統計解析ソフトウェアの種類

|  |  |
| --- | --- |
| 種類 | 内容 |
| コマンドライン型 | コマンド（関数と引数をコンピュータに伝える命令文）に対して、コンピュータが答えを返す対話型タイプ。 |
| プログラム言語型 | コード（プログラム）を記述して、コンパイル（コンピュータに理解できる信号に変換）して実行するタイプ。C、C++、Pythonなど。 |
| GUI型 | 主としてマウスの操作で用意された機能を実行できるタイプ。Excel、SPSS、SASなど。 |

Rは、「コマンドライン型」と「プログラム言語型」のハイブリッドということになります。コマンドラインで簡便に利用することもできますし、標準的な関数ではやりたいことができない場合やいつも同じ処理をする場合であれば、プログラムを書いておくこともできます。

　Rにも、「Rコマンダー（Rcmdr）」というパッケージがあり、ある程度のGUIでの操作を実現しています。しかしあくまでも、「Rの操作の補助」という位置づけであり、処理の結果はRの画面に出力されます。Rでの実務に慣れると、コマンドを打った方が早いと感じるようになるかもしれません。

　また、プログラム言語である、という観点からRを考えてみると、前述のCRANにあるパッケージ群を利用できるメリットは非常に大きいと言えます。プログラムをゼロから書き起こすのではなく、使えるものはCRANから取ってくることで、プログラミング作業の負担を大幅に軽減できます。

## 背景から見るR

　RはS言語の派生言語です。そこで、ここで少しS言語について紹介します。

　S言語は、1976年、アメリカ・ベル研究所でJohnChambersらによって、Fortranの統計分析環境のライブラリとして開発されました。当初は統計モデル用の関数を持っていなかったのですが、1988年にC言語で書き直されたことを機に統計モデル用の関数を実装する「S言語」としてリリースされました。その後、幾度かのバージョンアップを重ねて1998年のリリースで、ほぼ現在の姿になっています。

　リリースから現在に至るまで、関係した会社の買収等を経たものの、S言語そのものの仕様は、1998年以降ほとんど変わっていません。（1998年には、ACM（Association for Computing Machinery）のソフトウェアシステム賞を受賞）

　開発者のJohnChambersはS言語の歴史紹介サイト” Stages in the Evolution of S”　 （http://www.stat.bell-labs.com/S/history.html）で、こう言っています。

　“we wanted users to be able to begin in an interactive environment, where they did not consciously think of themselves as programming. Then as their needs became clearer and their sophistication increased, they should be able to slide gradually into programming, when the language and system aspects would become more important. This philosophy would be articulated explicitly later, but it was implicit from the start. ”

　意訳すると、「最初はユーザーにはプログラムを扱っているとは思わないようなインタラクティブな操作ではじめてもらい、徐々に彼らのニーズが明確になり内容が複雑になるのに伴って、いよいよシステムが必要だという時には、スムーズにプログラミング環境を使ってもらうことができる。それがS言語の開発当初からの暗黙の哲学でした。」となります。このことは、派生言語であるRの根底にも流れており、前述の「コマンドライン型とプログラム言語型のハイブリッド」はその表れと言えます。

　さて、翻ってRについてお話します。

　Rは1991年ニュージーランド・オークランド大学のRossIhakaとRobertGentlemanによりS言語を開発されました。その後、1995年にMartin Machler主導でRはGNU/GPLのフリーソフトとなり、翌1996年には公式メーリングリストの開始、1997年には’The R Core Group’が組織されRのソースコードを管理しています。なお、CoreGroupには、S-PLUSの開発メンバー（JohnChambersら）も参加しています。

　したがって、RはS言語の文法・構文やコマンドなどにおいて、非常に似ています。また、上述の通りその哲学（コマンドでもプログラムでも使用できる柔軟性）も受け継いでいると言えます。

## 困ったときは

本章の最後に、Rを使っていて困った時にどうするかについてご紹介します。

Rはオープンソースです。オープンソースソフトウェアは、原則的に利用した結果について無保証です。また、商用ソフトウェアのような、操作方法ややりたいことを相談できる親切な「ヘルプデスク」などはありません。オープンソースソフトウェアを使う側の正しい姿勢は、「自己責任」と「自己解決」です。

幸いなことに、Rについては公式にいくつかの解決手段が用意されています。また、ユーザーも多く、インターネットで調べれば比較的解決方法の糸口を得ることができるのも恵まれている点です。

### help()関数

　Rを使っていて困ったら、まずは用意されているドキュメントを確認しましょう。

　特定の関数の使い方や目的の関数、文法など一般的な説明を探すときに使うのがhelp関数です。

関数について調べる （例）optim：関数

|  |
| --- |
| > help(optim)  > ?optim |

引数の指定の仕方について調べる

|  |
| --- |
| > args(optim) |

文法など一般的な内容について調べる （例）Syntax：文法

|  |
| --- |
| > help(Syntax)  > ?Syntax |

予約語・特殊記号を調べる

システムで予め使う（＝ユーザーが使ってはいけない）語）や特殊記号について調べる（この場合、「” “（ダブルクォーテーション）」で囲む）

|  |
| --- |
| > help(“for”)  > ?”for” |

機能から関数と関数のヘルプドキュメントを探す。( )内は””で囲む

|  |
| --- |
| > help.search(“optimize”) |

使い方の例を調べる

|  |
| --- |
| > example(optim) |

※ただし結果の表記は英語のみとなります。ただし、インターネット上には一部の関数のhelpドキュメントを直訳したページがあります。（間瀬茂先生（東京工業大学教授）が管理運営するサイトhttp://www.is.titech.ac.jp/~mase/index-j.html）

### インターネット上の情報

　インターネット上には、Rについて書かれた膨大な情報があります。英語だけでなく、日本語の情報も多く、また自作関数のソースコードも掲載されていることも多いので、手早く解決方法を見つけることもできます。

　以下に代表的なサイトやよく利用するサイトを紹介します。

Google(https://www.google.co.jp/)

　何と言っても、困ったらまずGoogle。

WikiPedia(https://ja.wikipedia.org/wiki/)

　統計などの専門用語について、取り急ぎ概要を掴むにはWikiを見るのが早いかもしれません。ほぼ体系的に言葉の意味を知ることができます。

The R Project for Statistical Computing (http://www.r-project.org/index.html)

　Rのグローバルの公式サイトです。マニュアルやFAQ、メーリングリストの他、CRANの情報などが掲載されています。

RjpWiki (http://www.okada.jp.org/RWiki/)

　筑波大学の岡田昌史先生が運営する国内最大のR情報交換サイトです。基本的な使い方から、ハイレベルな統計処理についての議論など、たいへん興味深い情報が満載です。

R-Tips/R-Source (http://cse.naro.affrc.go.jp/takezawa/r-tips/r2.html)

　書籍「The R Tips―データ解析環境Rの基本技・グラフィックス活用集」の著者である舟尾暢男氏が作成したページです。基本的なRの使い方に関する情報が豊富です。

RとLinuxと… (http://rmecab.jp/wiki/index.php?FrontPage)

　数々のR関連、テキストマイニング関連の書籍の著者である石田基広先生（徳島大学）の備忘録サイトです。

金子邦彦研究室(http://www.kkaneko.com/index-j.html)

　九州大学の金子邦彦先生のウェブサイトです。Rに関しても非常にコンパクトに必要な情報をまとめています。なおRのページは http://www.kkaneko.com/rinkou/r/index.html

バイオスタティスティクス(http://stat.biopapyrus.net/r/)

　Rについて非常にわかりやすくまとめてあります。

　他にも、はてなブックマークなどのサイトで公開されているQ&Aや、データ分析をテーマとしているブログなど、インターネット上には数えきれないほどの情報があります。

### 掲示板やフォーラムで聞いてみる

　こちらもインターネット上の情報ですが、前項で紹介したRjpWikiやはてなブックマークなどの掲示板やフォーラムで、質問を投げるのも方法の一つです。

　ここでは、掲示板やプライベートなフォーラム等に投稿するに当たっての一般的なマナーについて紹介します。適切な答えを得るためにも、マナーを守ったコミュニケーションを心がけてください。

* 相手は質問に答える人ではありません。相手の都合、時間等を考慮しましょう。
* 相手はあなたのことを知りません。どんな状況（OSやパッケージのバージョン等）で、どのようなことがしたくて、どのようなエラーが出たのか、またこれまでにどんなことを試したけれどだめだったか、自分ではどう思うのか、など、簡潔かつ漏れのないように伝えましょう。
* 上記の方法（ドキュメントやインターネットの参照）は必ず行った上で質問しましょう。調べれば簡単にわかることを人に聞くのは失礼ですし、よい回答も期待できません。

どれも当たり前のことですが、掲示板等を見ていると、実際できていない人も多くいるようです。

### 操作中に困ったら

　とにもかくにも操作を中断したい時は、以下の方法で終了しましょう。

　学習初期によくあるのが、Rの画面で「>　コマンド」を入力し、Enterキーを押したところ、次の行に「＋　　」と表示されるだけで何も起こらない、という状態です。これは、1行目のコマンド文が完結していないため、続きを入力するようにRが催促しているのです。完了していない、とは後述する制御構文（ifやforなどを使った構文）で構文に続きがある場合や、単純にかっこ（　）の閉じる）がないなどミスにより起こる事象なのですが、ここは落ち着いてESCキーを押しましょう。

　非常に膨大なデータを扱う場合や、時間がかかる複雑な処理を行う場合などに、中断したいと思こともあります。そのときは、ctrlキー+cを同時に押します。これで処理は強制終了されます。

　どうにもならない時は、Rを終了します。Rの終了は、コマンドでq()またはquit()と入力し、Enterキーを押します。

表1-3-1　Rの終わり方

|  |  |
| --- | --- |
| 目的 | 操作 |
| 「＋　　」状態からの離脱 | ESC |
| 処理プロセスの強制終了 | ctrl + c |
| 処理プロセスの中断 | ctrl + z |
| Rの終了 | > q()　または　> quit() |

# Rで扱うデータの姿

　ここからは、実際のRの操作について述べていきます。まず本章では、Rで扱うデータ型およびデータの構造およびデータセット内の特定のデータにアクセスする方法、欠損値を含むデータの処理、データの加工の基礎について説明しています。

　なお、文中でデータの集まりのことを様々な表現で表しますが、特定のデータ構造を指す必要がない場合は、単に「データセット」と呼びます。

|  |
| --- |
| **サンプルコードの表記について**  　なお、以降の説明では、用語や標記について説明前のものも使用します。読み進めるにあたっては、以下のように理解してください。  　ベクトル：  値を持った個々のデータの集合。数値データがいくつも含まれている数列など。  　<-（代入演算子）：  　　左辺の変数に、右辺の値を代入する、という意味の表記。  　　例）x <- 2\*3 ・・・2×3の結果をxに代入するという意味 |

|  |
| --- |
| **変数の宣言**  以下のサンプルコード内では、いきなりxなどの変数が登場しています。R以外の他のプログラム言語では、通常変数のデータ型を事前に宣言する必要があるのですが、Rではこのような「変数のデータ型の宣言」をすることなく変数を使うことができます。 |

## データの型

### データの型とは

データの型とは、分析対象となるデータが、数値（実数や整数）であるか、文字列であるか、カテゴリカルデータであるかなど、データそのものの性質を表す分類です。

Rに限らず、データ処理ソフトウェアでは、データの型は非常に重要です。特に統計処理や機械学習においては、分析手法を選択する際に、データが数値であるか、文字列データであるか、カテゴリカルデータであるかという情報は非常に重要な判断材料となります。

また、処理結果がエラーなどとなる原因として、データが適切な型になっていない場合がほとんどです。

なぜデータ型を明確にしなくてはならないかというと、人が見れば同じく「１」は１ですが、コンピュータからすると、それが文字としての”１”なのか、数値としての１なのかは大変重要な情報となります。もし文字としての1であるなら、コンピュータは「1+1」を計算することができず、「数値ではないので計算できません！」というエラーを返します。

文字列を計算した例

|  |
| --- |
| > x<-"1"　　　　　　　　　　　　# 文字列の”1”を定義  > x+x　　　　　　　　　　　　　 # 文字“1”＋　文字”1”は？  Error in x + x : non-numeric argument to binary operator　　# エラーとなる |

#### データ型の種類

Rでは、以下のデータ型を扱うことができます（表2-1-1）。

表2-1-1データの種類

|  |  |  |
| --- | --- | --- |
| 種類 | 値 | 内容 |
| 文字列 | character | 数値や予約語ではない文字データ　例）”abc”、”愛知県” |
| 実数 | numeric | 実数。小数点を含む場合に使用する。 |
| 整数 | integer | 整数。小数点を含まない。 |
| 複素数 | complex | 複素数。1+1iなどの表記。 |
| 論理値 | logical | TRUE / FALSEの2値。 |

#### データのmodeとclass

データのこうした型に関する情報をmode、その状態をclassと言います。これらの情報を把握することは、適切なデータ処理を行うためには必須となります。

　Rのオブジェクトのmodeおよびclassを確認するには、mode()関数、class()関数を使います。単に値だけが存在する場合、modeもclassも同じになります。

オブジェクトのmodeとclassを調べる

|  |
| --- |
| > x<-factor(c(1,2,3,4,5,6))　　　　　　　　#　因子のベクトルを作成  > mode(x)　　　　　　　　　　　　　　　　　# modeはnumeric（実数）  [1] "numeric"  > class(x)　　　　　　　　　　　　　　　　 # classはfactor  [1] "factor" |

#### データ型の変換

よく似ているデータを他の型として扱いたい、ということも往々にしてあります。

例えば、x　<-　0 1 2 3 4 5 というベクトルがあるとするとき、このままだとinteger（整数）ですが、実はカテゴリカルデータ（因子）として扱いたい、というような場合です。

データの変換例

|  |
| --- |
| > x<-0:5  > x  [1] 0 1 2 3 4 5  > class(x)  [1] "integer"　　　　　　　　　　　　　　　　# そのままでは整数型  > x2<-as.factor(x)　　　　　　　　　　　　　 # 因子に変換※１  > class(x2)  [1] "factor" |

※１　as.factor()で変換する場合、引数にlevels等の値を設定できません

　データ型のclassを変換するのに使用する主な関数を以下の表にまとめました（表2-1-2）。

表2-1-2　データ型の変換関数

|  |  |
| --- | --- |
| 関数 | 内容　出力例：x<-c(0,1,2)とした場合 |
| as.numeric() | 実数型に変換　出力例：[1] 0 1 2 > class(x) →　[1] “numeric” |
| as.integer() | 整数型に変換　※代入時に 1L と記載しても整数となります。 |
| as.logical() | 論理型に変換　出力例：[1] FALSE TRUE TRUE |
| as.charcter() | 文字列に変換　出力例：[1] “0” “1” “2” |
| as.complex() | 複素数に変換　出力例：[1] 0+0i 1+0i 2+0i |

### 特殊なデータ

#### 欠損値、ヌル値、非数値

　データは、通常の値とは異なる特殊な値（ヌル値、欠損値、非数値など）を持つこともあります。これらの値は、読み込んだデータがもともと欠損値を含んでいたり、または計算過程で欠損したりと様々な理由で発生します。こうしたデータが多数含まれているデータセットでは、分析結果に悪い影響を及ぼしますので別途処理が必要となります。

ここでは、特殊なデータとして、ヌル値（NULL）、欠損値（NA）、非数値（NaN）について説明します。（表2-1-3）。

表2-1-3　特殊データの種類

|  |  |
| --- | --- |
| データの種類 | 内容 |
| NULL | NULL（ヌル値）は、データがない状態を指します。注意が必要なのは、NULLは「0ですらない」ということです。（0であれば値があり、これを対象とした計算が可能です）その性質から、配列などのオブジェクトの初期値として使われたり、既に設定された属性の初期化に使われるケースもあります。 |
| NA（NotAvailable） | NAは、「無意味な値」という意味で、いわゆる欠損値と言われるものです。 |
| NaN（Not a Number） | NaNは、「数値ではない値」という意味です。計算結果が数で表せない場合にNaNとなります。 |
| Inf（Infinity） | Infは「無限大」の意味です。通常0で商を求めることはできませんが、Rでは、例えば　10÷0 = Inf　となります。 |

これらの特殊なデータは、ほとんどの場合除去することになります。その場合に役立つのが、is.xxx()関数群です。対象となる値を判定し、TRUE / FALSEの論理値を返しますので、これを使ってデータの除去処理を行うことができます。

以下の例では、ベクトル内に含まれるNA値を除去する例です。

is.xxx()関数の利用例

|  |
| --- |
| > x<-c("a","b",NA,"c")　　　　　　　　　　 # 3番目にNAが含まれている  > x  [1] "a" "b" NA "c"  > is.na(x)　　　　　　　　　　　　　　　　 # 欠損値ならTRUE  [1] FALSE FALSE TRUE FALSE　　　　　　　　# 3番目がNA値であるという意味  > x[!is.na(x)] # ベクトルｘから、欠損値（NA）を除去  [1] "a" "b" "c"　　　　　　　　　　　　　　　# 「!」は否定を表す記号、[ ]は要素を指定  # する記号（サブセットで後述します） |

　上記のis.na()のように、欠損値などのデータを判定する関数として以下のものがあります（表2-1-4）。

表2-1-4　特殊データの判定関数

|  |  |
| --- | --- |
| 関数 | 内容 |
| is.null() | ヌル値ならTRUE |
| is.na() | 欠損値ならTRUE |
| is.nan() | 非数値ならTRUE |
| is.finite() | 有限値ならTRUE |
| is.infinite() | 無限大ならTRUE |
| complete.cases() | 欠損でなければTRUE |

#### 因子（factor）

因子（factor）とは、一般的にカテゴリカルデータなどと呼ばれているデータのことをRで扱う際の呼び名です。カテゴリカルデータとは、Yes/Noや都道府県名など、「要素の種類が有限（限定された）データセット」のことです。

データが集まったものをベクトル（後述）と言いますが、一般的なベクトルと因子ベクトルの違いが大きく影響してくるのは、統計モデル構築などの場面です。

Rにおいては、因子は対応する文字とともに保存された整数のベクトルです。

文字列ベクトルや数値ベクトルは、factor()関数を使って因子（factor）というclassに変換できます。多変量解析などでは、カテゴリカルデータを目的変数や入力変数として使うケースも多いので、必要に応じて数値→因子（factor）の変換を行います。

ベクトルの因子変換

|  |
| --- |
| > A<-c("yes","yes","no","yes","no")　　　　# factorを使わない例  > A  [1] "yes" "yes" "no" "yes" "no"  > class(A)  [1] "character"　　　　　　　　　　　　　　# 文字列（charcter）のまま  > table(A)  A  no yes  2 3  > B<-factor(A,levels=c("yes","no"))　　　　# Aを因子に変換。levelsで並び替え。  > B  [1] yes yes no yes no 　　　　　　　　　　# “”が取れている。=因子化  Levels: yes no　　　　　　　　　　　　　　 # levels（水準）の設定が可能  > class(B)  [1] "factor"　　　　　　　　　　　　　　　 # classがfactorに変化  > table(B)  B  yes no  3 2  > summary(A)　　　　　　　　　　　　　　　 # データの型が違うと、データの内容を調べ  Length Class Mode 　　　　　　 # るためのsummary()関数の結果も変わる  5 character character  > summary(B)  yes no  3 2 |

Rにおける因子（factor）では、データの種類のことをlevels（水準）と言います。一般的には、「ラベル（label）」などと言う場合もあります。本項では、データの種類と呼ぶことにします。

以下の例では、6個の成績データに、優・良・可・不可の4つのデータの種類が含まれています。このときデータの種類（levels）は優・良・可・不可となり、データの種類の数（nlevels()関数で調べます）は4となります。

因子（factor）のlevelsの例

|  |
| --- |
| > y<-c("優","良","可","不可","良","可")　　　　　# 元データとなるベクトル  > Y<-factor(y)　　　　　　　　　　　　　　　　　 # 因子化する  > levels(Y)　　　　　　　　　　　　　　　　　　　# 含まれるデータの種類を調べる  [1] "可" "不可" "優" "良" 　# ←　機械が認識する順番  > nlevels(Y)　　　　　　　　　　　　　　　　　　 # データの種類の数を調べる  [1] 4 |

因子データには、上述のように単にデータの種類を示すカテゴリカルデータだけではなく、比較演算をする場合などのように、その種類の間の順序情報が必要となる場合もあります。例えば優・良・可・不可などの成績データなどは、順序として優＞良＞可＞不可のような情報があると便利です。こうした順序情報を持った因子を、順序付き因子と言います。

順序付き因子の例

|  |
| --- |
| > y<-c("優","良","可","不可","良","可")　　　# 文字列ベクトル  > Y<-factor(y)　　　　　　　　　　　　　　　 # 因子化する  > table(Y)　　　　　　　　　　　　　　　　　 # 集計するとラベルがばらばら  可 不可 優 良  2 1 1 2  > Y<-factor(y,levels=c("優","良","可","不可"),ordered=T)　# levels  > summary(Y)  優 良 可 不可  1 2 2 1  > Y  [1] 優 良 可 不可 良 可  Levels: 優 < 良 < 可 < 不可 |

#### 日付型データ

　日付型のデータは、時系列での分析などにおいて重要な役割を果たします。一方で、分析に使用するためには適切な加工を伴うため、少なからずとっつきにくいイメージのあるデータの型でもあります。

　一旦、日付データが整備されさえすれば、後は数値同様、mean関数、max関数、min関数などの処理も可能になります。

日付型データへの変換を行うためには、主として次の３つの方法があります。次の表では、それぞれをどのような場面で使用するべきかについて説明します（表2-1-5）。

表2-1-5　日付型データへの変換方法の選び方

|  |  |
| --- | --- |
| 方法 | 内容 |
| as.Date() | R組み込み関数ですので、パッケージの読み込みなしで使えます。日付までのデータであればもっとも手軽に使用できますが、時間の情報を有するデータを扱う場合は適切ではありません。 |
| chronパッケージ | chronパッケージを導入してchron()関数として使用します。日付に加えて時間（分まで）を扱う場合に使用します。  ※ただしタイムゾーンは扱えません。  ※標準ではインストールされないので、次のようにインストールします。   |  | | --- | | > install.packages("chron") | |
| POSIX規格 | 日付に加えて、時間（秒まで）を扱う場合に使用します。また、データが分までであっても、タイムゾーン（日本ならJST）を使用する場合にはPOSIX規格を使用します。POSIXには2種類のclassがあります。  POSIXct：基準日（1970/1/1）からの経過秒数として保存されます。関数：as.POSIXct()  POSIXlt：以下の９つの各要素を持つリストとして保存されます。  秒（sec）  分（min）  時（hour）  日（mday）  月（mon）  年（year（値：西暦-1900））  曜日（wday（値：0=日曜日～6=土曜日））  当年の1/1からの経過日数（yday（値：0~））  サマータイム適用有無（isdst（値：あり=1/なし=0））  関数：as.POSIXlt()、strptime() |

　一般的には、元データがRに読み込まれた際に、日付として認識されるのですが、”2014/3/17”などが文字列として認識されている場合には、日付データに変換する必要があります。

文字列の日付を日付型のデータに変換する

|  |
| --- |
| > d<-"2014/3/17"　　　　　　　# “2014/3/17”という文字列  > class(d)　　　　　　　　　　# classを確認しても”character”になっている  [1] "character"  > D<-as.Date(d)　　　　　　　 # as.Date()で日付型に変換  > D  [1] "2014-03-17"　　　　　　　# Rの標準形式（年月日を”-“で区切る）になっている  > class(D)　　　　　　　　　　# classを確認すると、”Date”になっている  [1] "Date" |

　なお、例えば

140214

は、人が見ると何とか「2014-2-14」と解釈できますが、コンピュータにはどう読むのかを教えなければなりません。

日付データの変換

|  |
| --- |
| > d<-"140102"　　　　　　　　　　# 文字列としての数字  > D<-as.Date(d,format="%y%m%d")  > D  [1] "2014-01-02" |

as.Date()関数に引数formatを指定することで、元データのどの部分をどう解釈するかを指示します。上記のformat=以降の部分です（表2-1-6）。

表2-1-6　as.Date()関数で使用する日付の書式コード

|  |  |
| --- | --- |
| コード | 値 |
| %d | 日（10進数）　　 01、7、10など |
| %m | 月（10進数）　　 5、12など |
| %b | 月（英語の略語）　Jan、Feb、Marなど |
| %B | 月（英語の名称）　January、Februsryなど |
| %y | 西暦（2桁）　　　09、14など |
| %Y | 西暦（4桁）　　　2009、2014など |

なお、年月日を区切る記号として、/（スラッシュ）や-（ハイフン）がありますが、これらはそのまま記載できます。例えば、

2014-2-14　→　コードにすると、%Y-%m-%d

と指定することになります。

　元データの日付データは、上記以外にも様々な形をしています。例えば、サーバーのログなどでは、

17/mar/2014　09:15:00

のような形で日付を保管していることが多くあります。前述のとおり、秒までのデータがあり、このままas.Date()で変換することはできません。こうしたデータをRで使うための前処理についてお話します。

|  |
| --- |
| **Rのシステム設定値（locale）の変更**  　上記のような例では、POSIX規格の変換を行うのが定石ですが、そのために使用するstrptime()関数は、実は日本の標準時設定では正しく動作しません。少し突っ込んだ説明となりますが、日付の取り扱いは非常に重要なため、以下ではあえてシステム設定値の変更方法を含めて紹介します。 |

日付データの変換（システム設定値（locale）の変更含む）

|  |
| --- |
| > Sys.getlocale()　　　　　# システムのlocale設定値を確認  [1] "LC\_COLLATE=Japanese\_Japan.932;LC\_CTYPE=Japanese\_Japan.932;LC\_MONETARY=Japanese\_Japan.932;LC\_NUMERIC=C;LC\_TIME=Japanese\_Japan.932"  > Sys.setlocale("LC\_TIME","C")　　# “LC\_TIME”を”C”に変更  [1] "C"  > Sys.getlocale()  [1] "LC\_COLLATE=Japanese\_Japan.932;LC\_CTYPE=Japanese\_Japan.932;LC\_MONETARY=Japanese\_Japan.932;LC\_NUMERIC=C;LC\_TIME=C"　 # “LC\_TIME”が”C”に変わったことを確認  > d<-"17/mar/2014 09:15:00"　　　 # 日付データ（文字列の状態）  > D<-as.POSIXlt(d,format="%d/%b/%Y %H:%M:%S")　# POSIX規格の日付データに変換  > D  [1] "2014-03-17 09:15:00"  > Sys.setlocale("LC\_TIME","Japanese\_Japan.932")　# “LC\_TIME”を日本時間に戻す  [1] "Japanese\_Japan.932" |

　一旦値を格納した変数は、基本的に更新されません。したがって、システム設定値を変更した際に変数に格納した値は、システム設定値を戻した後も使用することができます。

as.Date()関数で使用したものも含め、以下にまとめます（表2-1-7）。

表2-1-7　POSIX規格での日付コード（as.POSIXlt()/as.POSIXct()/strptime()/strftime()）

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 意味 | コード | 値 | コード | 値 | コード | 値 |
| 年 | %Y | 西暦4桁 | %y | 西暦2桁 |  |  |
| 月 | %B | 英語月名 | %b | 英語月略名 | %m | 月（10進数） |
| 日 | %d | 日（10進数） | %x | 日（ロケール固有） |  |  |
| 時 | %H | 時（24時間） | %I | 時（12時間） | %X | 時（ロケール固有） |
| 分 | %M | 分（10進数） |  |  |  |  |
| 秒 | %S | 秒（10進数） |  |  |  |  |
| 時間帯 | %p | 午前・午後 |  |  |  |  |
| 週 | %W | 年間週数  （月曜開始） | %U | 年間週数  （日曜開始） |  |  |
| 曜日 | %w | 曜日（10進数） | %A | 英語週名 | %a | 英語週略名 |
| タイムゾーン | %z | GMTからの  オフセット | %Z | タイムゾーン  文字列 |  |  |

　なお、as.Date()で日付に変換した値のclassは"Date"、POSIXlt()またはstrptime()で日付に変換した値のclassは"POSIXlt" "POSIXt"、POSIXct()で日付に変換した値のclassは"POSIXct" "POSIXt"となります。以降は日付データの算術処理等を行いますが、同じclass同士でないと処理はできません。

日付/時間データの処理について、比較的多いのは日付の差を求める場合と、シーケンス（連続して並べること）の場合です。

　日付の差を求める場合ですが、実は単純に差を求めれば結果を出力します。このとき、出力された結果の値のclassは"difftime"となります。

日付の差の例

|  |
| --- |
| > D1<-as.Date("2014-3-17")  > D2<-as.Date("2014-3-19")  > D3<-D2-D1  > D3  Time difference of 2 days  > class(D3)  [1] "difftime" |

　日付/時間の差を求める手段として、difftime()関数も使うことができます。実際は上述のように差の計算を行いますが、例えば単位をdayではなく、他の単位で表示したい時などに使用します。difftime()関数は結果を「日」単位で出力しようとするため、これを明示的に制御することが必要です。

なお、使用できる時間単位は、”auto”、”secs”、”mins”、”hours”、”days”、”weeks”のいずれかとなります。

単位を変更（units）した日付の差

|  |
| --- |
| > D1<-as.Date("2013-12-17")  > D2<-as.Date("2014-3-19")  > difftime(D2,D1)　　　　　　　　　　# units（時間単位）を指定しないと、  Time difference of 92 days　　　　　 # 「日（days）」単位で出力される  > difftime(D2,D1,units="weeks")　　　# units=”weeks”を指定すると  Time difference of 13.14286 weeks　　# 「週（weeks）」単位で出力される |

　時系列分析などにおいては、日付データが連続していることが必須です。すなわち、元のデータセットにある日のレコード（記録）がなくても、時系列分析を行う際にはその日には「記録がない」というレコードを追加する必要があるわけです。

　そうした際に便利なのが、seq()関数です。seq()関数を使って、連続する日付のデータセットを作成し、元のデータセットにマージ（併合）していく方法です。具体的な使用例については、merge()関数の説明を要しますので、後述の「データの構造>データフレーム>データフレームの併合(merge)」の項で説明します。

## データの構造

データの構造とは、上述のような型を持ったデータが、「どのように集まっているのか」を定義するものです。データの構造には、ベクトル、配列、行列、リスト、データフレームがあります。

### ベクトル(vector)

　Rにおける「ベクトル」は、「同じ型をもつデータをいくつかまとめたもの」というオブジェクトを指します。これはRでの処理の基本単位です。おおよそ、「1次元の数列」を想起すれば、イメージに近いと思います。もちろん、文字列のベクトル、複素数のベクトルなども存在します。

|  |
| --- |
| 実は、データの型のセクションで、あえて説明をしていませんでしたが、ベクトル内の要素は記載の通り「すべて同一の型」であることが原則です。したがって、modeやclassでデータの型を調べた結果は、そのベクトルに含まれるデータの型の情報や状態を示しています。  　値が一つだけ（x <- 1など）の場合でも、それは要素が一つだけのベクトルと考えます。 |

　ベクトルは、１つのオブジェクトとして処理の対象となります。また、ベクトルの要素に対してアクセスするには、記号 [ ] を使います。以下に使用例のサンプルを紹介します。

ベクトルの操作

|  |
| --- |
| > x<-c(0,1,2,3,4,5) 　　　　　 # 0～5のベクトルを作る  > x  [1] 0 1 2 3 4 5  > x+1　　　　　　　　　　　　　# ベクトルの計算ではすべての要素に処理を一括して行う  [1] 1 2 3 4 5 6  > x[3] # ベクトルxの3番目の要素は2  [1] 2  > x[3]<-7　　　　　　　　　　　# 3番目の要素を7に変更（置換）  > x  [1] 0 1 7 3 4 5 |

### 配列(array)

　配列は、データを多次元で格納するデータセットです。ベクトルを多次元に拡張したものとも考えられます。Rの内部では、「ベクトルの積み重ね」として認識されていることもあり、ベクトル同様、原則的にはすべての要素が同一の型であることが必要です。

　なお、後述の行列は、2次元配列であり、配列の一種です。行列は配列の中で最も使用頻度が高いものと言えます。

配列の操作

|  |
| --- |
| > x<-array(1:12,dim=c(2,3,2))　　　　# 3次元配列を作成  > x  , , 1　　　　　　　　　　　　　　　　# 多次元配列のイメージ↓    [,1] [,2] [,3]  [1,] 1 3 5  [2,] 2 4 6  , , 2  [,1] [,2] [,3]  [1,] 7 9 11  [2,] 8 10 12  > x[2,2,1]　　　　　　　　　　　　　　# 要素の指定は、x[行,列，次元]の要領で指定  [1] 4  > apply(x,c(1,2),mean)　　　　　　　　# 1次元と2次元の各行列要素ごとの平均を計算  [,1] [,2] [,3]  [1,] 4 6 8  [2,] 5 7 9 |

### 行列(matrix)

　行列は、配列の特殊な形（2次元配列）です。配列同様、ベクトルの積み重ねとしてRは認識していますので、行列に含まれるデータはすべて同一のデータ型であることが必要です。実際のデータ分析には、行列が頻繁に使用されます。また、そのため行列処理に関しては、matrix()関数をはじめとして、多数の関数が用意されています。

|  |
| --- |
| 行列は特にプログラムを記述する際に使用されます。  「すべてのデータに対して処理をする」というプログラムを書く場合、他のプログラム言語の多くでは、for loopなどの繰り返し制御構文を利用します。  Rでは、行列をオブジェクトとして処理を記述するだけで、すべてのデータに対して処理が行われるため、繰り返し構文よりも高速に処理することができます。 |

#### 行列の基本操作

以下、サンプルを使って基本的な操作について見ていきます。

行列の操作

|  |
| --- |
| > X<-matrix(1:6,nrow=3,ncol=2)　　　　# matrix()関数で、行列を作る  > X  [,1] [,2]  [1,] 1 4  [2,] 2 5  [3,] 3 6  > X[3,2]　　　　　　　　　　　　　　　# 3行目2列目の値は6　[行番号,列番号]で指定  [1] 6  > X[,1]　　　　　　　　　　　　　　　 # 列を抜き出すこともできる  [1] 1 2 3  > X[,1,drop=F]　　　　　　　　　　　　# drop=Fは、抜き出したデータがベクトル化するの  [,1]　　　　　　　　　　　　　　 # を無効にし、行列のまま取り出すための設定  [1,] 1　　　　　　　　　　　　　　 # ※FはFALSEの省略。TはTRUE  [2,] 2  [3,] 3 |

#### 行列内部での計算

　単純に行方向の合計を出す、列方向の合計を出すなどはスプレッドシート（表計算）などでもよく行う処理です。Rには、行列内の計算について、以下のような関数が用意されています（表2-2-1）。

#2-2-1　行列内部の計算関数

|  |  |
| --- | --- |
| 関数 | 内容 |
| rowSums() | 行の総和 |
| colSums() | 列の総和 |
| rowMeans() | 行の平均 |
| colMeans() | 列の平均 |

行列内部の計算例

|  |
| --- |
| > X<-matrix(1:6,nrow=3,ncol=2)  > X  [,1] [,2]  [1,] 1 4  [2,] 2 5  [3,] 3 6  > rowSums(X)  [1] 5 7 9　　　　　　　　　# 1+4、2+5、3+6の結果  > colMeans(X)  [1] 2 5　　　　　　　　　　# 1,2,3の平均、4,5,6の平均 |

#### 行列の計算（特殊な演算子）

　行列はまた、他の行列との結合や計算など、行列同士の処理を行うことが多くあります。以下、使用例で見ていきます。

|  |
| --- |
| 注意が必要なのは、Rでは行列の計算について通常の演算子（+,-,\*,/）を使うと、「各要素ごとに計算」することです。これは、あくまでもRでの行列がベクトルの積み重ねであるためです。  通常のいわゆる「行列計算（積）」を行うためには、特殊な演算子　%\*%　を使用します。 |

行列計算（積）の例（演算子\*および%\*%の使用例）

|  |
| --- |
| > a<-matrix(1:1,nrow=2,ncol=3)　　　　　　　　　　# 2×3の行列  > b<-matrix(2:2,nrow=2,ncol=3)　　　　　　　　　　# 2×3の行列  > a\*b  [,1] [,2] [,3]　　　　　　　　　　　　　　　 # 各要素について積を計算  [1,] 2 2 2  [2,] 2 2 2  > a%\*%b　　　　　　　　　　　　　　　　　　　　　 # 行列の積　(2x3) \* (2x3)  Error in a %\*% b : non-conformable arguments　　　# 行数列数が適切でないためエラー  > a%\*%t(b)　　　　　　　　　　　　　　　　　　　　# t()により行列bを転置して再計算  [,1] [,2]　　　　　　　　　　　　　　　　　　# 行列の積　(2x3)\*(3x2)→(2x2)  [1,] 6 6  [2,] 6 6 |

#### データの結合（rbind/cbind）

　データを結合するケースは、既存のデータセットに同じ列数の新たなデータを足していく、などの分析調査上の運用ルールがある場合に発生します。Rでは、それぞれに行結合（rbind()関数）、列結合（cbind()関数）の関数が用意されています。

データの結合の例（rbind()関数 / cbind()関数の使用例）

|  |
| --- |
| > a<-matrix(1:1,nrow=2,ncol=3)  > b<-matrix(2:2,nrow=2,ncol=3)  > rab<-rbind(a,b)　　　　　　　　　　　　　　　　 # 行方向の結合  > rab  [,1] [,2] [,3]  [1,] 1 1 1  [2,] 1 1 1  [3,] 2 2 2  [4,] 2 2 2  > cab<-cbind(a,b)　　　　　　　　　　　　　　　　 # 列方向の結合  > cab  [,1] [,2] [,3] [,4] [,5] [,6]  [1,] 1 1 1 2 2 2  [2,] 1 1 1 2 2 2 |

#### データのソート・並び替え（order）

　データの昇順/降順の並び替えは非常によくある操作です。Rでは直接並び替える関数があるわけではないのですが、order()関数の値を基に並び替えを行います。

ソート（並び替え）の例（order()関数の使用例）

|  |
| --- |
| > x<-matrix(runif(12),nrow=3,ncol=4)　　　　　　　　# ランダムな値の行列を作成  > x  [,1] [,2] [,3] [,4]  [1,] 0.1155622 0.5833347 0.3628593 0.5579168  [2,] 0.5297476 0.9191317 0.4135231 0.5993930  [3,] 0.1303159 0.6570229 0.5886494 0.6658495  > order(x[,1])　　　　　　　　　　　　　　　　　　　# 1列目の大きさの順  [1] 1 3 2  > x[order(-x[,1]),]　　　　　　　　　　　　　　　　 # 1列目を降順で並び替え  [,1] [,2] [,3] [,4]  [1,] 0.5297476 0.9191317 0.4135231 0.5993930  [2,] 0.1303159 0.6570229 0.5886494 0.6658495  [3,] 0.1155622 0.5833347 0.3628593 0.5579168 |

### リスト(list)

リストは、データ型もデータセットのサイズも異なる様々なオブジェクトを、単一のRオブジェクトに保管する際に使用します。リストでの考え方・操作方法は、同じmodeのデータフレームでも基本的に共通です。実際の分析においてはデータフレーム形式のデータを主に使うことになりますので、リストの扱い方は基礎知識として押さえておくとよいでしょう。

リストの使用例

|  |
| --- |
| > x<-list(1:3,c("a","b","c"),factor(c("yes","no","yes","yes")))  > x[1]　　　　　　　　　　　　　　　　　# 数値、文字列、因子のオブジェクトを保管  [[1]]　　　　　　　　　　　　　　　　　 # x[1]はリストの要素オブジェクト  [1] 1 2 3  > x[[1]]　　　　　　　　　　　　　　　　# x[[1]]は要素オブジェクトのデータ  [1] 1 2 3  > x[[1]][2]　　　　　　　　　　　　　　 # x[[1]][2]は要素オブジェクトの特定データ  [1] 2  > x[[1]][2]<-5　　　　　　　　　　　　　# リストの要素オブジェクトの特定データを置換  > x[[1]]  [1] 1 5 3  > mode(x[[1]])　　　　　　　　　　　　　# なおリストに保管しても元のmodeは変更しない  [1] "numeric"  > mode(x[[2]])  [1] "character" |

### データフレーム（data.frame）

　データフレームは、最も身近に感じるデータセットの構造です。名簿や企業のリスト、アンケートデータなど、縦にレコード、横に項目といった形でデータを収録しており、実際の分析対象データに一番近い状態と言えます。

上記のようなデータを想起するとわかるのですが、基本的に「すべて同一のデータ型」にはならないケースがほとんどです。列ごとには、同じ型のデータが並びますが、例えば性別は因子、年齢は数値、住所や氏名は文字列など、まったく異なるデータの型を内包しています。すなわち、データフレームは、リストの一態様と言えます。リストと異なるのは、各データの長さ（大きさ）がすべて一致している点です。データフレームのmodeはlistですが、classはdata.frameとなります。また、各列にラベルを持っていることも、他のデータ構造と異なる点です。

　データの構造としては、縦方向のベクトルが横に並んでいる、というのがデータフレームの本質です。つまり、列の一つ一つがベクトルになっているわけです。以降の処理でも、縦方向の列を取り出して扱うことが多くなります。

　実際に、Rで分析を始めるためにデータを読み込む場合、大半がこのデータフレームとして読み込むことになります。

本項で使用するサンプルのデータフレームxについて、以下のように設定します。本項内においては、xはこのデータフレームを指すものとします。

サンプル用データフレームの作成

|  |
| --- |
| > gender<-factor(c("F","F","M","M","M"),levels=c("M","F"))　# 因子ベクトル  > heights<-c(170,156,175,168,178)　　　　　　　　　　　　　 # 実数ベクトル  > weights<-c(54,45,75,68,74)　　　　　　　　　　　　　　　　# 実数ベクトル  > x<-data.frame(g=gender,h=heights,w=weights)　　　　 　# データフレーム化  > x  　　g h w  1 F 170 54  2 F 156 45  3 M 175 75  4 M 168 68  5 M 178 74  > summary(x)　　　　　　　　　　　　　　　　　　　　　　　　# summary()関数で要約  g h w 　　　　 　　　　　　　# 列ごとで要約されている  M:3 Min. :156.0 Min. :45.0  F:2 1st Qu.:168.0 1st Qu.:54.0  Median :170.0 Median :68.0  Mean :169.4 Mean :63.2  3rd Qu.:175.0 3rd Qu.:74.0  Max. :178.0 Max. :75.0 |

#### データフレームへのアクセス（データの指定）

　データフレームの各要素にアクセスするには以下のようにします。なお、抜き出したデータの集合も、mode=リスト、class=データフレームの属性を持ちます。

データフレームでのデータ指定

|  |
| --- |
| > x$g 　　　　　　　　　　　　# 「データフレーム名$列名」のように列を指定  [1] F F M M M # この場合ベクトルでデータが返る  Levels: M F  > x["h"]　　　　　　　　　　　　　# []を使うと縦行列のままデータを取り出す  h  1 170  2 156  3 175  4 168  5 178  > x[["h"]]　　　　　　　　　　　　# [[ ]]はリストの要素抜出。ここでは列を指定  [1] 170 156 175 168 178　　　　　 # 「$」を使った場合と同じ  > x[3,2]　　　　　　　　　　　　　# x[行番号、列番号]で特定の値を取り出す  [1] 175  > x[x$g=="F"&x$w>50,] 　　　　# x[条件の指定,]で、条件に合致するレコードのみを抽  g h w　　　　　　　　　 # 出する　※行番号側に条件指定でレコード抽出  1 F 170 54 |

　通常、データフレームの列データにアクセスするには、上記のとおり$マークを使って、「x$g」などのようにしますが、データフレームの名前が長い場合など、都度データフレーム名を入力することが煩雑に思うこともあります。

　その場合には、attach()関数を使用して、列名だけでアクセスできるようにしてから作業をします。また、作業の終了や別のデータフレームを使う場合など、attach()関数を解除するには、detach()関数を使います。

attach()関数とdetach()関数の使用例

|  |
| --- |
| > sample.data.frame<-data.frame(g=gender,h=heights,w=weights)　# 長いデータフレーム名  > attach(sample.data.frame)  > h　　　　　　　　　　　　　　　　　　　　　　　　# 列名hだけでアクセス可能  [1] 170 156 175 168 178  > detach(sample.data.frame)　　　　　　　　　　　　# attach解除  > h  Error: object 'h' not found　　　　　　　　　　　　# エラーとなる |

#### データの編集

データの編集について、個別のデータの編集から、行一括、列一括または条件に合致するデータの一括編集も行うことができます。

データ編集の例

|  |
| --- |
| > x[3,1]<-"F"　　　　　　　　　　 # 3行目2列目の1つの値だけを編集  > x  g h w  1 F 170 54  2 F 156 45  3 F 175 75  4 M 168 68  5 M 178 74  > x[x$g=="M",][,1]<-"F"　　　　　 # 条件に合致するものを一括編集  > x　　　　　　　　　　　　　　　 # gがMのレコードの1列目をFに編集  g h w  1 F 170 54  2 F 156 45  3 F 175 75  4 F 168 68  5 F 178 74 |

行または列の削除

|  |
| --- |
| > x<-x[-3,]　　　　　　　　　　　　# 3行目の削除　Rでは「-3」の表記で行削除できる  > x  g h w  1 F 170 54  2 F 156 45  4 M 168 68  5 M 178 74  > x<-x[,-2]　　　　　　　　　　　　# 2列目(h)の削除  > x  g w  1 F 54  2 F 45  4 M 68  5 M 74  > x$g<-NULL　　　　　　　　　　　　# 列gの削除　列の削除は$で列名を指定しNULLを代入  > x　　　　　　　　　　　　　　　　# しても可能  w  1 54  2 45  4 68  5 74 |

#### データの結合・併合の例（cbindとdata.frame）

行列で使用したrbind()、cbind()はデータフレームでも利用できます。cbind()と同じく横方向に結合する関数としてdata.frame()も使えます。ただし、結合するそれぞれのデータフレームに同じ列名のデータがある場合に、data.frame()では自動的に「.1」などの連番がつきます。

※ただし、data.frameを使用しても、cbind()を使用しても、結合されたデータはmode=リスト、class=データフレームとなります。

データフレームの結合例

|  |
| --- |
| > I<-c(1,2,3,4,5)　　　　　　　　　　　　# ID用のベクトルを作る  > A<-data.frame(ID=I,H=heights)　　　　　# IDとHのデータフレーム  > B<-data.frame(ID=I,W=weights)　　　　 # IDとWのデータフレーム  > data.frame(A,B)　　　　　　　　　　　 # ここでは、横結合関数としてのdata.frame  ID H ID.1 W　　　　　　　　　　　　# 3列目の列名がID.1になっている  1 1 170 1 54  2 2 156 2 45  3 3 175 3 75  4 4 168 4 68  5 5 178 5 74  > cbind(A,B)　　　　　　　　　　　　　　# cbindで同じことをする  ID H ID W　　　　　　　　　　　　　# 3列目の列名はID  1 1 170 1 54　　　　　　　　　　　　　# データフレームでは、同じ列名は禁止されてい  2 2 156 2 45　　　　　　　　　　　　　# るため、このデータフレームは信頼性に乏しい  3 3 175 3 75  4 4 168 4 68  5 5 178 5 74 |

#### データフレームの併合（merge）

　実際によく使うのはこの併合（merge）です。データフレームのある列のデータをキーとして、２つ以上のデータを併合します。引数allでは、T（内部結合：一致するデータだけにする）、F（完全外部結合：一致しないデータもすべて保持する）を指定することができます。

データの併合(merge)の例

|  |
| --- |
| > A<-A[order(-A$ID),]　　　　　　　　# 単純な横結合ではないことを見るためデータフレー  > A　　　　　　　　　　　　　　　　　# ムAの並び方を変える  ID H  5 5 178  4 4 168  3 3 175  2 2 156  1 1 170  > AB<-merge(A,B,all=T)　　　　　　　 # この状態で、merge()を使う  > AB　　　　　　　　　　　　　　　　 # all=Tはキーの値が一致しないものもデータとして  ID H W　　　　　　　　　　　　　# 残すという設定。=Fとすれば、一致したデータのみ  1 1 170 54　　　　　　　　　　　　　# を含むデータフレームが出力される  2 2 156 45  3 3 175 75  4 4 168 68  5 5 178 74 |

　日付型データのセクションで、時系列分析を行うためには、日付に抜けがない状態を作る必要があり、そのためにseq()関数とmerge()関数を使うことについて触れましたが、ここまででmerge()までの説明が終わりましたので、実際に日付データに歯抜けがあるデータセットを、シーケンスで連続する日付を持つデータセットに加工する例を紹介します。

時系列分析の事前準備：レコードのない日付のデータを補う例

|  |
| --- |
| > day<-c("2014-2-1","2014-2-3","2014-2-5")  > day<-as.Date(day)  > products<-c("dress","juery","cosmetics")  > products<-factor(products)  > numbers<-c(10,3,24)  > DF<-data.frame(day,products,numbers)  > DF　　　　　　　　　　　　　　　　　　　　　# データフレームDFは日付に抜けがある  day products numbers　　　　　　　　# ため、時系列分析では使えない  1 2014-02-01 dress 10  2 2014-02-03 juery 3  3 2014-02-05 cosmetics 24  > sday<-seq(as.Date("2014-2-1"),by="days",length=5)　# seq()関数で2014/2/1から5日  > SDAY<-data.frame(day=sday)　　　　　　　　　　　　 # 間の連続する日のデータフレーム  　　　　　　　　　　　　　　　　　　　　　　　　　　 # を用意する  > MDF<-merge(DF,SDAY,by="day",all=T)　　　　　　　　 # DFとSDAYをdayをキーに併合  > MDF　　　　　　　　　　　　　　　　　　　　　　　　# この時、データがなくてもすべ  day products numbers　　　　　　　　　　　 # て保持するため引数：all=TRUE  1 2014-02-01 dress 10　　　　　　　　　　　 # を指定していることに注意  2 2014-02-02 <NA> NA  3 2014-02-03 juery 3  4 2014-02-04 <NA> NA  5 2014-02-05 cosmetics 24 |

#### 列データを追加する（transform）

元データだけでなく、分析過程においては途中の結果を保持しておきたくなることもあります。単に列を追加するdata.frame()やキーでの併合（merge()）とは用途が異なります。transform()関数を使った列の追加について見ていきます。

列を追加する

|  |
| --- |
| > AB  ID H W  1 1 170 54  2 2 156 45  3 3 175 75  4 4 168 68  5 5 178 74  > transform(AB,feets=AB$H/3.3)　　　# 列を追加して身長をフィートに変更した値を保持  ID H W feets  1 1 170 54 51.51515  2 2 156 45 47.27273  3 3 175 75 53.03030  4 4 168 68 50.90909  5 5 178 74 53.93939 |

　実は、さらに簡単に列を追加する方法があります。データフレームに対して新たな列名を指定する方法です。実際に例を見てみます。

列を追加する②

|  |
| --- |
| > AB　　　　　　　　　　　　# データフレームABの中身を確認  ID H W  1 1 170 54  2 2 156 45  3 3 175 75  4 4 168 68  5 5 178 74  > AB$feets<-AB$H/3.3　　　　# 新たな列feetsを指定。  > AB  ID H W feets  1 1 170 54 51.51515  2 2 156 45 47.27273  3 3 175 75 53.03030  4 4 168 68 50.90909  5 5 178 74 53.93939 |

### データ構造の変換と検査

　2.1 データの型>データの型とは>データの変換で説明したのと同じように、データ構造についてもas.xxx()関数があります。また、データの構造を検査するis.xxx()関数も用意されています。以下、主なものを紹介します。

表2-2-2　データ構造に関する主な変換関数

|  |  |
| --- | --- |
| 関数 | 説明 |
| as.vector() | ベクトルに変換 |
| as.matrix() | 行列に変換 |
| as.array() | 配列に変換 |
| as.list() | リストに変換 |
| as.data.frame() | データフレームに変換 |
| as.factor() | 因子に変換 |
| as.ordered() | 順序付き因子に変換 |

表2-2-3　データ構造に関する主な論理関数

|  |  |
| --- | --- |
| 関数 | 説明 |
| is.vector() | ベクトルならTRUE、そうでなければFALSEを返す |
| is.matrix() | 行列ならTRUE、そうでなければFALSEを返す |
| is.array() | 配列ならTRUE、そうでなければFALSEを返す |
| is.list() | リストならTRUE、そうでなければFALSEを返す |
| is.data.frame() | データフレームならTRUE、そうでなければFALSEを返す |
| is.factor() | 因子ならTRUE、そうでなければFALSEを返す |
| is.ordered() | 順序付き因子ならTRUE、そうでなければFALSEを返す |

　データの型や構造、およびその他R上のオブジェクトについて、変換するにはas.xxx()関数、論理値のデータセットを得たいのであれば、is.xxx()関数を使う、と考えてください。もちろん使用前にそうした関数が存在するか、また使用方法について調べることが必要です。

## ～統計Tips 1～　変数

|  |
| --- |
| 統計学の上でも値の性質は重要な事項です。統計学ではこれまで見てきたようなcharacterやlogicalなどの値（因子や日付含む）を質的変数、特にlogicalや因子が２つ（yes/no、男性/女性など）の変数を二値変数といいます。  一方で、numericやinteger、complexなどのデータを量的変数といいます。 |

## ～統計Tips 2～　記述統計

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **１変数の記述統計**  　一つのベクトル（変数のこと）に含まれるデータの基本統計量・代表値には次のものがあります。  代表値   |  |  |  | | --- | --- | --- | | 記述統計量 | 説明 | コマンド | | 平均 | 全ての値の和　/　値の件数 | mean() | | 最大値/  最小値 | 最大値：最も大きい値、  最小値：最も小さい値 | max()  min() | | 中央値 | 値を順に並べた場合の中央の値。値の個数が偶数の場合は、中央に該当する２つの値の平均とされる。  注意：範囲の真ん中ではない。1,2,100なら中央値は2。 | median() | | 四分位 | 最小値、第1四分位、中央値、第3四分位、最大値の５つ。  値があれば、該当の値　1,2,10,100,1000なら、  min.:1 1stQu:2 median:10 3rdQu:100 max:1000  値がなければ、各区間の等分　1,2なら  min.:1.00 1stQu:1.25 median:1.50 3rdQu:1.75 max:2.00 | quantile() | | 最頻値 | 最も多く出現する値　質的変数の代表値として使われる | max(table()) |   散布度   |  |  |  | | --- | --- | --- | | 記述統計量 | 説明 | コマンド | | 不偏分散 | 値と平均の差の2乗の総和を、データ個数－１で割ったもの。  分散（不偏分散）=  {(x1-mean(x)) 2+(x2-mean(x)) 2+...+(xn-mean(x))2}/(n-1) | var() | | 標準偏差 | 分散の平方根 | sd() | | 範囲 | 最大値と最小値  またはその差：最大値－最小値 | range()  max()-min() |   これらの基本統計量は、「そのデータセットをもっともよく表す値」として、データを把握するのに必要な情報です。さらに、代表値と散布度を使うのは、「データの中心や代表する点」と「データの範囲・ばらつき」を知ることで、データの概要を把握しようと考えているのです。量的変数に関して、平均と分散/標準偏差が重要、というのはこういうことなのです。  **2変数の記述統計**  　２つのベクトルの「関係性」をどう見るか、というのがここでのテーマです。  　細かいことを言うと、量的変数同士の関係を「相関」、質的変数同士の関係を「連関」と言います。  １．相関について  　２変数の関係性は以下の３つのパターンがあります。   1. 一方が大きいほど、他方も大きい　...　正の相関 2. 一方が大きいほど、他方が小さい　...　負の相関 3. 両方の大小に特に関係性がない　　...　無相関   　こうした２変数の関係の様子をざっとつかむのには散布図という視覚化が便利です。   |  |  | | --- | --- | |  | 正の相関  　散布図が右肩上がり | |  | 負の相関  　散布図が右肩下がり | |  | 無相関  　散布図に関係性が見られない |   　中学・高校では、方眼紙に一生懸命点を書き込んでいましたが、Rならplot(x,y)と入力するだけです。  この２変数の関係を１つの指標で表したものが、共分散と相関係数です。  共分散は、「各変数の平均との差」を個々のセットごとに乗算した値（積）の総和です。  ・・・わかりにくいので、ベクトルxとyを使った式で説明します。  　なお、Rで表現すると、sum(x-mean(x))\*(y-mean(y))/(length(x)-1)ですが、関数cov(x,y)でも同じ結果となります。  ただし、共分散では、各変数の単位が変わる（m→cmなど）と、値に大きく影響します。この問題について、各変数の値がどうあれ、変数同士の関係性の強さを示す指標として用意されたのが相関係数です。  　これもRであれば、cor(x,y)でOKです。  　なお、相関係数rは、－１～１までの値を取ります。値が正なら正の相関、負なら負の相関、０なら無相関ということです。ここで問題なのは、「どのあたりの数値だと関係性の強弱が言えるか」ということですが、厳密な決まりはありません。分析対象や分野に応じて判断すべきでしょう。（私は強いか弱いかで言ったら、という言い方をよくします。-1~-0.5または0.5~1なら強い関係性がありそう、と言っています。）  ２．連関について  　質的変数の関係性を表すものとして、代表的なのはクロス集計表です。行方向×列方向にそれぞれ変数の要素を並べ、度数を調べます。Rではtable()関数を使って作成することができます。以下に例を示します。  table()関数の使用例   |  | | --- | | > Windows<-c("好き","好き","嫌い","好き","嫌い","好き","好き","嫌い","嫌い","好き")  > Mac<-c("好き","嫌い","好き","好き","好き","嫌い","好き","好き","嫌い","嫌い")  > table(Windows,Mac)  Mac  Windows 嫌い 好き  嫌い 1 3  好き 3 3 |   　さて、連関においても相関のように一つの指標で関係性を表すことはできないでしょうか。実は疑似的に、係数を求めることができます。これがファイ係数です。  　ファイ係数は、上記のように「好き」「嫌い」の2種類の因子のみを持つデータ（2値変数）の値を1/0の値に変換し、これを相関係数と同じ計算方法で出した結果です。  ファイ係数の算出   |  | | --- | | > Windows<-ifelse(Windows=="好き",1,0)  > Mac<-ifelse(Mac=="好き",1,0)  > Windows  [1] 1 1 0 1 0 1 1 0 0 1  > Mac  [1] 1 0 1 1 1 0 1 1 0 0  > cor(Windows,Mac)  [1] -0.25 |   　もちろん、因子が3以上あるときに、0,1,2などとしてしまうと、おかしなことになります。序数と基数の違いを意識してファイ係数を使う必要があります。 |

# データを整備する

　本章では、分析処理に進む前の最終段階として、データの整備について説明します。実践的なシーンを想定して、分析プロジェクトのデータ保管場所の設定、サンプルデータセットを取り込むための手順、取り込んだデータの要約、データの整形手法と欠損値の処理について記載していきます。

## ワーキングディレクトリの設定

　ワーキングディレクトリ（作業ディレクトリ：Windowsでの、いわゆるフォルダのこと）の設定は、次の２つの点で重要です。１つは、分析対象となるデータセットのファイルを保管する場所の固定、もう一つは、分析プロセスで生成したRファイルやコード（プログラム）を記載した.Rファイルの保管場所の固定です。「固定」と言ったのは、他のプログラミング言語同様、Rでもファイルへのアクセスはパス（path：特定のファイルまでの辿り方）を使って行います。例えば、自作関数test.Rを実行する際に、ワーキングディレクトリの概念がなければ、実行時にはいちいちパスを指定しなくてはならなくなるでしょう。

　ワーキングディレクトリの設定は非常に簡単です。getwd()で確認して、setwd(“作業ディレクトリのパス”)でパスを設定します。

ワーキングディレクトリの確認と設定（フォルダWDからフォルダMyWDに変更）

|  |
| --- |
| > getwd()　　　　　　　　　　　　　　　　　# 現在のワーキングディレクトリを確認  [1] "C:/Users/*UserHome*/Desktop/WD"  > setwd("C:/Users/ *UserHome* /Desktop/MyWD")　# ワーキングディレクトリを設定  > getwd()  [1] "C:/Users/ *UserHome* /Desktop/MyWD"　# 変更後のワーキングディレクトリを確認 |

## データの読み込みと出力

　ここまで、説明のためのデータセットは、その内容を示すためにすべて都度作成してきましたが、実務では大半はRの外部のデータセットを読み込んで分析を行うことになります。本項では、Rの外部データの取り込み方法について説明します。

　また、分析の結果生成されたデータセットを、配布可能な状態にするために結果データの出力についても触れていきます。

### ベクトル・行列の読み込み（scan）

　scan()関数は、基本的にベクトルや行列など、同じ型のデータセットを読み込む際に使用しますが、リストを読み込むこともできます。センサーデータなどの比較的大きく、形の整ったデータセットを取り込む際に使用されます。

scan()関数の単純な利用例

|  |  |
| --- | --- |
| sample1.txt   |  | | --- | | 54 22 35 44  21 40 23 60  11 29 56 24 |   > data1<-scan("sample1.txt") # sample1.txtを読み込んでdata1に代入  Read 12 items  > data1  [1] 54 22 35 44 21 40 23 60 11 29 56 24　# ベクトルとしてデータを保持 |

異なるデータ型を含むデータセットでのscan()関数利用例

|  |  |  |
| --- | --- | --- |
| sample2.txt   |  | | --- | | 2014-2-1,A,200  2014-2-1,B,300  2014-2-2,A,400  2014-2-2,C,200  2014-2-3,B,100 |   > data2<-scan("sample2.txt",what=list(date="",product="",number=0),sep=",")   |  | | --- | | 解説  　上記では、what=の引数で、   1. 元データをリストとして取り込むこと、 2. リスト要素名をdate,product,numberとすること、 3. それぞれ文字列、文字列、数値であること（文字列は""、数値は0と指定）   を設定していしている。  また、sample2.txtでは、データ間の区切り文字がカンマ","であるため、sep=","としてRに区切り文字を認識させている。 |   Read 5 records　　　　　　　　　　 # 引数については後述  > data2　　　　　　　　　　　　　　# リストとして読み込んでいる  $date  [1] "2014-2-1" "2014-2-1" "2014-2-2" "2014-2-2" "2014-2-3"  $product  [1] "A" "B" "A" "C" "B"  $number  [1] 200 300 400 200 100  > Data2<-data.frame(data2)　　　　　# リストの要素を結合してデータフレームに変換  > Data2  date product number  1 2014-2-1 A 200  2 2014-2-1 B 300  3 2014-2-2 A 400  4 2014-2-2 C 200  5 2014-2-3 B 100  > class(Data2$date) # 日付が因子として認識されている  [1] "factor"  > Data2$date<-as.Date(Data2$date)　 # 日付型データに変換  > class(Data2$date)  [1] "Date" |

　scan()関数の設定（引数）について、主なものを以下の表にまとめます（表3-2-1）。

表3-2-1　scan()関数の主な引数

|  |  |
| --- | --- |
| 引数 | 説明 |
| file | 読み込むデータファイルの指定。  設定例１：  ,file="C:/Users/ UserHome /Desktop/MyWD/sample.txt"  なお、ワーキングディレクトリ設定ができていれば、単に  設定例２：  ,file=”sample.txt”  とすればよい。さらに、どちらの例でもfile=は省略可能。  　なお、fileを指定しな場合は、キーボードからの入力モードとなり、空行を入力（実際にはEnterキーを2回）することで、入力終了となる。 |
| what | 読み込むデータ列の型を指定。型の異なるデータを読み込むには、上の例のようにリストとして指定する。  設定例１：数値データのみの場合、  無設定または,what=0  ※文字列：””、数値：0を指定。  設定例２：異なる方を含む場合、  　,what=list(date="",product="",number=0)　文字列、文字列、数値の設定 |
| sep | 区切り文字の指定。カンマ”,”、タブ"\t"（R上は￥はバックスラッシュ）が一般的。  設定例：  ,seq=”,” |
| skip | データを読み込む際に読み飛ばす行の指定。例えば欠損値を含むデータをデータ読み込み時に除外するときなどに使用する。 |
| nlines | 読み込むデータの最大行数を指定。 |
| na.strings | データに含まれる独自の欠損値表示をRに伝える。  設定例：データファイル上の欠損値表記=N/Aの場合、  　,na.strings=”N/A” |

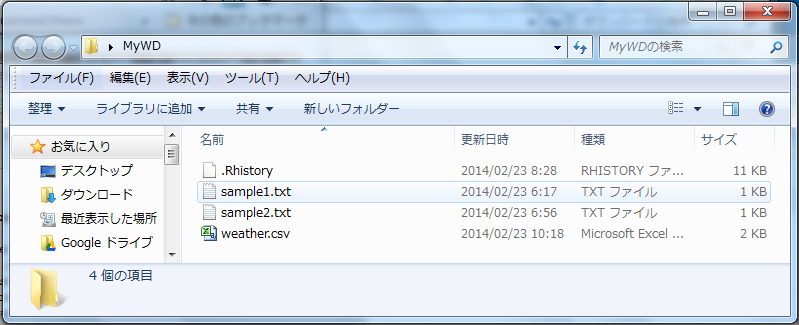
※その他にもいくつかの引数設定が可能です。詳細はhelp(scan)で確認してください。

### データフレームの読み込み（read.table）

　実務で最も多いのは、データセットをデータフレームとして読み込むread.tableです。分析対象のファイルは、主にtxtファイル、csvファイルのようなテキストファイルの形をしており、かつ保持するデータは列ごとに異なっていることが通常です。

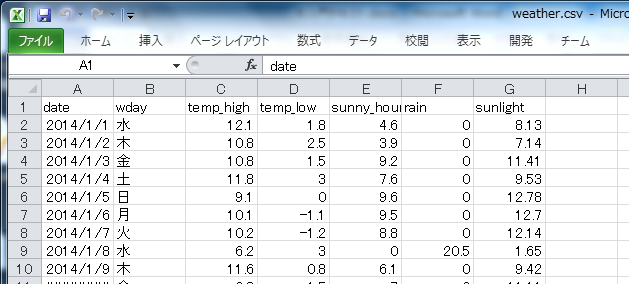
　こうしたデータをRに読み込むには、read.table()関数を使用します。（scan()関数との違いは、上記のようなファイルの中に収まる程度のデータを扱う、という点です。実務でも大半はこうしたファイルの読み込みとなると考えられますので、まずはread.table()の使い方を習得するのがよいでしょう。

読み込むファイル（weather.csv：気象庁からダウンロードしたデータを基に加工）



気象庁データダウンロード：http://www.data.jma.go.jp/gmd/risk/obsdl/index.php

Excelでweather.csvを開いたところ



read.table()関数でのデータファイル読み込み例

|  |  |
| --- | --- |
| > setwd("C:/Users/*UserHome*/Desktop/MyWD")　　　　# ワーキングディレクトリ設定  > weather<-read.table("weather.csv",sep=",",header=T)   |  | | --- | | 解説  　上記は、「区切り記号はカンマ","、1行目を項目名（header=T）として、weather.csvを読み込む」、という記述になる。 |   > nrow(weather)　　　　　　　　　　　# データ件数は31件  [1] 31  > head(weather,10)　　　　　　　　　 # 取り込んだデータの最初の10件を表示  date wday temp\_high temp\_low sunny\_hour rain sunlight  1 2014/1/1 水 12.1 1.8 4.6 0.0 8.13  2 2014/1/2 木 10.8 2.5 3.9 0.0 7.14  3 2014/1/3 金 10.8 1.5 9.2 0.0 11.41  4 2014/1/4 土 11.8 3.0 7.6 0.0 9.53  5 2014/1/5 日 9.1 0.0 9.6 0.0 12.78  6 2014/1/6 月 10.1 -1.1 9.5 0.0 12.70  7 2014/1/7 火 10.2 -1.2 8.8 0.0 12.14  8 2014/1/8 水 6.2 3.0 0.0 20.5 1.65  9 2014/1/9 木 11.6 0.8 6.1 0.0 9.42  10 2014/1/10 金 6.3 -1.5 7.0 0.0 11.11 |

　read.table()関数の設定（引数）について、主なものを以下の表にまとめます（表3-2-2）。読み込むデータセットの状態に合わせて、適切な設定を行うことが必要です。

表3-2-2　read.table()関数の主な引数

|  |  |
| --- | --- |
| 引数 | 説明 |
| file | 読み込むファイルを指定。　設定例：file=”sample.csv” |
| sep | 区切り文字の指定。カンマ”,”、タブ"\t"（R上は￥はバックスラッシュ）が一般的。  設定例：  ,seq=”,” |
| header | 1行目を項目名とする（TRUE）か、否か（FALSE）（指定がなければFALSE（1行目からデータと認識される）となっている）  設定例：　,header=T |
| col.names | 項目名を設定したベクトルを指定することで、読み込んだデータに項目名を付加します。   |  | | --- | | > colnames<-c("date","wday","temp\_high","temp\_low",  + "sunny\_hour"," rain","sunlight") # 項目名ベクトルを作成  > weather<-read.table("weather.csv",sep=",",skip=1,  + col.names=colnames)　# 1行目を読み飛ばし（,skip=1）、項目名に  　　　　　　　　　　　 # colnamesを利用。  > head(weather,5)  date wday temp\_high temp\_low sunny\_hour X.rain sunlight  1 2014/1/1 水 12.1 1.8 4.6 0 8.13  2 2014/1/2 木 10.8 2.5 3.9 0 7.14  3 2014/1/3 金 10.8 1.5 9.2 0 11.41  4 2014/1/4 土 11.8 3.0 7.6 0 9.53  5 2014/1/5 日 9.1 0.0 9.6 0 12.78 | |
| stringsAsFactors | 読み込み過程で、データを因子化するかしないかを設定。read.tableでは、文字列はすべて因子化しようとするため、,stringsAsFactors=F（FALSE）として、自動変換を抑制することができる。 |
| na.strings | 欠損値の文字列をRに伝える。 |
| comment.char | データ中に#があると、以降のデータはコメントとみなされてしまうため、,comment.char=”?”のように他の文字列を指定することで、正しく読み込まれる。データにコメントが含まれなければ,comment.char=””とすればよい。 |
| dec | 小数点の書式を指定。欧州等のように、小数点にカンマ(“,”)を指定するような場合は、,dec=”,”を指定する。 |
| encoding | 漢字、ひらがな、カタカナといった日本語など、ASCII文字以外が含まれる場合は、encoding=”shift\_jis”などとして、文字コードを指定する。 |
| nrow | 読み込む最大行を指定する。 |
| skip | 読み飛ばす行を指定する。 |
| fill | TRUEを設定すると、読み込み時に列数が少ない行があった時に、自動的にNA等を補う。  設定例：  　,fill=T |
| colClasses | 列のデータの型を指定する。設定は列数と同じ長さのベクトルに、numeric、characterなど型を指定して行う。  また列を読み飛ばす際にも利用する。この場合は、上記のベクトルにNULLを指定する。   |  | | --- | | > colcls<-c("Date","factor","numeric","numeric","numeric",  + "numeric","numeric") # すべての列のデータ型を指定  > weather<-read.table("weather.csv",sep=",",header=T,  + colClasses=colcls)　# colClassを指定。  > class(weather$date)　# dateは因子から日付に変わっている  [1] "Date" | |

　他にも、カンマ区切り（csv）、タブ区切り（tsv/txt）ファイルを読み込むread.csv()、read.csv2()、read.delim()などがあります。read.table()を基にしているため、引数などは同じものが使用できます。

　csvファイル同様、使用頻度が高いのがMicrosoft-Excelのxls/xlsxファイルです。R以外のソフトウェアで作成されたファイル（バイナリファイル）を直接読み込むことはできませんが、いくつか方法があります。

方法①：　Excelファイルをcsvファイルとして保存し、read.table()で読み込む

　Excelを開き、「名前を付けて保存」からcsvファイルとして保存します。その上で、R上でread.table()でデータを読み込みます。

※ちなみに、著者はいつもこの方法をとります。分析環境に余計なパッケージを導入したくないという意図があるからです。

方法②：　Excelのデータセルをコピーし、Rでクリップボードを読み込む

　Excelのデータセルをコピーすることで、一旦コンピュータのメモリ（クリップボード）上にデータを保管します。このクリップボードにあるデータを、以下のコマンドでRに読み込みます。

クリップボードのデータをRに読み込む例

|  |
| --- |
| > data<-read.delim("clipboard") |

方法③：　xls/xlsxファイル読み込み用のパッケージを使用する

代表的なものには、xlsxパッケージ（Java）、gdataパッケージ（Perl）などがあります。インストールすると、いずれもread.xlsx()関数を使えるようになります。

　Excel以外のソフトウェアに関しては、foreginパッケージをインストールすることでread.xxx()関数（xxxはソフトウェアを指す表記が入ります）が使えるようになり、直接データを読み込めます。SPSSやSAS、GNU Octaveなど多様なファイルをサポートしています。

### データセットの出力

　Rで分析処理を行った結果としてのデータセットを、別のソフトウェアでの再分析や、共同作業者および他の研究者・分析者に渡す、ということもあります。その際に、相手がどんなソフトウェアを使っているかわからない場合もあります。そうした時は、ソフトウェアがなんであれ、読み込める形式で渡すのが一般的です。基本的には、csvファイル、txtファイルに出力することになります。

　Rには、前述のscan()関数と同じようなデータを扱うwrite()関数と、read.table()と同じようなデータを扱うwrite.table()関数が用意されています。

　※なお、保存するファイル名にパスを記載しない場合はワーキングディレクトリに保存されます。

write()関数の使用例

|  |
| --- |
| > data1<-c(1,2,3,4,5,6)  > data2<-c(7,8,9,10)  > write(data1,"data1.txt",sep=",")　　　　　　　# ベクトルdata1をdata1.txtとして  　　　　　　　　　　　　　　　　　　　　　　　　# ワーキングディレクトリに保存  > write(data2,"data1.txt",append=T,sep=",")　　 # data2をdata1.txtに追加保存  > scan("data1.txt",what=0,sep=",")　　　　　　　# 保存されたファイルを読み込んで確認  Read 10 items  [1] 1 2 3 4 5 6 7 8 9 10 |

表3-2-3　write()関数の主な引数

|  |  |
| --- | --- |
| 引数 | 説明 |
| x | 書き出すベクトル、行列を指定。 |
| file | 出力するファイルのパスおよびファイル名を指定。  基本的にはテキストファイル（sample.txt、sample.csvなど）を指定する。  なお、パスの指定がない場合は、ワーキングディレクトリに保存される。 |
| ncolumns | データに含まれる列数（1行の要素数）を指定。 |
| append | 既に同名のファイルが同じ場所にある場合、そのファイルにデータを追加するにはT（TRUE）、既存のデータを破棄して上書きするにはF（FALSE）を指定する。 |
| sep | 区切り文字の指定。カンマ”,”、タブ"\t"（R上は￥はバックスラッシュ）が一般的。 |

write.table()関数の使用例

|  |
| --- |
| > gender<-c("F","F","M","M","M")  > heights<-c(170,156,175,168,178)  > weights<-c(54,45,75,68,74)  > data<-data.frame(gender,heights,weights)　　# データフレームdataを作成  > write.table(data,"dataset.csv",sep=",",quote=F)　# dataをdataset.csvとして保存  > read.table("dataset.csv",sep=",")  gender heights weights  1 F 170 54  2 F 156 45  3 M 175 75  4 M 168 68  5 M 178 74 |

表3-2-4　write.table()関数の主な引数

|  |  |
| --- | --- |
| 引数 | 説明 |
| x | 書き出すデータフレームを指定。 |
| file | 出力するファイルのパスおよびファイル名を指定。  基本的にはテキストファイル（sample.txt、sample.csvなど）を指定する。  なお、パスの指定がない場合は、ワーキングディレクトリに保存される。 |
| append | 既に同名のファイルが同じ場所にある場合、そのファイルにデータを追加するにはT（TRUE）、既存のデータを破棄して上書きするにはF（FALSE）を指定する。 |
| quote | 個々のデータに付与されるダブルクォーテーション（”）をつけるかどうかを設定。指定しなければTRUEとなり、ダブルクォーテーションがつく。  通常はクォーテーションをつけないので、F（FALSE）を設定する。  設定例：  　,quote=F |
| sep | 区切り文字の指定。カンマ”,”、タブ"\t"（R上は￥はバックスラッシュ）が一般的。 |
| eol | 各行の最後の文字を指定。通常は\n（\はバックスラッシュ） |
| na | データ中に欠損値として使われる文字列を指定 |
| dec | 小数点を表す文字を指定。 |
| row.names | 行名を付与するかどうかの論理値（T/F）または付与する行名を与える文字ベクトルを指定。 |
| col.names | 列名を付与するかどうかの論理値（T/F）または付与する列名を与える文字ベクトルを指定。 |
| qmethod | 文字列に含まれる引用記号をどう処理するかを指定。escape/doubleで指定。 |
| fileEncoding | 文字コードを指定。 |

## データの概要を把握する

　分析の最初の一歩は、まず分析対象であるデータについて理解することです。レコード件数をはじめとして、どのような値が含まれているか、どういう意味なのか、また分析するにあたっては、列データのそれぞれの型や因子の種類と度数、日付データであればいつからいつまでか、欠損値がどれくらい含まれているか、など、データについて理解しなくてはならないことは多岐に亘ります。

　ここでは、Rに読み込んだデータについて分析前の集約・要約方法についてまとめ、以降の分析の道筋を考えるプロセスについて説明します。

### 読み込みデータの最初（最後）の一部を見る（head、tail）

　読み込んだデータがどのような内容なのかを知るためには、データを格納した変数を入力してEnterキーを押せばすべてのデータが表示されます。実はprint()関数を使っている状態なのですが、Rではprint()を省略してもデータを表示します。なお、明示的にprint()を使用する場面としては、文字列データを表示させる際に、ダブルクォーテーション(“)を表示させない（,quote=F）場合などがあります。

　しかし、データ件数が10,000件、100,000件などデータの量が大きくなってくると、すべて表示させると表示完了まで時間がかかる、列名が見られない、など不都合な点も出てきます。

　そこで、データの最初の数件、最後の数件を表示させるのがhead()関数、tail()関数です。head()、tail()とも、引数には(データフレーム,行数)を指定します。

head()関数、tail()関数の使用例

|  |
| --- |
| > data<-read.table("weather.csv",sep=",",header=T)  > head(data,n=5)　　　　　　　　　　　　　　　　　　# head()の例　“n=”は省略可  date wday temp\_high temp\_low sunny\_hour rain sunlight  1 2014/1/1 水 12.1 1.8 4.6 0 8.13  2 2014/1/2 木 10.8 2.5 3.9 0 7.14  3 2014/1/3 金 10.8 1.5 9.2 0 11.41  4 2014/1/4 土 11.8 3.0 7.6 0 9.53  5 2014/1/5 日 9.1 0.0 9.6 0 12.78  > tail(data,n=5)　　　　　　　　　　　　　　　　　　# tail()の例  date wday temp\_high temp\_low sunny\_hour rain sunlight  27 2014/1/27 月 8.2 -1.3 9.9 0.0 14.33  28 2014/1/28 火 11.4 -1.3 8.4 0.0 13.08  29 2014/1/29 水 12.6 2.6 9.9 0.0 15.03  30 2014/1/30 木 6.5 2.2 0.0 12.5 1.08  31 2014/1/31 金 12.8 3.5 9.7 0.5 14.66 |

### 条件にあったデータを見る（subset、split）

　全体を俯瞰した後は、見たいデータにフォーカスしてさらに理解を深めます。条件抽出のための関数は比較的多くありますが、代表的なsubset()関数、およびsplit()関数について見ていきます。

　subset()関数では、引数をsubset(データフレーム,条件,列名のベクトル)と指定します。列名のベクトルは、分析に必要な列のみをc関数でベクトル化して指定します。

subset()関数の使用例

|  |
| --- |
| > data<-read.table("weather.csv",sep=",",header=T)  > subset(data,data$wday=="月")　　　　　　　　　　# wdayが月のレコードだけを抽出  date wday temp\_high temp\_low sunny\_hour rain sunlight  6 2014/1/6 月 10.1 -1.1 9.5 0 12.70  13 2014/1/13 月 7.6 -0.2 8.8 0 12.77  20 2014/1/20 月 9.0 -2.1 7.1 1 11.64  27 2014/1/27 月 8.2 -1.3 9.9 0 14.33  > subset(data,data$wday=="月",c(date,temp\_high))　# wdayが”月”のレコードのみを抽出  date temp\_high　　　　　　　　　　　　　　# し、dateとtemp\_highのみを表示  6 2014/1/6 10.1  13 2014/1/13 7.6  20 2014/1/20 9.0  27 2014/1/27 8.2 |

　split()関数は、その名の通りデータを分割する関数です。実行すると条件に合致（TRUE）するデータと、合致しない（FALSE）データに分けて、全件データが表示されます。

　引数は、split(データフレーム,列名または条件)と指定します。なお、split()関数は、’TRUE’と’FALSE’の２つの要素を含むリストを返します。したがって、リストの要素の取り出し方（[[要素番号または要素名]]）で条件に合ったデータのみを取得できます。

split()関数の使用例

|  |
| --- |
| > data<-read.table("weather.csv",sep=",",header=T)  > data$date<-as.Date(data$date)　　　　　　　　# dateを日付型に変換  > subdata<-subset(data,data$date<"2014-1-10")　# 2/10以前のデータを抽出  > split(subdata,subdata$wday=="火")　　　　　　# 火曜日をTRUEとして抽出  　　　　　　　　　　　　　　　　　　　　　　　 # splitは条件にあうものと合わないもの  $`FALSE`　　　　　　　　　　　　　　　　　　　 # に分けられる（split）  date wday temp\_high temp\_low sunny\_hour rain sunlight  1 2014-01-01 水 12.1 1.8 4.6 0.0 8.13  2 2014-01-02 木 10.8 2.5 3.9 0.0 7.14  3 2014-01-03 金 10.8 1.5 9.2 0.0 11.41  4 2014-01-04 土 11.8 3.0 7.6 0.0 9.53  5 2014-01-05 日 9.1 0.0 9.6 0.0 12.78  6 2014-01-06 月 10.1 -1.1 9.5 0.0 12.70  8 2014-01-08 水 6.2 3.0 0.0 20.5 1.65  9 2014-01-09 木 11.6 0.8 6.1 0.0 9.42  $`TRUE`  date wday temp\_high temp\_low sunny\_hour rain sunlight  7 2014-01-07 火 10.2 -1.2 8.8 0 12.14  > split(subdata,subdata$wday=="火")[['TRUE']]　　　　　# split()はTRUEとFALSEの  date wday temp\_high temp\_low sunny\_hour rain　 # リストを返す  7 2014-01-07 火 10.2 -1.2 8.8 0　 # [['TRUE']]　で条件に合った  sunlight　　　　　　　　　　　　　　　　　　　　　　 # リストの要素のみを返す  7 12.14 |

### データの性質を見る（class、mode、sapply、lapply、str）

　次に、対象となるデータの性質を確認する方法について見ていきます。これまでも、mode()関数やclass()関数といった関数でデータの型や構造を確認しました。ここでは、データフレームという一つのオブジェクトではなく、データフレームに含まれるベクトル、リストの性質を確認する方法について考えます。

個々のデータの性質を見る方法の例

|  |
| --- |
| > gender<-factor(c("F","F","M","M","M"))  > heights<-c(170,156,175,168,178)  > weights<-c(54,45,75,68,74)  > data<-data.frame(gender,heights,weights)　　# データフレームの作成  > class(data)　　　　　　　　　　　　　　　　# dataのclassはdata.frame  [1] "data.frame"  > class(data$gender)　　　　　　　　　　　　 # dataに含まれる特定の列（gender）の  [1] "factor"　　　　　　　　　　　　　　　　 # classはfactor  > mode(data$gender)　　　　　　　　　　　　　# modeはnumeric（因子化されると名前つき  [1] "numeric"　　　　　　　　　　　　　　　 # 数値として保存される） |

　一つ一つ見ていく場合はこれでもよいのですが、列数が多い時には少々大変です。データフレームに含まれる列データの性質を一括で確認するにはsaaply()関数やlapply()関数を使って、以下のようにします。

　sapply()、lapply()を含む、apply()関数群は、データフレームに含まれるすべての行または列または要素に対して一括して同じ関数・処理を行うための関数です。

saaply()関数、lapply()関数の使用例

|  |
| --- |
| > sapply(data,class)　　　　　　　　# sapply(データフレーム,適用する関数)の構文  gender heights weights 　　　# 結果は主に簡素化されたベクトルで返る  "factor" "numeric" "numeric" 　　　# ※[1]がついていないのに注意  > lapply(data,class)　　　　　　　　# lapply(データフレーム,適用する関数)の構文  $gender　　　　　　　　　　　　　　 # 結果はリスト形式で出力される  [1] "factor"  $heights  [1] "numeric"  $weights  [1] "numeric" |

全ての列ではなく、特定の複数列について確認したい時も往々にしてあります。条件にあう列だけを取り出す場合と、直接列を指定して取り出す場合について説明します。

　復習になりますが、任意の複数の列を指定するには、data[,c(“列名”,”列名”,･･･)]のように、列番号に条件を指定するのでした。抽出した内容もデータフレームですので、そのままsapply()関数に渡しても計算されます。

　なお、以下の例のように列名を指定することもできますし、c(2:3)のように列番号を指定して「ここからここまで」という指定もできます。多変量解析などでは、多数の入力変数が列として横に並ぶことが多いので、このような指定方法の方が適している場合もあります。

任意の複数の列についての調べ方の例

|  |
| --- |
| > sapply(data[,c("gender","heights")],class)　# 列名がgender、heightsの列のclassを  gender heights 　　　　　　　　　　　　　# 調べる  "factor" "numeric"  > sapply(data[,c(2:3)],class)　　　　　　　　 # c(2:3)のように連続する列番号を指定し  heights weights 　　　　　　　　　　　　　# ても結果が得られる  "numeric" "numeric" |

　また、条件に合致する列を抽出して処理を行うには次のように列番号に条件を設定します。ここでは、classがnumeric（数値）である列（ベクトル）を抽出し、各列の平均を求めています。

条件に合致する列についての調べ方の例

|  |
| --- |
| > sapply(data[,sapply(data,class)=="numeric"],mean)　# 条件に合う列の指定  heights weights 　　　　　　　　　　　　　　　　　　 # meanは平均を求める関数  169.4 63.2 |

　さて、ここまでデータの中身やデータの性質を調べる関数の説明をしてきましたが、実はもっと簡単に一括してデータの中身や性質を調べる方法があります。

　str()関数は、データの件数、列数、各列の性質などを表示してくれます。

str()関数の使用例

|  |
| --- |
| > str(data)  'data.frame': 5 obs. of 3 variables:　　　　　　　# 件数5、列数3のデータフレーム  $ gender : Factor w/ 2 levels "F","M": 1 1 2 2 2　　# genderのclassと値  $ heights: num 170 156 175 168 178　　　　　　　　 # heightsのclassと値  $ weights: num 54 45 75 68 74　　　　　　　　　　　# weightsのclassと値 |

### 基本統計量を見る（table、summary）

　データそのものや性質を見た後は、件数や基本的な統計量を見ます。その際に使用する関数にはtable()関数やsummary()関数、str()関数など便利な関数が用意されています。

　table()関数は、データセットの簡単な集約をするほか、簡単なクロス集計も行うことができます。

また、addmargins()関数を併せて使うと、table()関数の結果に各行・各列の合計(sum)を最終行または最右列に付加することができます。addmargins()関数の引数には、クロス集計の結果（tableまたはデータフレーム）とmargins（行または列の指定）があります。margins=1は行（縦方向）の合計（sum）、margins=2は列（横方向）の合計（sum）を付加します。行・列とも集計したい場合には、marginsにc(1,2)を指定します。

　なお、addmargins()関数には、,FUN=という引数があり、sum以外の集計関数を指定できますが、そもそもtable()関数が度数を集計する関数であるため、データは因子となります。一方で平均（mean）や中央値（median）などは、数値データの集計に使う関数です。データの型などの把握が、適切な関数選択、設定にも影響する一例です。

table()関数、addmargins()関数の使用例

|  |
| --- |
| > job<-factor(c("employee","selfemployed","employee","student","selfemployed"))  > gender<-factor(c("F","F","M","M","M"))  > age<-c(25,30,22,18,40)  > data<-data.frame(job,gender,age)  > data  job gender age  1 employee F 25  2 selfemployed F 30  3 employee M 22  4 student M 18  5 selfemployed M 40  > table(data$job)　　　　　　　　　　　　# 1つの列の集計　因子についての合計  employee selfemployed student  2 2 1  > table(data$job,data$gender)　　　　　　# 2つの列のクロス集計    F M  employee 1 1  selfemployed 1 1  student 0 1  > addmargins(table(data$job,data$gender),c(1,2))　# addmargins()を使うことで、行・列  　　　　　　　　　　　　　　　　　　# の合計を付加することもできる。  F M Sum　　　　　　　　　　　　　　# なお、c(1,2)は行・列の指定  employee 1 1 2　　　　　　　　　　　　　　# 1=列（下に合計）、2=行（右に合計）  selfemployed 1 1 2  student 0 1 1  Sum 2 3 5 |

　数値データの集計には、summary()が便利です。因子データについては、各列ごとの因子の種類とその件数、数値データについては、列内のデータの最小値(Min.)、第一分位(1st Qu.)、中央値（median）、平均（mean）、第三分位（3rd Qu.）、最大値（Max.）を返します。

　もちろん、特定の列について実行することも可能です。

summary()関数の使用例

|  |
| --- |
| > summary(data)　　　　　　　　　　　　　　　# データフレームに含まれる全部の列の集計  job gender age  employee :2 F:2 Min. :18  selfemployed:2 M:3 1st Qu.:22  student :1 Median :25  Mean :27  3rd Qu.:30  Max. :40  > summary(data$age)　　　　　　　　　　　　 # 特定の列についての集計  Min. 1st Qu. Median Mean 3rd Qu.  18 22 25 27 30  Max.  40 |

## その他のデータの整形

　ここまで、データの結合や併合、列の追加、行・列の削除、データの抽出、並び替えなど、データの型、構造とともに基本的なデータの加工方法について述べてきました。本項では、上記以外のデータの整形手法について説明します。

### データ形式の変更（table、dcast（reshapeパッケージ））

　スーパーマーケットなどでは顧客の購入履歴は、「トランザクションデータ」と呼ばれる形式で蓄積されていきます。すなわち、1件分のレコードには、「顧客ID」「購入品目」「数量」「合計」「購入日付」などです。しかし、例えばアソシエーション分析（連関分析などとも言います）で使用する場合、このままの形では分析できません。

　実はこの整形を簡単に行うのは前述したtable()関数です。

table()関数を使った整形例

|  |
| --- |
| > ID<-as.factor(c(101,102,103,101,103,104))  > prd<-c("apple","orange","melon","banana","cherry","grape")  > num<-c(1,5,3,4,10,3)  > total<-c(500,400,300,400,700,300)  > date<-as.Date(c("2014-2-1","2014-2-2","2014-2-2","2014-2-1","2014-2-3","2014-2-4"))  > data<-data.frame(ID,prd,num,total,date)  > data　　　　　　　　　　　　　　　　　　　　# サンプルデータはこういう形をしている  ID prd num total date  1 101 apple 1 500 2014-02-01  2 102 orange 5 400 2014-02-02  3 103 melon 3 300 2014-02-02  4 101 banana 4 400 2014-02-01  5 103 cherry 10 700 2014-02-03  6 104 grape 3 300 2014-02-04  > table(data$ID,data$prd)　　　　　　　　　　 # table()関数でクロス集計    apple banana cherry grape melon orange  101 1 1 0 0 0 0  102 0 0 0 0 0 1  103 0 0 1 0 1 0  104 0 0 0 1 0 0 |

　ただし、table()関数では、IDと商品（prd）以外のデータは無視されます。（もちろん引数にdataを指定すれば、すべての変数についてリストを出力しますが、分析には不向きです。）実際には、同一人物が同時に購入するパターンなどを探るのであれば、購入日付の考慮も必要です。また、数量を値に持つべき場合もあります。そのようなときに使うのがreshape2パッケージのdcast()関数です。

　dcast()関数と対になる関数に、acast()関数があります。acast()関数は、多次元配列を扱えるのですが、基本的にベクトル、配列、行列など同一の型のデータセットを扱います。dcast()関数はデータフレームを扱うことができます。

　dcast()関数はreshape2パッケージに含まれています。以下の方法でインストールします。

reshape2パッケージのインストール

|  |
| --- |
| > install.packages("reshape2")  > library(reshape2) |

※reshape/reshape2パッケージは、後述のグラフ作成のためのggplotパッケージに含まれています。ggplotパッケージをインストールすれば、上記のインストール手続きは不要です。

　dcast()関数がどのような挙動をするのかは、まずは使用例を見てください。その後でdcast()関数の引数について説明します。

dcast()関数の使用例

|  |
| --- |
| > dcast(data,ID+date~prd,length)　　　　　　　　　　　　# 値はtable()同様だが、IDと  Using date as value column: use value.var to override.　# 日付の組合せで集計している  ID date apple banana cherry grape melon orange  1 101 2014-02-01 1 1 0 0 0 0  2 102 2014-02-02 0 0 0 0 0 1  3 103 2014-02-02 0 0 0 0 1 0  4 103 2014-02-03 0 0 1 0 0 0  5 104 2014-02-04 0 0 0 1 0 0  > dcast(data,ID+date~prd,value.var="num",sum)　　　　　 # 値を数量（num）にして集計  ID date apple banana cherry grape melon orange # 集計方法は合計（sum）を指定  1 101 2014-02-01 1 4 0 0 0 0  2 102 2014-02-02 0 0 0 0 0 5  3 103 2014-02-02 0 0 0 0 3 0  4 103 2014-02-03 0 0 10 0 0 0  5 104 2014-02-04 0 0 0 3 0 0 |

表3-4-1　dcast()関数の主な引数

|  |  |
| --- | --- |
| 引数 | 説明 |
| data | 対象となるデータフレームを指定。 |
| formula | 目的変数~（チルダ）説明変数のように指定。  設定例：  　ID~prd　目的変数：ID、説明変数：prd  　ID+date~prd　目的変数：IDとdateの組合せ、説明変数：prd  目的変数は「基準となる列」、説明変数は「対応させる列に含まれる重複しない要素（上記例では、商品（prd）列に含まれるapple、banana…などになる）」と考える。 |
| fun.aggregate | aggregateは「集約」の意味。省略するとtable()関数と同じくlength（要素数）を値とする。上記の例でnumを値とした場合などは合計（sum）を指定する。その他必要に応じて平均（mean）、中央値（median）などを指定する。 |
| margins | 列合計、行合計を行う場合に指定する。（分析用データセットを用意する場合には不要なことが多い） |
| subset | 説明変数の要素を指定する。  設定例：  　,subset=.(prd==”apple”)  ※かっこの前に”.（ドット）”がついていることに注意   |  | | --- | | > dcast(data,ID+date~prd,subset=.(prd=="apple"),  + value.var="num",sum)  ID date apple  1 101 2014-02-01 1 | |
| fill | 値がない場合の処理について設定。明示的に設定しないとNAになる。  設定例：  　,fill=0　欠損値の場合、数値0が代入される。 |
| drop | 値が全くない行を非表示にするかしないかの設定。  設定例：  　,drop=T（標準）：TRUEに設定すると、値がない行は非表示になる。  　,drop=F　　　 ：FALSEを設定すると、値がない行もすべて表示する。 |
| value.var | 値となる列を指定。  設定例：  　,value.var=”num”  　,value.var=data$num  いずれの表記でも可。  指定しない場合、lengthの値が入る。 |

### 重複データの削除（unique、duplicated）

　元のデータセットに含まれるデータは、可能な限り使用するのが基本です。しかし、時には重複するデータを排除して分析を進める必要も出てきます。

　unique()関数は、データセットに含まれる重複行を排除する関数です。また、duplicated()関数は、重複している行データをTRUE、重複していない行データをFALSEとする論理ベクトルを返します。なお以下の例のとおり、重複データでも最初の行データはFALSEとなっていることに注意してください。以下に使用例を見ていきます。

unique()関数の使用例

|  |
| --- |
| > ID<-as.factor(c(101,102,103,101,103,104))  > gender<-as.factor(c("M","F","F","M","F","M"))  > data<-data.frame(ID,gender)  > data  ID gender　　　　　　　　　　　　　　　# ID：101と103が重複している  1 101 M  2 102 F  3 103 F  4 101 M  5 103 F  6 104 M  > unique(data)　　　　　　　　　　　　　　# unique(data)で、重複行を削除  ID gender  1 101 M  2 102 F  3 103 F  6 104 M  > duplicated(data)　　　　　　　　　　　　# duplicated()関数は、重複したデータをTRUE  [1] FALSE FALSE FALSE TRUE TRUE FALSE　 # とする論理値ベクトルを返す。 |

### 欠損値の扱い方

　欠損値については、その概要を第2章Rで扱うデータの姿>2.1データの型>特殊なデータ>欠損値、ヌル値、非数値のセクションでも説明しました。ここではデータフレームに含まれる欠損値NAの処理方法について、考えていきます。

　欠損値の扱いについては、大きく２つの方法があります。一つは欠損値を含むデータを除去する方法であり、もう一つは欠損データを補完する方法です。いずれをとるかについては、適切な分析を行う上でどちらが適切か、という観点から方法を決めるべきです。

#### 欠損データの除去

　欠損値を含むデータは基本的には除去することになります。分析の結果に影響するからです。ここでは欠損値を除去する様々な方法について説明します。

　基本統計量を求める関数（平均（mean）、分散（var）、合計（sum）、最小値（min）、最大値（max）など）には、引数に,na.rm=があり、TRUEを指定すると、欠損値NAを含むデータを集計から除外します。

　na.rm引数がない関数においても、第2章の欠損値のセクションで見たとおり、ベクトルであればx[!is.na(x)]のように否定記号”!”を使って、データセットからNAを含まないデータを抽出して関数に渡すことができます。

この方法は、データフレームにおいて、列を指定して欠損値除去を行う際にも使用できます。つまり、ある列データについては欠損値が含まれていると分析結果に影響がでるが、別の列データについては、欠損値を含んでいても行データとしては残しておきたい場合などに有効です。

“!”を使用した欠損値の除去例

|  |
| --- |
| > x<-c("a","b",NA,"c")　　　　　　# ベクトルの場合  > x[!is.na(x)]  [1] "a" "b" "c"  > data　　　　　　　　　　　　　　# データフレームの場合  ID gender age  1 101 M 25  2 102 F 30  3 103 <NA> 22  4 101 M 18  5 103 F 40  6 104 M 23  > data[!is.na(data$gender),]　　　# data$genderを指定すると、3番目の行データが除去  ID gender age　　　　　　　　　# される  1 101 M 25  2 102 F 30  4 101 M 18  5 103 F 40  6 104 M 23  > data[!is.na(data$age),]　　　　# 欠損値を含まないdata$ageを指定すると、3番目の行  ID gender age　　　　　　　　 # データはそのまま残る  1 101 M 25  2 102 F 30  3 103 <NA> 22  4 101 M 18  5 103 F 40  6 104 M 23 |

　線形回帰モデル（lm()関数）、一般線形回帰モデル（glm()関数）などでは、引数に,na.action=na.omitを設定することで、処理前のデータについて、欠損値を含む行データを除去します。

　na.omit()関数として、独立して使用することもできます。以下の例を見てください。

na.omit()の使用例

|  |
| --- |
| > data  ID gender age  1 101 M 25  2 102 F 30  3 103 <NA> 22  4 101 M 18  5 103 F 40  6 104 M 23  > om.data<-na.omit(data)　　　# 引数にデータフレームを指定するだけでよい  > om.data  ID gender age  1 101 M 25  2 102 F 30  4 101 M 18  5 103 F 40  6 104 M 23 |

#### 欠損データの補完（replace）

欠損値NAを0と解釈しても問題ない場合や、平均値を充てることで足りる場合などについては、データを置換します。データの置換にはreplace()関数を使用します。

replace()関数は、引数に(対象ベクトル,要素の番号,代替値)をとります。分かりにくいのですが、対象データはベクトルでありデータフレームでは列を指定します。また、要素の番号についてもベクトルで渡します。下記の例の中では、which()関数を使っていますが、which()関数は「( )内の条件に合う要素の番号」を返す関数です。

注意が必要なのは、今回代替値として列の平均を指定していますが、mean()関数では、欠損値NAを含む場合、結果がNAとなるので、引数に,na.rm=Tを指定することが必要です。

replace()関数の使用例

|  |
| --- |
| > data　　　　　　　　　　　　# 元データ  ID gender age  1 101 M 25  2 102 F 30  3 103 M 22  4 101 M 38  5 103 F NA  6 104 M 40  > data$age<-replace(data$age,which(is.na(data$age)),mean(data$age,na.rm=T))  > data　　　　　　　　　　　　# 代替値に列の平均値を指定している。注意点は、mean()  ID gender age　　　　　　　# na.rm=Tを指定しないと、代替値そのものがNAとなる。  1 101 M 25  2 102 F 30  3 103 M 22  4 101 M 38  5 103 F 31  6 104 M 40 |

　なお、replace()関数を使用せず、以下のように記述しても同じ結果となります。

|  |
| --- |
| data$age[which(is.na(data$age))]<-mean(data$age,na.rm=T) |

## ～統計Tips 3～　推測統計

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 推測統計は、分析対象の全体(母集団)の姿(母集団の統計量=母数)を小規模な抽出データ(標本)で推測する、という統計手法です。つまり、すべてのデータを収集することが不可能・現実的ではない場合に、入手できた一部のデータを使って、全体を推し量ってみよう、というものです。推定統計手法には、「推定」と「検定」の2種類があります。大まかにいうと、次のようになります。なお、統計量や推測値と言ったりしますが、「～量」は平均、中央値などの集計方法、「～値」はその値という意味です。  推定・・・標本から母集団の統計量（母数）を推定する。  検定・・・母集団についての仮説を立て、その仮説を採用すべきか否かを検証する。  **推定**  　推定は、母数の具体的な値を「推定」する手法です。推定方法には、「平均値は○である」と１つの推定値を出す点推定と、「平均値は△～□である」のように幅を持った区間で推定値を出す区間推定があります。  　推定の考え方は、母集団から抜き出した標本の統計量から母集団の統計量（推定量）を類推する、というシンプルなものです。つまり、標本の平均≒母集団の平均、のような要領です。    　課題意識としては、標本の統計量がそのまま母集団の統計量（母数）と考えてよいか、ということです。  　考え方の出発点は、母集団の統計量（母数）と、１回だけ抽出した標本の統計量は必ずしも一致しない、ということです。例えば、母集団が（１，２，３，４，５，６）だとしたとき、標本として２，３を取り出して平均を出したとしても、母集団の平均に一致しません。  　しかし、母集団から何回もサンプルを無作為に取り出して、その統計量を見たときに、その値に一定の出現傾向があるのではないか、と考えるわけです。  　例えば標本を抽出して平均値を計算する、ということを何回も繰り返した時、その平均値がどのような値になりやすいか、どのような値になりにくいか、の傾向が見えてきます。これを標本統計量の確率分布と言います。（なおこの時の平均値を確率変数と言います）。  この標本統計量の確率分布を標本分布と言います。そして、「何回も抽出・計算を繰り返した標本の統計量」の統計量は、母数に一致する（推定量に不偏性がある、と言います）ということになります。下図は、平均値50、標準偏差5の正規分布をとる母集団から、標本を5つ取り出して平均を出す、という作業を10000回行ったときの標本分布のヒストグラムと、母集団の正規分布グラフです。確かに母集団の平均である50と、標本分布の平均50が一致しているのが確認できます（下図で平均は山のピークのxの値）     |  | | --- | | for (i in 1:10000){  X<-rnorm(n=5,mean=50,sd=5)　# 平均50、標準偏差5の正規分布に従う母集団から標本を  mean\_X[i]<-mean(X)　　　　　# 5つ取り出して、その平均値をmean\_Xに順に格納の意味  }　　　　　　　　　　　　　　 # mean\_Xはlength=10000のベクトルとなる  hist(mean\_X,freq=F,ylim=c(0,0.2))  curve(dnorm(x,mean=50,sd=sqrt(5)),add=T) # 平均50、標準偏差5の母集団の分布グラフ |   　さて、記述統計のところで「データの中心や代表する点」と「データの範囲・ばらつき」を知ることがデータの概要を把握するには重要だと言いました。推定でも推定値を考える際のデータの把握方法は同じです。標本分布が大きく幅広い場合は、母数の推定が難しくなります。そこで使うのが標準誤差という考え方です。標準誤差は、「推定量の標本分布の標準偏差」と定義されます。つまり、標本分布における平均値からの距離、すなわち「ばらつき」「散布度」をもって標本分布の範囲を考えます。なお、標準誤差は次のように算出されます。  　　標本分布の分散 = 母集団の分散（）÷　標本分布の数（n）の公式に基づき、  　　標準誤差 = 標本分布の標準偏差 ＝ 母集団の標準偏差（δ）÷　標本数の平方根（）  　この時、標準誤差の解釈は、0なら理想的、もし外れてもおおむね標準誤差の範囲まで、と評価することになります。サンプル数が多いと、この標準誤差が小さくなり、母数推定の精度が上がる、ということになります。  　以上から、推定では、  　・母集団の統計量（母数）　←　「標本分布（標本の推定量の確率分布）」の推定量  　・標本分布のズレ　←　標本誤差で把握  と考えてください。  　なお、標本分布は平均だけでなく、分散や中央値などでも行われます。標本誤差を小さくする最適な標本統計量を探るのも統計解析の重要な作業の一つです。  **検定**  　検定は、統計的仮説検定と言います。推定では、標本の統計量から母集団の統計量を推定しましたが、検定の目的は、標本のデータの差や関連性から母集団のデータの差や関連性を考え、結論として差や関連性があると言えるか否かを判定する、というものです。  　推定の時の課題意識もそうですが、推測統計においては、100%の母集団推測は不可能であるという大前提があります（100%分かったら、統計は不要です）。当然、「たまたま異常値に偏った標本が取れてしまったのではないか」という不安は常にあるわけです。検定において課題意識となるのは、標本に含まれるデータから得た結論が、母集団において偶発的なものなのか、おおよそ適切なものと言ってよいのか、の判断をしたい、ということです。その観点から、検定の手順を見ていきます。  　検定は以下の手順で進めます。   |  |  | | --- | --- | | 手順 | 内容 | | 帰無仮説を設定する | 「主張したいこと（A）ではないこと（notA）が起こる確率は非常に低い=Aは妥当である。」と証明するために、まずはnotAを仮説として検証の対象とする。このときのnotAを帰無仮説（無に帰したい説）という。また、帰無仮説における標本分布を帰無分布という。 | | 検定統計量の設定する | 統計量というと混乱しやすいが、ここでは検定手法の選択と考えるとよい。以下に基本的な検定手法と利用場面をまとめる。   |  |  | | --- | --- | | 検定 | 内容 | | 標準正規分布による検定 | 帰無分布が正規分布に従う場合（分散も分かっている）ときに使用。 | | t検定  (t.test) | 母集団の分散が不明な場合に、帰無分布にt分布を利用する検定。t分布は自由度により分布の形が決まる。t検定においては、自由度=n-1（標本数-1）として利用。Rでは、t.test()関数を使う。  　　t.test( 標本統計量ベクトル , 帰無仮説 ,... )  引数は、変数ベクトル、2項検定の場合の2つ目のベクトル、帰無仮説を設定するmu（平均値）などがある。 | | 無相関検定  (cor.test) | 帰無仮説を「母集団において相関が0である」とする検定。母集団に２つの変数があり、これらが相関しているかどうか、を検定する。（例）母集団に国語と英語のテストが含まれている場合など  Rでは、cor.test()関数を利用する。  　 cor.test( 変数1 , 変数2 )  なお、無相関検定では、自由度n-2のt分布が使われている。 | | 独立性検定  (chisq.test) | 質的変数の独立性（無連関）に関する検定。  質的変数については、その度数を検定対象とする。カイ2乗分布を使用するためカイ2乗検定ともいう。Rでは、chisq.test()関数を利用。  　chisq( 度数のクロス集計表 ,...) |   この他にも、分布を前提としないノンパラメトリック検定などもある。 | | 有意水準と棄却域の設定 | 有意水準は、どの程度低い水準だったら、帰無仮説を棄却するか、という基準。5%または1%に設定されることが多い。表記としてα=0.05などと記載される。帰無仮説のもとで有意水準以下の発生確率の範囲を棄却域と呼び、標本データの検定統計量の結果が棄却域に入る場合は、帰無仮説を棄却する。 | | 標本データの検定統計量（実現値）を求める | 標本データを検定統計量で計算する。その結果が棄却域に入れば帰無仮説を棄却、採用域に入れば帰無仮説を採用する。  この時、検定統計量の結果以上の結果が出る確率をp値といい、p<αの時に、帰無仮説を棄却することになる。 | | 仮説の棄却/採用を決める | 検定統計量と、有意水準の大小から、帰無仮説の採否を決定する。 |   検定には、目的に応じて様々な検定手法が開発されています。しかし、基本的な考え方は上記のとおりです。それぞれの手法が最適な結論を導くために基本をどう修正しているのか、という観点で見ていくと、理解しやすくなります。  こうした検定の精度（採否判定の誤り）の評価として、危険率や検定力などがあります。ここでは説明を割愛しますが、こうした数値も見ながら検定を進める必要があります。 |

# データ計算・処理の基礎

　本章では、データの計算・処理の基本として、行列計算を含むRでの演算、関数の使用方法と関数適用に便利なapply()関数群の紹介、そしてグラフ化の基礎について説明します。

　また、CRANにある様々なパッケージの導入方法についても紹介します。

## 数値計算と行列計算

### 数値計算と演算子

　Rでの数値計算では、単に

|  |
| --- |
| > 1+1  [1] 2 |

と、すれば結果を求められます。この例で使っている「+（加算記号）」などを演算子と呼びます。以下にRで使用できる算術演算子について紹介します。

表4-1-1　算術演算子の一覧

|  |  |
| --- | --- |
| 記号 | 説明 |
| + | 加算、足し算　和を返す |
| - | 減算、引き算　差を返す |
| \* | 乗算、かけ算　積を返す |
| / | 除算　割り算　商を返す |
| ^ | べき乗　例：x^2 →ｘ2のこと |
| %/% | 整数商　商の整数部分 |
| %% | 剰余　商のあまり部分 |

　また、演算子には比較演算子/論理演算子というのもあります。＜（小なり）、＞（大なり）などの不等号や、!　（否定）などがこれに含まれます。前述の算術演算子および比較演算子や論理演算子は単体で使うことはあまりありません。むしろifやwhichなどの条件式などを作る際に頻繁に使用します。※

表4-1-2　比較演算子の一覧

|  |  |
| --- | --- |
| 記号 | 説明 |
| == | 等号　=（イコール）の意味　使用時は=を２つ重ねることに注意  ※なお、”=”　一つの場合、代入演算子 ”<-“ と同じ意味を持つが、非推奨の表現である。 |
| != | 否定　　の意味 |
| <= | 以下　　の意味　使用例：which(x<=5)… （xが5以下の要素番号） |
| >= | 以上　　の意味　使用例：if(i>=10)… （iが10以上ならば…） |
| < | 小なりの意味 |
| > | 大なりの意味 |

表4-1-3　論理演算子の一覧

|  |  |
| --- | --- |
| 記号 | 説明 |
| ! | 否定の意味　論理値と併せて使われることが多い  使用例：ベクトルxに含まれるNA値以外を抽出する場合  　!is.na(x) 「「xでNAと判定された要素」ではない要素がTRUE」  ※この例では、is.na(x)の結果ベクトルに含まれるTRUEをFALSEに、FALSEをTRUEにする |
| &、&& | AND（かつ）の意味　論理積  &は、要素ごとに比較をしていく際に使用、  &&は、ベクトルの最初の要素から順に評価が定まるまでしか行わない  &,&&の使用例   |  | | --- | | > x<-c(1,2,3,4,5)  > y<-c(5,4,3,2,1)  > z<-c(3,3,3,3,3)  > (x>y) & (x>z)　　　　　　　# 条件を満たす要素がTRUEになる  [1] FALSE FALSE FALSE TRUE TRUE  > (x>y) && (x>z)　　　　　　 # 最初の要素で評価確定している  [1] FALSE | |
| |、|| | OR（または）の意味　論理和  &同様、|は要素ごと、||は左から順に評価し確定したら評価終了  |、||の使用例   |  | | --- | | > x<-c(1,2,3,4,5)  > y<-c(5,4,3,2,1)  > z<-c(3,3,3,3,3)  > (x<y) || (x>z)  [1] TRUE  > (x<y) | (x>z)  [1] TRUE TRUE FALSE TRUE TRUE | |
| xor() | 排他的論理和の意味　関数形式で指定  排他的論理和とは、２つのベクトルがあるとき、他方と異なる値の場合はTRUE、その他の場合はFALSEとなる  xor()の使用例   |  | | --- | | > x<-c(0,1,1,0)  > y<-c(1,1,0,0)  > xor(x,y)  [1] TRUE FALSE TRUE FALSE | |

　なお、演算子が一つの式に複数含まれている場合について、一般的な数学同様どの演算子から計算されるかが決まっています。

表4-1-4　演算子の計算順序とまとめ

|  |  |  |
| --- | --- | --- |
| 順位 | 演算子 | 説明 |
| 1 | $ | リスト（データフレーム含む）要素を指定する |
| 2 | [,[[ | [　ベクトル要素を指定、[[　リスト要素を指定 |
| 3 | ^ | べき乗 |
| 4 | - | 単項演算のマイナス |
| 5 | : | 公差1の等差数列　1:3 → 1,2,3 |
| 6 | %\*%など | %で囲った演算子　他に%%、%/%などがある |
| 7 | \* , / | 乗算、除算 |
| 8 | + , - | 加算、減算 |
| 9 | < , >, <= , >= | 不等号 |
| 10 | == , != | 等号、否定 |
| 11 | ! | 否定 |
| 12 | &,&&,|,|| | 論理積、論理和 |
| 13 | <<- | 永続代入 |
| 14 | <- , -> | 代入演算子　右向き-> もある |

### 行列計算

　行列計算についても、数値計算同様に演算子を使用することができます。ただし、行列の積を求める場合、一般的な行列積には”%\*%”を使用することに注意してください。”\*”で計算すると、それは「要素ごとの積」となります。以下、用例を見てみます。

行列計算の例

|  |
| --- |
| > A<-matrix(c(1:6),2,3)  > B<-matrix(2,2,3)  > C<-matrix(3,3,2)  > A # 2×3の行列  [,1] [,2] [,3]  [1,] 1 3 5  [2,] 2 4 6  > B　　　　　　　　　　　　 # 2×3の行列  [,1] [,2] [,3]  [1,] 2 2 2  [2,] 2 2 2  > C　　　　　　　　　　　　　# 3×2の行列  [,1] [,2]  [1,] 3 3  [2,] 3 3  [3,] 3 3  > A+B　　　　　　　　　　　 # 行列の和　各要素を加算  [,1] [,2] [,3]  [1,] 3 5 7  [2,] 4 6 8  > A+C　　　　　　　　　　　　# （2×3）＋（3×2）の行列和は計算できない  Error in A + C : non-conformable arrays  > A\*B　　　　　　　　　　　　# 各要素の積  [,1] [,2] [,3]  [1,] 2 6 10  [2,] 4 8 12  > A%\*%C　　　　　　　　　　　# 一般的な行列積　乗算記号は %\*% を使う点に注意  [,1] [,2]　　　　　　　 # （2×3）×（3×2）→　（2×2）になっている。  [1,] 27 27  [2,] 36 36  > A%\*%t(B)　　　　　　　　　 # t()は行列転置の関数　これによりAとBの積を求められる  [,1] [,2]  [1,] 18 18  [2,] 24 24 |

## 関数

標準で実装される関数の他に、ライブラリにある膨大な数のパッケージを利用できるのもRのメリットです。本項では、関数についての基礎的な使い方、標準で用意されている数学関数、統計関数を紹介します。

また、関数の中でも非常に便利で使用頻度が高い、apply()関数群について説明します。

最後にCRANからのパッケージの導入方法についてご紹介します。

### 関数の利用

#### 関数について

ここまでの説明の中でも、かなり多くの関数を紹介してきましたが、そもそも関数とは何か、どう使うのかをここでまとめます。

関数は基本的に関数名と引数で構成されており、関数名と（　）内に引数を指定して使用します。

　関数名（引数1= , 引数2= …）

　これまで見てきたように、関数を単体で使えば結果を出力しますし、変数に代入すれば、二次利用も可能です。

関数の利用例

|  |
| --- |
| > sum(c(1,2,3,4,5))　　　# sumという関数名とc(1,2,3,4,5)という引数  [1] 15  > x<-sum(c(1,2,3,4,5))　 # 変数xに代入  > x+15  [1] 30 |

　そもそも、関数はfunction()で定義されたプログラムです。例えばreplace()関数について、R上で「replace」（かっこなし）と入力すると、replace()関数のソースコード（プログラム）を見ることができます。

replace()関数のコード

|  |
| --- |
| > replace　　　　　　　　　　　　# ソースコードではreshape<-function…と書かれている  function (x, list, values) 　　　# functionのあとの()内が、replace()関数の引数  {  x[list] <- values　　　　　　# 内部でこのような処理をしているから、適切に引数を指  x　　　　　　　　　　　　　　# 定しないとエラーメッセージが出ることが理解できる  } |

　なお、第1章でhelp()関数の使用方法について説明しましたが、このドキュメントには、各関数の引数について詳細が書かれています。初めて使う関数などについてはぜひ目を通して下さい。

#### 数学基本関数

　数学でよく使う関数もRで使用することができます。

以下に数学の基本関数についてまとめます。

表4-2-1　数学基本関数の一覧

|  |  |  |  |
| --- | --- | --- | --- |
| 関数 | 説明 | 入力例 | 結果 |
| sin() | サイン　正弦関数 | sin(1.2) | 0.932039 |
| asin() | sinの逆関数　ArcSin | asin(0.9) | 1.11977 |
| cos() | コサイン　余弦関数 | cos(1.2) | 0.362358 |
| acos() | cosの逆関数　ArcCos | acos(0.9) | 0.451027 |
| tan() | タンジェント　正接関数 | tan(1.2) | 2.572152 |
| atan() | tanの逆関数 | atan(0.9) | 0.732815 |
| sinh() | ハイパボリックサイン | sinh(1.2) | 1.509461 |
| asinh() | sinhの逆関数 | asinh(1.2) | 1.015973 |
| cosh() | ハイパボリックコサイン | cosh(1.2) | 1.810656 |
| acosh() | coshの逆関数 | acosh(1.2) | 0.622363 |
| tanh() | ハイパボリックタンジェント | tanh(1.2) | 0.833655 |
| atanh() | tanhの逆関数 | atanh(0.9) | 1.472219 |
| log() | 自然対数 | log(1.2) | 0.182322 |
| log10() | 常用対数（底が10） | log10(1.2) | 0.079181 |
| log2() | 底が2の対数 | log2(1.2) | 0.263034 |
| log1p() | log(1+x) | log1p(1.2) | 0.788457 |
| exp() | 自然対数の底数の（）乗 | exp(1.2) | 3.320117 |
| expm1() | exp()-1 | expm1(1.2) | 2.320117 |
| sqrt() | 平方根の計算 | sqrt(4) | 2 |
| round() | 四捨五入 | round(2.6) | 3 |
| trunc() | 整数部分 | trunc(3.2) | 3 |
| floor() | 小数切り捨て | floor(2.4) | 2 |
| ceiling() | 小数切り上げ | ceiling(3.2) | 4 |
| signif(x,n) | xを有効数字n桁で表示 | signif(1013,2) | 1000 |

※入力例、結果については、バイオスタティスティクス　数学基本関数より引用しました。

http://stat.biopapyrus.net/r/math-function.html

#### 統計基本関数

統計解析・分析を行うに当たっては、統計基本関数も重要です。実は多くは既にこれまでの説明の中で触れているものですが、ここで改めてまとめてみます。

表4-2-2　統計基本関数の一覧

|  |  |
| --- | --- |
| 関数 | 説明 |
| length() | ベクトルの長さ（要素数のこと） |
| sum() | ベクトル要素の総和 |
| prod() | ベクトル要素の積 |
| max() | 最大値 |
| min() | 最小値 |
| mean() | 平均値　（全レコードの合計）/　件数の算術平均 |
| median() | 中央値　最大値と最小値から等間隔の値 |
| quantile() | 分位点 |
| range() | 最大値と最小値の2値のベクトルを返す |
| cor(x,y) | ベクトルxとyの相関係数 |
| sd() | 標準偏差 |
| var() | 不偏分散：偏差平方和をデータ数-1で割った値 |
| summary() | ベクトルの統計量要約 |

### apply()関数群

　前項までは、基本的な関数について説明してきました。基本的には引数にベクトルや行列を指定して使います。しかし場合によっては、一括してグループ化された対象データに対して関数を適用したい、と考える場面も出てきます。そうした時に使用するのがapply()関数群です。

　第3章データを整備する>3.3 データの概要を把握する>データの性質を見る　の項でも、apply()関数群のsapply()関数とlapply()関数について既に触れています。その時はclass()関数を指定の列に適用する例でした。

　apply()関数群には、apply()、sapply()、lapply()、tapply()、mapply()がありますが、何をどのような場合に使用するのか、というのが本項のテーマです。

　使い分けの基準は、「データの構造」と考えてください。以下、次の３つの場面・構造で説明していきます。

1. データの構造が、ベクトル、配列、データフレーム
2. データの構造がリスト
3. データの構造がトランザクション

#### データの構造がベクトル、配列の場合：apply()関数

　apply()関数は、データセットがベクトルや配列の場合に適用します。配列を対象とする、ということは、多次元のデータに対しても適用できる、ということです。

　apply()関数の大きな特徴は、引数にMARGINS（次元）を設定することです。引数の設定方法は、

apply(データセット，次元（MARGINS），処理する関数)

となります。以下の例で説明します。

apply()関数の使用例

|  |
| --- |
| > ary2<-array(1,c(4,3,2))　　　　# 要素がすべて1の3次元配列を作成  > ary2  , , 1  [,1] [,2] [,3]  [1,] 1 1 1  [2,] 1 1 1  [3,] 1 1 1  [4,] 1 1 1  , , 2  [,1] [,2] [,3]  [1,] 1 1 1  [2,] 1 1 1  [3,] 1 1 1  [4,] 1 1 1  > apply(ary2,1,sum)　　　　　　　# 1次元目について合計（横方向に合算）  [1] 6 6 6 6  > apply(ary2,2,sum)　　　　　　　# 2次元目について合計（縦方向に合算）  [1] 8 8 8  > apply(ary2,3,sum)　　　　　　　# 3次元目について合計（奥方向に合算）  [1] 12 12 |

　注意点としては、実はデータフレームを引数に設定することもできますが、結果がおかしな状態になります。データフレームは通常、データの型が一様ではないデータの集合（リストの一態様）です。apply()関数では、プログラム言語上短絡評価（最初に評価が下されると、以降は判断しないという方式）しますので、例えば、先頭の列が「氏名」などの文字列であった場合は、その後に年齢や収入などのデータがあっても、文字列として判断します。(classなどで起こります)

　また、合計(sum)などでは、もちろんすべて数値データでないとエラーになります。

データフレームにapply()関数を適用した例

|  |  |  |
| --- | --- | --- |
| > body  gender heights weights  1 F 170 54  2 F 156 45  3 M 175 75  4 M 168 68  5 M 178 74  > apply(body,2,sum)　　　　　　　　　 # 文字列は計算できない、と怒られる例  Error in FUN(newX[, i], ...) : invalid 'type' (character) of argument  > apply(body,2,class)　　　　　　　　　# 短絡評価の例  gender heights weights  "character" "character" "character"   |  |  | | --- | --- | | 解説  　apply()関数でも、データフレームの列を指定すれば、処理はできます。しかし、後述する2次元専用の関数sapply()、lapply()があることを考えれば、無理に適用することはないでしょう。  apply()関数をデータフレームに適用（列指定）   |  | | --- | | > apply(body[,c(2,3)],2,mean)  heights weights  169.4 63.2 | | |

したがって、apply()関数の説明としては、「ベクトル、配列などのデータの型がすべて同じデータセットを扱う場合に適用」し、「多次元のデータも扱える」ということになります。

#### データの構造がリストの場合：sapply()関数、lapply()関数

　データの構造がデータフレームの場合は、sapply()関数、またはlapply()関数を使用します。この２つの関数は、データフレームを含むリストに対して適用します。

２つの関数は対象および設定方法は基本的に同じですが、出力方式が異なります。sapply()関数は名前つきベクトルで結果を出力し、lapply()関数はリストで結果を出力します。

データフレームに、apply()関数ではなくsapply()関数、lapply()関数を適用させる理由としては、２つあります。

1. 短絡評価をしない。（最初の要素（列）だけで判断せずすべての要素（列）について評価する）
2. 次元（MARGINS）の値は常に2であり、次元の設定が不要。

したがって、データフレームを扱うときにはよほど理由がなければ、sapply()関数、lapply()関数を使う、と考えてください。

sapply()関数、lapply()関数の引数の指定については、次元（MARGINS）は不要です。

　 sapply(データセット，処理する関数)

　 lapply(データセット，処理する関数)

　以下に使用例を示します。

sapply()関数、lapply()関数の使用例

|  |
| --- |
| > body  gender heights weights  1 F 170 54  2 F 156 45  3 M 175 75  4 M 168 68  5 M 178 74  > sapply(body,class)　　　　　　　# 短絡評価していない（すべての列が適切に表示される）  gender heights weights 　　# 出力形式が簡素化されたベクトル  "factor" "numeric" "numeric"  > lapply(body,class)　　　　　　　# 出力形式がリスト  $gender  [1] "factor"  $heights  [1] "numeric"  $weights  [1] "numeric"  > sapply(body[,c(2,3)],mean)　　　# 計算するには列指定するのはapply()と同じ  heights weights 　　　　　　　　　# 次元指定が不要であることと、短絡評価しないこと  169.4 63.2　　　　　　　　　 # から、データフレームにはsapply()、lapply()を使う |

#### データの構造がトランザクション

　前出のようにPOSデータなど行が時間の経過とともに次々と追加されていくデータをトランザクションデータと言います。多くの場合、顧客IDや商品、価格、購入日などの情報が含まれていますが、例えば、顧客IDや商品は因子（factor）となるデータです。復習ですが因子は要素の種類数が限定されたベクトルでした。これから説明しようとしているのは、こうした因子ごとに処理を適用させたい、という場合に使うtapply()関数です。

　tapply()関数が対象としているデータは、「集計対象のベクトル」と「因子の種類で構成された要素を値とするベクトル」の2つです。引数の設定の仕方は以下の通りです。

　 tapply( 集計対象データ（列），因子を含むベクトル（列），処理する関数 )

以下、使用例を見てみましょう。

tapply()関数の使用例

|  |
| --- |
| > pos  ID date prd num total  1 101 2014-02-01 orange 1 500  2 102 2014-02-02 grape 5 400  3 103 2014-02-02 grape 3 300  4 101 2014-02-01 melon 4 400  5 103 2014-02-03 banana 10 700  6 104 2014-02-04 orange 3 300  > tapply(pos$total,pos$prd,range)　　# 集計対象がtotal、因子の列がprd、集計方法は  $banana　　　　　　　　　　　　　　　# range（範囲：最大値と最小値）。  [1] 700 700　　　　　　　　　　　　　# 結果はリストになっている。一つの因子に複数の値  　　　　　　　　　　　　　　　　　　 # があるため。  $grape  [1] 300 400  $melon  [1] 400 400  $orange  [1] 300 500  > tapply(pos$total,pos$prd,sum)　　　# 今度は集計方法：sum（合計）の例  banana grape melon orange 　　　　# 結果が名前つきベクトルになっている  700 700 400 800 |

　なお、tapply()関数のような因子ごとでの処理は、重回帰分析等多変量解析では重要になってきます。

### パッケージの利用

　さて、これまでRの強みの一つとして、CRANにある膨大なパッケージを使用できることを上げてきました。ここでは、こうしたパッケージの導入方法について説明します。

　操作としては、非常に簡単で、以下のコマンドを入力します。

パッケージのインストールコマンド

|  |
| --- |
| > install.packages("パッケージ名") |

　ただし、パッケージをインストールしただけでは関数を使うことはできません。個別にRに読み込む必要があります。そのためのコマンドがlibrary()です。引数にあたるパッケージに”（ダブルクォーテーション）がついていないことに注意してください。

なお、引数無しでlibrary()と入力すると、インストールしている（読み込まれていないものも含む）の一覧が表示されます。

もう一点、library()に関して、一旦Rを終了し、次に起動すると、前にlibrary()で読み込んだパッケージも再度読み込まなくてはならない点に注意してください。

library()の使用例

|  |
| --- |
| > library(関数名) |

　パッケージの導入について、もうひとつ注意すべき点があります。あるパッケージを導入する際に、そのパッケージに含まれる関数が別のパッケージの関数を「引用」している場合、その別のパッケージも自動的にインストールする、ということです。

　例えば、前出のreshapeパッケージですが、これはグラフ表示のためのggplotパッケージを実行するうえで必要となる（依存関係がある）ため、明示的にインストールしなくてもggplotパッケージをインストールする際に自動的にreshapeパッケージもインストールします。

　通常は、こうした自動インストールは大変便利なのですが、自動インストールを回避したい、というときは、引数として　,dependencies=FALSE　と指定します。

自動インストールを回避したインストールの例（ggplot2の例）

|  |
| --- |
| install.packages("ggplot2", dependencies = FALSE) |

## 視覚化（グラフ）の基礎

　Rにはグラフ作図の機能も備わっていて、これが強みの一つとも言えます。視覚化（グラフ作図）についても、Rには多数のパッケージが用意されています。

　本節では、その中でも最もベーシックで、最も使いやすいplot()関数について説明します。また、その他よく使用するグラフ作図関数についても紹介します。

### その前に（高水準グラフィックス/低水準グラフィックス）

　Rのグラフ関数には、高水準グラフィックスと低水準グラフィックスの2種類があります。違いは使い方にあります。まず高水準グラフィックスで描画した上で、低水準グラフィックスを重ねていく、というイメージです。以下にグラフィックスに関する代表的な関数を紹介します。（なお、高水準＋高水準の場合でも、後から重ねるグラフの引数に、,add=Tを設定すると、重なります）

表4-3-1　グラフィックス関連関数

|  |  |  |
| --- | --- | --- |
| 関数 | 水準 | 機能 |
| curve | 高水準 | 曲線を描く |
| plot | 高水準 | 様々なグラフを描く |
| hist | 高水準 | ヒストグラムを描く |
| matplot | 高水準 | 多次元のplot |
| lines | 低水準 | 折れ線グラフを描く |
| points | 低水準 | ドットプロットを描く |
| text | 低水準 | 文字列を書き込む |
| rect | 低水準 | 長方形を描く |
| legend() | 低水準 | 凡例を描く　,legend=(c(“sun”,”temp”)) |
| abline | 低水準 | 直線を描く |

### グラフを作成する（plot：高水準グラフィックス）

　それでは、早速グラフを作っていきます。少し多めの観測データを使いたいので、前出（第3章 データを整備する>3.2 データの読み込みと出力>データフレームの読み込み）で使用した気象データ（weather.csv）から、最高気温と日射量の２つの要素だけを取り出したW（データフレーム）を作成します。

データの取り込みと加工

|  |
| --- |
| > wet<-read.table("weather.csv",header=T,sep=",")  > wet$date<-as.Date(wet$date) |

　ここから、グラフを作るのですが、コードは以下の通りです。

Plot()関数での作図例

|  |
| --- |
| > plot(wet$sunlight,wet$temp\_high) |

以上です。驚くほど簡単にプロットできました。しかし、プロットしただけなので、データの意味合いなどはなかなか読み取れません。

### 回帰線を引いてみる（abline：低水準グラフィックス）

　回帰線とは、ざっくり言うと「すべての点からの距離の総和が最小となる直線」です。Rには回帰式を求める関数としてlm()関数などがあります。これを使って、上図に線を引いてみます。

回帰線を加える例

|  |
| --- |
| > plot(wet$sunlight,wet$temp\_high)  > x<-lm(wet$sunlight~wet$temp\_high)  > abline(x) |

　これを見ると、関係性があるように見えます。そこで、では２つの値の相関はどうか、という議論に進んでいくわけです。ちなみに、相関係数をcor()関数で求めた結果は以下の通りです。

tempとsunの相関係数

|  |
| --- |
| > cor(wet$sunlight,wet$temp\_high)  [1] 0.383256 |

### グラフの種類

　上記の例では、グラフが点のプロットで作成されました。例えばこれが時系列のデータだったらどうでしょうか。どちらかというと、折れ線グラフのほうがよさそうです。こうしたグラフの種類の指定には、plot()関数の引数、,type= を指定します。

表4-3-2　plot()関数のtype引数

|  |  |
| --- | --- |
| 引数 | 説明 |
| ,type=”p” | ドットプロット　点を配置する（無指定の場合はこれになる） |
| ,type=”l” | 折れ線グラフ |
| ,type=”b” | 点と線の併記 |
| ,type=”c” | type=”b”の点を消したもの |
| ,type=”o” | 点と線を重ねて表示 |
| ,type=”h” | 各点からx軸までの垂線をプロット |
| ,type=”s” | 値を段階で表示（縦棒グラフがつながったイメージ）　左隣の値が基準 |
| ,type=”S” | 右隣の値を基準として段階で表示 |
| ,type=”n” | 軸のみ表示 |

　今度は同じく気象データから、日付と温度を取って折れ線グラフで表現した例を紹介します。

折れ線グラフ作図の例

|  |
| --- |
| > plot(wet$date,wet$sunlight,type="l") |

### タイトルをつける

　グラフにタイトルや軸名を付けます。まずはplot()関数の引数を表にまとめます。

表4-3-3　タイトルに関する引数

|  |  |
| --- | --- |
| 引数 | 説明 |
| ,main=”” | メインタイトルの設定　設定例：,main=”温度と日照量” |
| ,sub=”” | サブタイトルの設定 |
| ,xlab=””/ylab=”” | x軸、y軸の設定 |
| ,xlim=””/,ylim=”” | x軸、y軸の範囲設定　設定例,xlim=c(0,10) のように最小値と最大値のベクトルを設定する |
| ,asp=”” | x軸とy軸のアスペクト比　縦横の比率 |
| ,log=”” | メモリの対数表示　設定値は以下の３つのみ  　,log=”x”　x軸を対数表示  　,log=”y”　y軸を対数表示  　,log=”xy”　両軸を対数表示 |
| ,panel.first=grid(,) | 補助線設定　設定例：,panel.first=grid(8,8)　gridの引数は（縦線の数,横線の数）と考える |
| ,lowess | 線を平滑化する |
| ,col= | 線、点の色の設定　設定例：,col=1（黒）または ,col=”black” |
| ,pch= | 点の形　設定値は、  　,pch=0　点の形を四角にする  　,pch=1　点の形を丸にする（標準はこちら） |
| ,cex= | 点の大きさ |
| ,lty= | 線の種類を設定  　,lty=1　実線  　,lty=2　破線１  　,lty=3　点線  　,lty=4　破線２  　,lty=5　破線３  2,4,5については、破線の種類が異なる |
| lwd= | 線の太さ |

　上表をもとに、タイトルを「date vs sunlight」、x軸ラベルを「date」y軸ラベルを「sunlight」横の補助線を10本、点と線を併記し、点と線の色は赤、線は破線という設定をしたのが以下になります。

修飾されたグラフの例

|  |
| --- |
| > head(wet,5)  date wday temp\_high temp\_low sunny\_hour rain sunlight  1 2014-01-01 水 12.1 1.8 4.6 0 8.13  2 2014-01-02 木 10.8 2.5 3.9 0 7.14  3 2014-01-03 金 10.8 1.5 9.2 0 11.41  4 2014-01-04 土 11.8 3.0 7.6 0 9.53  5 2014-01-05 日 9.1 0.0 9.6 0 12.78  > plot(wet$date,wet$sunlight,main="date vs sunlight"  + ,xlab="date",ylab="sunlight",panel.first=grid(0,10),type="o",col=2,lty=3,) |

　統計解析・分析は、常にデータとのにらめっこです。こうした視覚化は、データの姿を見るために、昔から培われてきた技術です。そうした背景からも、Rのグラフィックは非常に奥が深く、またサポートするパッケージ群も充実しているのです。

　本項での概括的な説明は以上ですが、ぜひ参考文献等を紐解き、統計手法に応じた視覚化の方法論を身につけてください。

## ～統計Tips 4～　多変量解析

|  |
| --- |
| 現実世界のデータには、1変量で存在するものは少ないのではないでしょうか。それぞれの変数が複雑に絡み合ってデータはできています。  　こうした多数の変量の関係を解明していこう、というのが多変量解析のテーマです。とはいえ、ベースになっているのはこれまで見てきた記述統計や推測統計です。以下、主な多変量解析について、概要を紹介します。  **回帰分析（重回帰分析）（lm)**  回帰分析は、いうなれば、xを与えるとyの値が求まる式を求める分析と言えます。例えば、y=2x+3という懐かしい一次式も、立派な回帰式（単回帰分析）と言えます。  ただし注意が必要なのは、観測データの多くは直線上に並んでいない、ということです。したがって回帰分析のテーマは、「観測されたデータからの距離が最小となる直線を求める」となります。直線を求める、というのは、例の2や3の値を求めるということです。  さて、上記のy=2x+3は変数xが一つでしたが、これが複数になると重回帰分析となります。式は、 と、複雑になりますが、要は傾きと切片を求めているわけです。（は、パラメータとも言います。）  回帰分析は、後述する機械学習やその他の分析の基礎となる重要な考え方です。  **主成分分析（prcomp、princomp）**  　主成分分析は、回帰分析の応用と考えてください。といっても、上記のθを「重み」と考え、どの要素（）が結果yに対して重要度が高いか、と考える分析です。  **因子分析（factanal）**  　回帰分析では観測データから回帰線を求めました。主成分分析では、観測データから係数（重み）を求めました。因子分析では、そもそも観測データそのものがどのような要因によって構成されているか、共通する要因は何か、を考える分析です。  **判別分析（lda）**  　判別分析は、回帰とはやや趣が違います。最も違う点は「引く線」です。回帰分析では、最適なy（目的変数）を求める「線」でしたが、判別分析で求める線は対象とする２つ（または複数）の変数グループの境界線となる線（線形判別関数）です。各観測データが、正ならグループ１、負ならグループ2というように、個々のデータを「判別」していくわけです。  　他にもさまざまな多変量解析手法がありますが、この「どんな線を引くか」すなわち線の関数を考えるという観点は、後述する機械学習でも重要なポイントとなります。 |

# プログラミングの基礎

　プログラミング、というと非常に敷居が高いイメージがあります。しかし、Rにおけるプログラミング、少なくとも本テキストで言うプログラミングは、あくまで「処理の自動化」「反復利用する」ための手段です。

　いつも行う手順をプログラム化し、コマンドラインでの操作の労力を軽減するのが目的です。

　本章では、Rにおけるプログラミングの基礎的な知識に触れた後、実際の統計解析場面での実例を通して、Rをどのように使用するのかについて説明します。

## 制御構文

　制御構文には、大きく2つのテーマがあります。一つは「もし～だったら～を処理」という条件分岐、もうひとつは「～という処理を何回繰り返す / ～という条件まで繰り返す」という繰り返しです。

　この２つの方法を習得すれば、プログラミングの表現の幅が飛躍的に広がります。

### 条件分岐構文（if）

　「もし～だったら～を処理」など、設定した条件に応じて処理を変えたい、という場面は非常に多く起こります。これをどうRで表現するか、が本項のテーマです。

#### if文の構文

　基本的な構文は以下の通りです。

if ( 条件１ ) 処理１

　条件１がTRUEなら、処理１を実行する、という意味です。簡単な例を示します。

if文の例

|  |
| --- |
| > i<-10  > j<-10  > if (i==j) print("match!")  [1] "match!" |

　「もし～なら、△、そうでなければ●」のように分岐することもできます。elseを使って次のように記述します。

if ( 条件１ ) {

処理１

} else {

処理２

}

　条件１がTRUEなら、処理１を実行、条件１がFALSEなら、処理２を実行、という意味になります。複数行にわたるときは、処理の部分を{ }で囲みます。また、複数条件がある場合には次のように記述します。

if ( 条件１ ) {

処理１

} else if ( 条件２ ) {

処理２

} else if ( 条件３ ) {

処理３

} else {

処理４

}

　条件１がTRUEなら、処理１を実行、条件１がFALSEで、条件２がTRUEなら、処理２を実行、条件１と２がFALSEで、条件３がTRUEなら処理３を実行、どの条件でもFALSEなら、処理４を実行、という意味です。条件判断は上から順にされ、TRUEが出た時点で他の条件は無視されます。したがって、条件１にも条件２にも合致する場合、実行されるのは処理１となります。なお、else if の節の数はどれだけあっても問題ありません。

　記法として、次のような形も許されています。

簡素な記法の例

|  |
| --- |
| > i<-5  > j<-10  > l<-if (i==5) i else j　# もしiが５なら、ｌにiを代入、iが5でなければlにjを代入  > l  [1] 5 |

　Rのコードは簡潔を旨とします。コードが複雑になればなるほど、こうした簡潔な表現を心がけるべきです。

　なお、if文を使用する場合に最も注意しなくてはならない点は、上記の各条件において、TRUE/FALSEの判断は「最初の要素」について行われる、ということです。どういうことかというと、例えば、x<-c(1,2,3,4,5)というベクトルがあって、順に条件を見ていきたいとします。その場合、if文では、最初の要素、すなわち1を基準に判断してしまうのです。したがって、「xの要素の中で、もし3より大きい場合は、～」という処理ができません。

　このような場合には、後述の繰り返し処理や、前出のapply()関数群と組み合わせていくことになります。

#### ifelse()関数

　if文は、「最初の要素だけで判断する」と説明しましたが、特にベクトルを扱う場合には、ifelse()関数という関数も用意されています。こちらは、ベクトルの要素すべてについて判断してくれます。設定する引数は、条件、TRUEの場合の処理、FALSEの場合の処理の３つです。

ifelse( 条件 , TRUEの場合の処理 , FALSEの場合の処理 )

使用例を見てみます。

ifelse()関数の使用例

|  |
| --- |
| > x<-c(-5:5)  > y<-ifelse(x>0,1,0)  > y  [1] 0 0 0 0 0 0 1 1 1 1 1 |

### 繰り返し構文

　繰り返し構文は、同じ処理を繰り返す場合に使用します。細かく言うと、「処理１を、変数を１ずつ変えながら条件を満たすまで実行する」と言った方が良いかもしれません。ただし、Rでは繰り返し構文を使う場面はあまりないかもしれません。なぜなら、すでに紹介した行列計算やapply()関数群があり、全レコードに対しての処理の実行は繰り返し構文を使うまでもないからです。

　とはいえ、他のプログラム言語の素養を持つ方や、特殊なシミュレーションを行う場合などを想定すると抑えておく価値はあるかもしれません。

　繰り返し構文には、for、while、repeatがあります。以下それぞれについて紹介していきます。

#### for文

　for文は繰り返し処理の代表です。基本的な構文は次の通りです。

for ( 変数 in ベクトル ) {

　処理

}

　変数は、繰り返し用の変数の意味です。主には処理の中で使うベクトルなどの要素番号として使用されます。使用例を見てみます。

for構文の使用例

|  |
| --- |
| > x<-c(-10:-1)  > x  [1] -10 -9 -8 -7 -6 -5 -4 -3 -2 -1  > for (i in c(1:10) ){　　　　　　　　　　　# 繰り返し用変数iが1～10まで繰り返す  + y[i]<- x[i]　　　　　　　　　　　　　　　# yのi番目にxのi番目を代入  + }　　　　　　　　　　　　　　　　　　　　 # }を入力することでコードの終わりを宣言  > y　　　　　　　　　　　　　　　　　　　　 # プロンプト”> “が戻ってくる  [1] -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 |

if文同様、forも1行の文中で使われます。しかし、変数への代入などはできません。forはあくまでも処理手順の記述だからです。上記の例を一文で記載すると次のようになります。

一文で記載するfor文の例

|  |
| --- |
| > for (i in c(1:10) ) y[i]<-x[i]  > y  [1] -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 |

#### while文、repeat文

　for文同様、while、repeatも繰り返しを制御します。なお、for文を含め、繰り返しを停止するにはbreakを使うこともできます。（repeatに関しては、停止条件を設定できないので、breakは必須です。）以下使用例を見てみましょう。

while文の使用例

|  |
| --- |
| > i<-1  > while(i<=10) i<-i\*2　　# iが10以下の場合にi\*2を繰り返し、iに結果を代入  > i　　　　　　　　　　　# i=8のときはまだ条件範囲内なので、8\*2=16は実行される。  [1] 16　　　　　　　　　 # i=16>10のため、処理を停止（16\*2は実行されない）  >  > i<-1　　　　　　　　　 # 上記と同じ例。　if文を使って停止条件を設定。  > while(TRUE){　　　　　 # i>10を満たした場合、breakで処理を抜ける  + i<-i\*2  + if (i>10) break  + }  > i  [1] 16 |

repeat文の使用例

|  |
| --- |
| > i<-1  > repeat{　　　　　　　　# 上記while文の例と同じ例　repeatには停止条件を設定できない  + i<-i\*2  + if (i>10) break　　　# したがって、if (条件) breakは必須  + }  > i  [1] 16 |

## Rのコード編集

　まずは、Rでのプログラムの構成およびプログラムファイルの作成方法、およびRコンソールでの呼び出し方について説明します。

### Rのプログラムファイルの中身

　Rのプログラムファイルは、「.R」などの拡張子を付けたテキストファイル（以下、.Rファイルと呼びます）です。テキストファイルなので、編集は主にテキストエディタを使用します。テキストエディタとは、例えばMicrosoftWindowsのメモ帳など、文字修飾情報のないプレーンなテキストを編集するソフトウェアのことです。

　では、.Rファイルの中身には、何が書かれているのでしょうか。これは

そもそも、Rの操作はコマンドラインのコンソール（画面）で行うのが主流です。そうするとわざわざ別のファイルにコードを書いておくのはなぜか、ということになります。

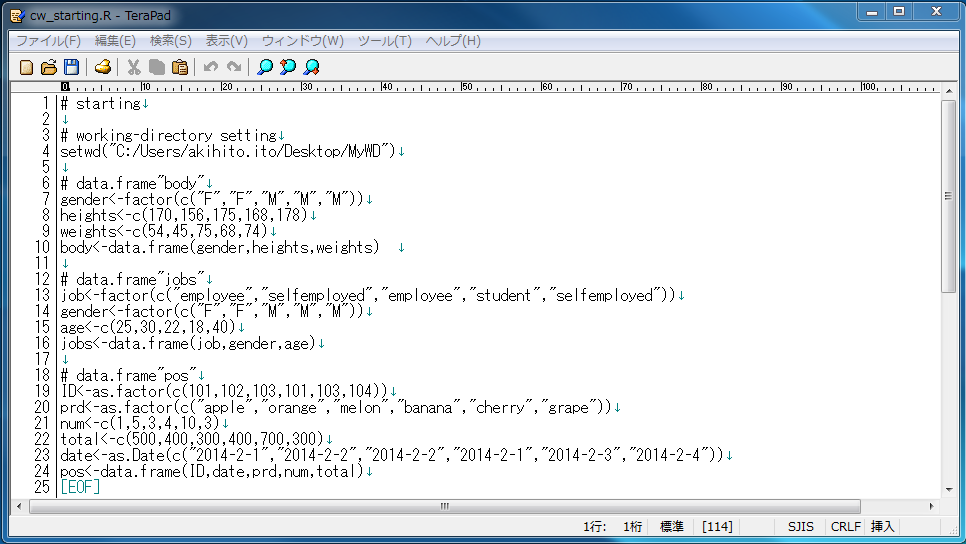
.Rファイルを作成する目的は、大きく２つあります。ひとつはいつも繰り返し行う処理を記録しておいて作業を簡素化する「自動化/処理の集約」、もう一つは標準関数では実現できない処理、複数の関数を組み合わせて複雑な処理を行うための自作の「関数」を作成することです。関数作成に関しては、その先には開発した分析理論の実装検証など、他者が見てわかる記録を残す、というのも考えられます。

いずれにしても「自動化/処理の集約」と「関数作成」の２つの目的が.Rファイルの中身、ということになります。

#### 処理の自動化

処理の集約については、例えば分析手法の検証のために仮に作成したサンプルのデータセットを、Rを起動するたびに作成するのは非常に面倒です。もちろん一連の作業ということであれば、同じ内容のサンプルにしなくてはなりません。こうしたデータセットや変数を.Rファイルに書き残しておけば、効率的に作業を始めることができます。以下に、本テキストを作成するに当たって作成した”cw\_starting.R”を紹介します。

作業自動化・処理の集約の例（”cw\_starting.R”の例）



cw\_starting.Rソースコード

|  |  |
| --- | --- |
| # starting  # working-directory setting　　　　　　　　　　# ワーキングディレクトリの設定  setwd("C:/Users/akihito.ito/Desktop/MyWD")  # data.frame"body"　　　　　　　　　　　　　　# データフレーム”body”の設定  gender<-factor(c("F","F","M","M","M"))  heights<-c(170,156,175,168,178)  weights<-c(54,45,75,68,74)  body<-data.frame(gender,heights,weights)  # data.frame"jobs"　　　　　　　　　　　　　　# データフレーム”jobs”の設定  job<-factor(c("employee","selfemployed","employee","student","selfemployed"))  gender<-factor(c("F","F","M","M","M"))  age<-c(25,30,22,18,40)  jobs<-data.frame(job,gender,age)  # data.frame"pos"　　　　　　　　　　　　　　　# データフレーム”pos”の設定  ID<-as.factor(c(101,102,103,101,103,104))  prd<-as.factor(c("apple","orange","melon","banana","cherry","grape"))  num<-c(1,5,3,4,10,3)  total<-c(500,400,300,400,700,300)  date<-as.Date(c("2014-2-1","2014-2-2","2014-2-2","2014-2-1","2014-2-3","2014-2-4"))  pos<-data.frame(ID,date,prd,num,total)   |  | | --- | | 解説  　最初のワーキングディレクトリの設定については、Rの初期設定で起動時のディレクトリ設定をすることもできますが、分析プロジェクト単位でディレクトリが変わることを考えて、私はRのインストール時のデフォルト（初期値）のままにしています。そのディレクトリにこのcw\_starting.Rを置いて、R起動時に実行しています。 | |

　cw\_starting.Rをsource()関数で読み込みます。ファイル名はダブルクオーテーション（”）で囲むことに注意してください。

なお、読み込み後にちゃんと変数やワーキングディレクトリが変更されているかも確認します。変数の確認は、objects()関数で行います。引数なしで入力します。ワーキングディレクトリの確認は復習になりますが、getwd()関数を使います。

cw\_starting.Rの読み込み

|  |
| --- |
| > source("~/cw\_starting.R")　　　　　　　# cw\_starting.Rファイルの読み込み  > objects()　　　　　　　　　　　　　　　# 読み込まれた変数の確認  [1] "age" "body" "date" "gender" "heights" "ID"  [7] "job" "jobs" "num" "pos" "prd" "total"  [13] "weights"  > getwd()　　　　　　　　　　　　　　　　# ワーキングディレクトリの確認  [1] "C:/Users/akihito.ito/Desktop/MyWD" |

#### 関数の作成

関数に関しては、第4章 データの計算・処理の基礎>4.2 関数>関数の利用 のところでも触れましたが、「関数<-function(引数){処理内容}」を用いた文書です。ファイルの形式は自動化と同じく.Rファイルです。

関数を自作するには、大きく２つの方法があります。ひとつは、既存関数を引用（ラッピング）して、作成するもの、もうひとつは、既存関数を書き直して新たな関数を作成するものです。完全にゼロから作るにしても、少なからず既存関数を利用する上記のどちらかになります。

後者はオープンソースならでは、ではありますが、改修する関数がまた別の関数を引用していたり、methodを使った複雑な関数群で構成されていたりすると、非常に手間がかかります。できることなら、前者で確実に動作する関数を土台に、積み重ねていく方が効率的で、必要な結果に短期間でたどり着くことができます。

それでは、関数を作成する例を紹介します。少々地味ですが、ロジスティック回帰やニューラルネットワークのベースとなるシグモイド関数を作ってみます。

ここでは、自然対数の底数のべき乗を返すexp()関数を引用しています。後は一般的なシグモイド関数の公式を再現しています。

標準シグモイド関数の公式

\varsigma _{1}(x)={\frac  {1}{1+e^{{-x}}}}

シグモイド関数の作成：sigmoid.R



.Rファイルを、ワーキングディレクトリに保存します。その上で、以下のコマンドを実行します。

自作シグモイド関数の利用

|  |
| --- |
| > source("sigmoid.R")　　　　　　　　　　　# 自作関数の読み込み  > a<-c(-5:5)　　　　　　　　　　　　　　　 # サンプルのベクトルデータの作成  > sigmoid(a)　　　　　　　　　　　　　　　 # 自作関数に適用  [1] 0.006692851 0.017986210 0.047425873  [4] 0.119202922 0.268941421 0.500000000  [7] 0.731058579 0.880797078 0.952574127  [10] 0.982013790 0.993307149  > plot(sigmoid(a),type="o")　　　　　　　　# プロットすると以下のようになる |

## ～統計Tips 5～　機械学習とデータマイニング

|  |
| --- |
| 統計Tipsの最後は、機械学習とデータマイニングについて紹介します。  **機械学習**  　機械学習は、MachineLearningの直訳です（もっといい名前もあったろうにと思います）。  　さて、推測統計では収集不可能な母集団から、標本を取り出して母集団について推測しようというアプローチでした。また、多変量解析では、歴史的に非常に優れた数々の理論が構築されましたが、それは実データでの手計算・検証が難しかったから、とも言えます。  　近年、コンピュータの性能が飛躍的に向上したことにより、これまで難しかった全件計算が可能となり、また複雑な計算アルゴリズムも実装が容易になりました。そして、コンピュータを使って実データの統計的処理を行うのが機械学習ということになります。  　機械学習にもさまざまな手法がありますが、基本的な概念は以下の図で示されます。    　この図の中で、非常に重要な概念は、黄色い部分「教師あり学習」「教師なし学習」の分類です。ここで言う「教師あり」とは、観測データの他に学習すべき「答え」がある、つまり、xだけでなくyの値もある、ということです。逆に「教師なし」は観測されたデータしかない、という意味です。  次に薄赤色の分類ですが「答えとなるデータがある」という場合には、分析手法として「回帰」と「分類」の2種類があるということです。回帰と分類の違いは、実はTips4で説明した、回帰分析と判別分析の違いとほぼ同じです。もちろん、分析手法としては、機械学習ならではというものまで多岐に亘りますが、基本的な理論ベースは同じと考えてよいでしょう。  一方で、「答えとなるデータがない」場合には、クラスタリングが適しています。クラスタリングは、観測データ同士の関係性から、データをいくつかの「クラスタ」に分類するものですが、yに当たる目的変数を要しません（わかりにくいかもしれませんが、判別分析は「教師あり学習」に分類されます。分析対象となるデータに「このデータはAに属し、別のデータはBに属し...」というようにデータに分類名（教師データ、ラベルとも言います）がついていて、これを基に線形判別関数を導き出すからです）。  分析実務では、クラスタリングを使って教師がないデータに対してクラスタというラベルをつけ、これを教師として「教師あり学習」の分析に持ち込む、ということを比較的よく行います。  **データマイニング**  　最後にデータマイニングについて紹介します。データマイニングは、これまで見てきた統計的手法、多変量解析、機械学習などデータ分析の様々な手法を駆使して、データから有意義な知見を「掘り当てる」という仕事です。  　この「有意義な」が重要で、特に分析の実ビジネスにおいては、分析によってメリットを享受する人物にとって「価値がある」と評価されること、これが「有意義である」という状態です。  そしてそのためには、「データ分析によってメリットを享受すべき人物は誰」で、その人は「どのようなミッションを持っているのか」といった背景的情報を把握し、そこから「分析の目的」と「検証すべき仮説」を導き出し、最適な分析手法の選定、分析のためのデータ加工、分析の実施を経て、出てきた結果について上記の人物にとってどのような意味があるかを評価し、それを価値あるものと理解してもらえるようなプレゼンテーションを行うところまでが分析者の役割です。  　ツールを使っての「分析」自体は、理論とツールの使い方さえ分かればそれほど大変ではありません。それよりも、いかに核心に迫る目的設定ができるか、そして結果をいかに価値あるものとして伝えることができるか、が分析者の腕でありデータマイニングの醍醐味と言えるでしょう。 |

# 参考文献

【書籍】

* P.スペクター　著　石田基広／石田和枝　訳:「Rデータ自由自在（原題:Data Manipulation with R）」丸善出版,2012
* Norman Matloff著　大橋真也監訳　木下哲也訳：「アート・オブ・Rプログラミング（原題：THE ART OF R PROGRAMMING）」オライリージャパン,2012
* 山田剛史、杉澤武俊、村井潤一郎　共著：「Rによるやさしい統計学」オーム社,2008

【WEB上の資料】

* 間瀬茂　著：「R基本統計関数マニュアル」

<http://cran.r-project.org/doc/contrib/manuals-jp/Mase-Rstatman.pdf‎>

* R Development Core Team著：「R言語定義」

<http://cran.r-project.org/doc/contrib/manuals-jp/R-lang.jp.v110.pdf‎>

* 舟尾暢男　著：「R-Tips（PDF版）」

<http://cse.naro.affrc.go.jp/takezawa/r-tips.pdf‎>

【ウェブサイト】

* Google

<https://www.google.co.jp/>

* The R Project for Statistical Computing

<http://www.r-project.org/index.html>

　Rのグローバルの公式サイトです。マニュアルやFAQ、メーリングリストの他、CRANの情報などが掲載されています。

* RjpWiki

<http://www.okada.jp.org/RWiki/>

　筑波大学の岡田昌史先生が運営する国内最大のR情報交換サイトです。基本的な使い方から、ハイレベルな統計処理についての議論など、たいへん興味深い情報が満載です。

* R-Tips/R-Source

<http://cse.naro.affrc.go.jp/takezawa/r-tips/r2.html>

書籍「The R Tips―データ解析環境Rの基本技・グラフィックス活用集」の著者である舟尾暢男氏が作成したページです。基本的なRの使い方に関する情報が豊富です。

* RとLinuxと…

<http://rmecab.jp/wiki/index.php?FrontPage>

　数々のR関連、テキストマイニング関連の書籍の著者である石田基広先生（徳島大学）の備忘録サイトです。

* 金子邦彦研究室

<http://www.kkaneko.com/index-j.html>

　九州大学の金子邦彦先生のウェブサイトです。Rに関しても非常にコンパクトに必要な情報をまとめています。なおRのページは

<http://www.kkaneko.com/rinkou/r/index.html>

* バイオスタティスティクス

<http://stat.biopapyrus.net/r/>

　Rについて非常にわかりやすくまとめてあります。

# 変更履歴

|  |  |  |
| --- | --- | --- |
| **版数** | **変更内容** | **年月日** |
| 1.0 | 新規作成 | 2014/MAR/2 |