Brief Introduction to R

どんぐり研究所　孫 建強

Brief Introduction to R

Contens

基本文法

- ◯ 条件構文
- ◯ 繰り返し構文
- ◯ 関数
- ◯ パッケージ

基本文法

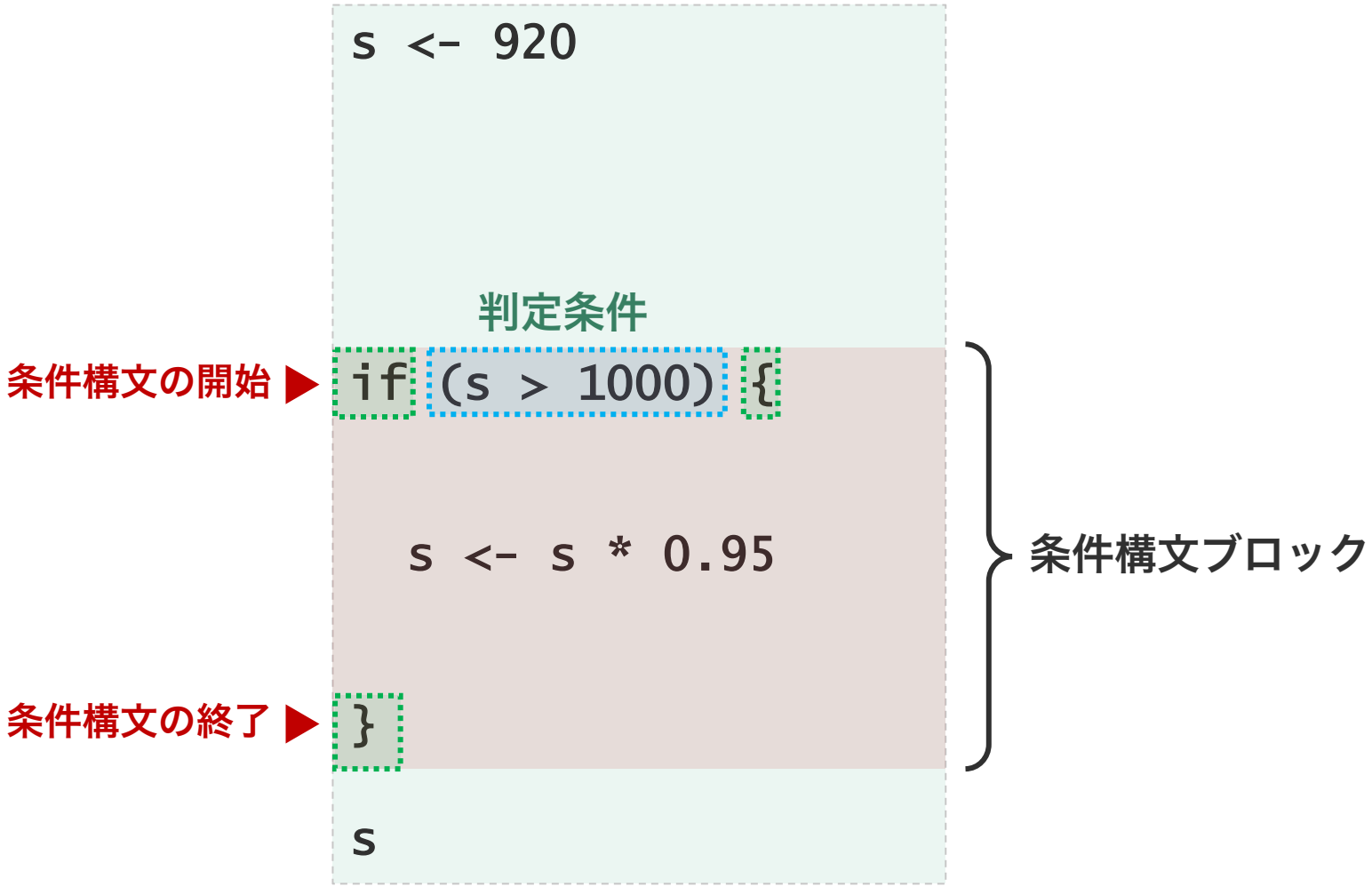
- ◯ 条件構文
- ◯ 繰り返し構文
- ◯ 関数
- ◯ パッケージ

条件構文

R の条件構文は if、else if、else などの単語（予約語）を使う。条件構文は必ず if から始まる。条件構文の 1 行目には、if とともに条件を書く。条件成立時に行う処理は、2 行目以降に書く。ただし、条件成立時の処理が、条件構文の一部であることを明示するために、その処理前後を { } で囲む。右の例は、「小計が 1000 円を超えたならば 5% 割引（もし s > 1000 ならば、s に 0.95 をかける）。」処理を行なっている。

```
s <- 920

if (s > 1000) {
  s <- s * 0.95
}

s
# [1] 920



s <- 1800

if (s > 1000) {
  s <- s * 0.95
}

s
# [1] 1710
```

if 構文



if 構文

条件構文は、与えられた条件が真（TRUE）であるかどうかを判定して、分岐処理を行う構文である。条件は不等式などの判定で与えるのが一般的である。

演算子	処理
a == b	a と b が等しいならば TRUE
a != b	a と b が等しくなければ TRUE
a > b	a が b よりも大きければ TRUE
a >= b	a が b 以上ならば TRUE
a < b	a が b よりも小さければ TRUE
a <= b	a が b 以下ならば TRUE

```
s <- 920

x <- (s > 1000)
# [1] FALSE


if (s > 1000) {
  s <- s * 0.95
}

s
# [1] 920
```

入れ子構造

複数の条件に対して条件判定を行うとき、条件構文を 2 つ重ねた入れ子構造にすることで実現できる。例えば、「毎月 20 日に 1000 円以上の買い物を行った時に 5% 値引きする」といった処理は、まず、会計日が 20 日かどうかを判定して、次に、会計日が 20 日であれば購入金額が 1000 円以上かどうかを判定する。これを if 構文で書くと右のようになる 。

```
s <- 1200
d <- 20


if (d == 20) {
  if (s >= 1000) {
    s <- s * 0.95
  }
}


s
# [1] 1140
```

if 構文が 2 つあるので、2 つとも閉じることを忘れずに！

論理演算

複数の条件を同時に判断するときは、入れ子構造の条件構文を利用するほか、複数の条件を論理演算した結果を条件構文の判定条件として使うこともできる。論理演算でよく使われる演算として、論理積と論理和である。論理積は、英語の AND に相当するものである。論理和は、英語の OR に相当するものである。

論理演算子	演算
x && y	論理積
x & y	論理積（ベクトル）
x || y	論理和
x | y	倫理和（ベクトル）
xor(x, y)	排他的論理和

```
s1 <- s2 <- 1200
d <- 20


if (d == 20) {
  if (s1 >= 1000) {
    s1 <- s1 * 0.95
  }
}

s1
# [1] 1140


if ((d == 20) && (s2 >= 1000)) {
  s2 <- s2 * 0.95
}

s2
# [1] 1140
```

if 構文の入れ子構造

2 つの判定条件を論理演算している

論理演算

条件 1	条件 2	論理積	論理和	排他的論理和
x	y	x && y	x || y	xor(x, y)
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	FALSE

if-else 構文

ある条件の真偽判定に基づいて処理を分けたい場合は、条件構文 if を 2 つ書くことで対応できる。しかし、if 構文を 2 つ続けて書くことで、同じ条件を 2 回判断していることになる。理論的にも、処理的にも煩雑である。このような場合は、if と else を用いて条件構文を作成すると、理論的にも処理的にもシンプルになる。

```
a <- 180
b <- 120
s <- 0


takeout <- TRUE



if (takeout == TRUE) {
    s <- (a + b) * 1.08
}

if (takeout != TRUE) {
    s <- (a + b) * 1.10
}

s
# 324
```

if-else 構文

ある条件の真偽判定に基づいて処理を分けたい場合は、条件構文 if を 2 つ書くことで対応できる。しかし、if 構文を 2 つ続けて書くことで、同じ条件を 2 回判断していることになる。理論的にも、処理的にも煩雑である。このような場合は、if と else を用いて条件構文を作成すると、理論的にも処理的にもシンプルになる。

```
a <- 180
b <- 120
s <- 0


takeout <- TRUE



if (takeout) {
    s <- (a + b) * 1.08
}

if (!takeout) {
    s <- (a + b) * 1.10
}

s
# 324
```

if-else 構文

ある条件の真偽判定に基づいて処理を分けたい場合は、条件構文 if を 2 つ書くことで対応できる。しかし、if 構文を 2 つ続けて書くことで、同じ条件を 2 回判断していることになる。理論的にも、処理的にも煩雑である。このような場合は、if と else を用いて条件構文を作成すると、理論的にも処理的にもシンプルになる。

```
a <- 180
b <- 120
s <- 0


takeout <- TRUE



if (takeout) {
    s <- (a + b) * 1.08
}

if (!takeout) {
    s <- (a + b) * 1.10
}

s
# 324
```

```
a <- 180
b <- 120
s <- 0


takeout <- TRUE



if (takeout) {
    s <- (a + b) * 1.08

} else {
    s <- (a + b) * 1.10
}

s
# 324
```

if-else 構文

ある条件の真偽判定に基づいて処理を分けたい場合は、条件構文 if を 2 つ書くことで対応できる。しかし、if 構文を 2 つ続けて書くことで、同じ条件を 2 回判断していることになる。理論的にも、処理的にも煩雑である。このような場合は、if と else を用いて条件構文を作成すると、理論的にも処理的にもシンプルになる。

```
a <- 180
b <- 120
s <- 0


takeout <- TRUE



if (takeout) {
    s <- (a + b) * 1.08


} else {
    s <- (a + b) * 1.10
}

s
# 324
```

条件判定 ▶

条件成立時に実行

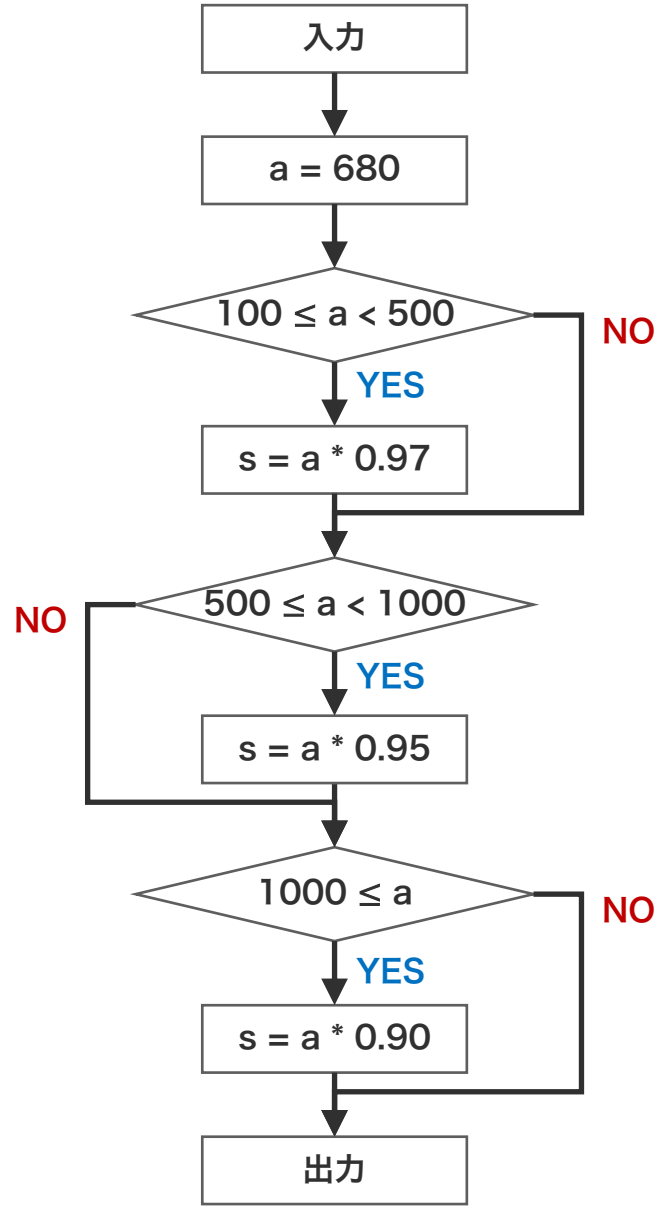
条件不成立時に実行

if-else if-else 構文

1 つのオブジェクトに対して複数の閾値を設けて条件判定したい場合は、複数の if 構文を使って書くことができる。しかし、この場合、ロジックも複雑になることに注意する必要がある。

小計	割引
100 円以上 500 円未満	3%
500 円以上 1000 円未満	5%
1000 円以上	10%

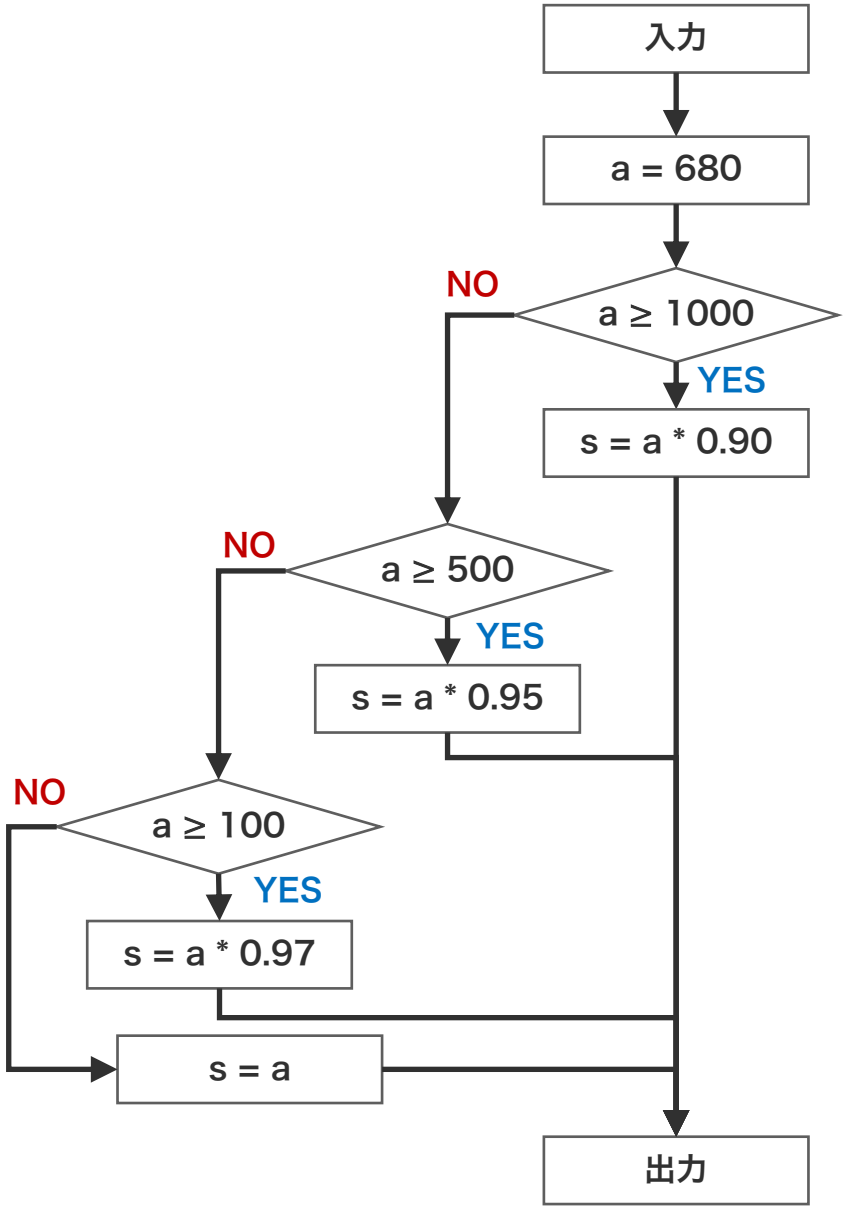
```
a <- 680
s <- 0


if (100 <= a && a < 500) {
    s <- a * 0.97
}
if (500 <= a && a <1000) {
    s <- a * 0.95
}
if (a >= 1000) {
    s <- a * 0.90
}



s
# 646
```



if-else if-else 構文

複数条件で処理を分岐させるときは、複数の if 構文を組み合わせるよりも、if-elif-else 構文を適切に使用した方がロジックもシンプルになる。また、if-elif-else 構文を使用することで、条件の取りこぼしが少なく、デバッグも行いやすい。

小計	割引
100 円以上 500 円未満	3%
500 円以上 1000 円未満	5%
1000 円以上	10%




```r
a <- 680
s <- 0


if (a >= 1000) {
    s <- a * 0.90

} else if (a >= 500) {
    s <- a * 0.95

} else if (a >= 100) {
    s <- a * 0.97

} else {
    s <- a
}

s
# 646
```

if-else if-else 構文

複数条件で処理を分岐させるときは、複数の if 構文を組み合わせるよりも、if-elif-else 構文を適切に使用した方がロジックもシンプルになる。また、if-elif-else 構文を使用することで、条件の取りこぼしが少なく、デバッグも行いやすい。

小計	割引
100 円以上 500 円未満	3%
500 円以上 1000 円未満	5%
1000 円以上	10%

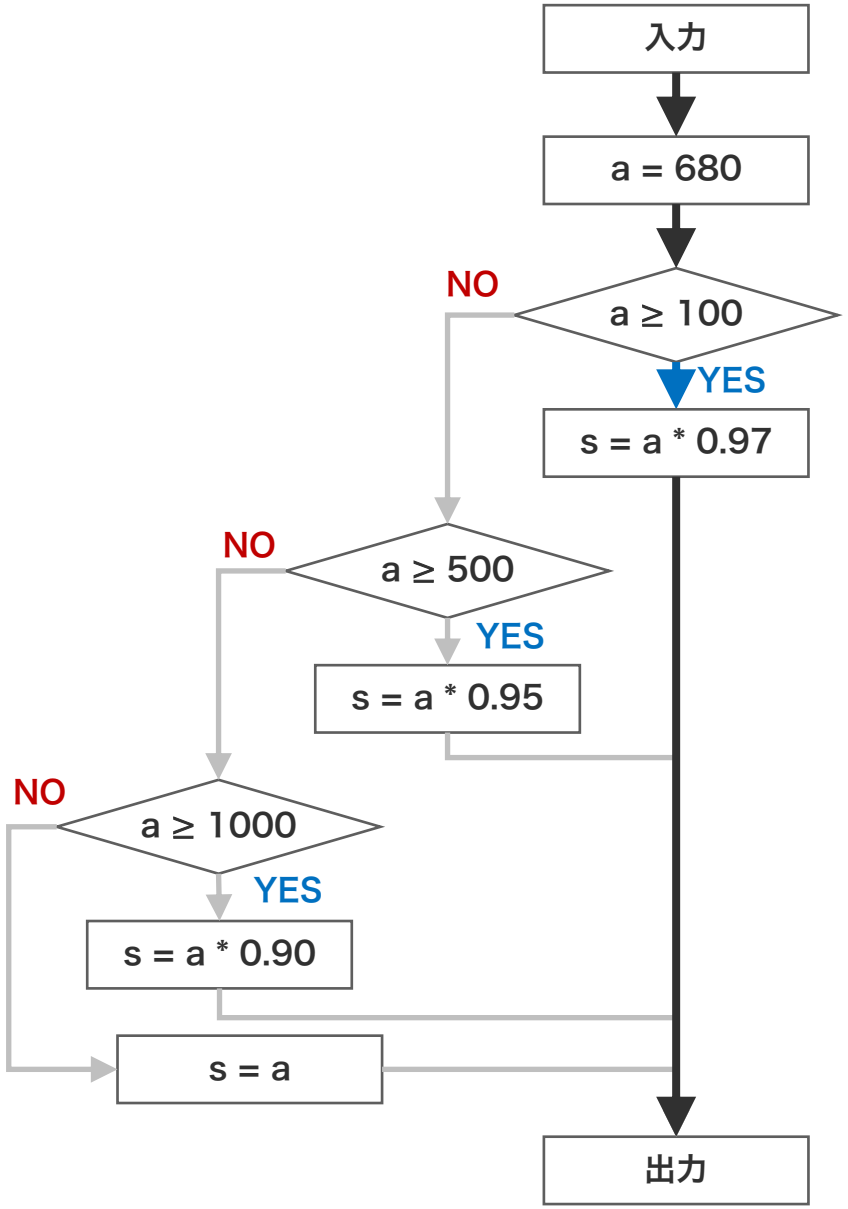
```
a <- 680
s <- 0


if (100 <= a && a < 500) {
    s <- a * 0.97
}
if (500 <= a && a <1000) {
    s <- a * 0.95
}
if (a >= 1000) {
    s <- a * 0.90
}




s
# 646
```

```
a <- 680
s <- 0


if (a >= 1000) {
    s <- a * 0.90

} else if (a >= 500) {
    s <- a * 0.95

} else if (a >= 100) {
    s <- a * 0.97

} else {
    s <- a
}

s
# 646
```

if-else if-else 構文

条件が複数あるとき、各条件の優先順序に注意を払う必要がある。条件の優先順序を間違えると、想定外の処理が行われることがある。

小計	割引
100 円以上 500 円未満	3%
500 円以上 1000 円未満	5%
1000 円以上	10%




```
a <- 680
s <- 0


if (a >= 100) {
    s <- a * 0.97

} else if (a >= 500) {
   s <- a * 0.95

} else if (a >= 1000) {
    s <- a * 0.90

} else {
    s <- a
}

s
# 659.6
```

基本文法

- ○ 条件構文
- ○ 繰り返し構文
- ○ 関数
- ○ パッケージ

繰り返し構文

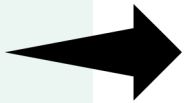
ベクトル a の各要素の合計を計算して、変数 n に代入する手順を考えよ。ただし、一度に 2 数値の足し算しかできないものとする。

```
a <- c(2, 3, 1, 5, 10)
n <- 0









n
# 21
```

繰り返し構文

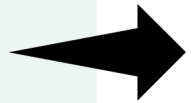
ベクトル a の各要素の合計を計算して、変数 n に代入する手順を考えよ。ただし、一度に 2 数値の足し算しかできないものとする。

```
a <- c(2, 3, 1, 5, 10)
n <- 0



n <- n + 2
n <- n + 3
n <- n + 1
n <- n + 5
n <- n + 10




n
# 21
```



```
a <- c(2, 3, 1, 5, 10)
n <- 0



n <- n + a[1]
n <- n + a[2]
n <- n + a[3]
n <- n + a[4]
n <- n + a[5]

n
# 21
```

i = 1, 2,..., 5 のとき、n <- n + a[i] の繰り返しである。



```
a <- c(2, 3, 1, 5, 10)
n <- 0


for (i in 1:5) {
    n <- n + a[i]
}




n
# 21
```

i を 1, 2, ..., 5 に変化させながら、以下の処理を行う構文。

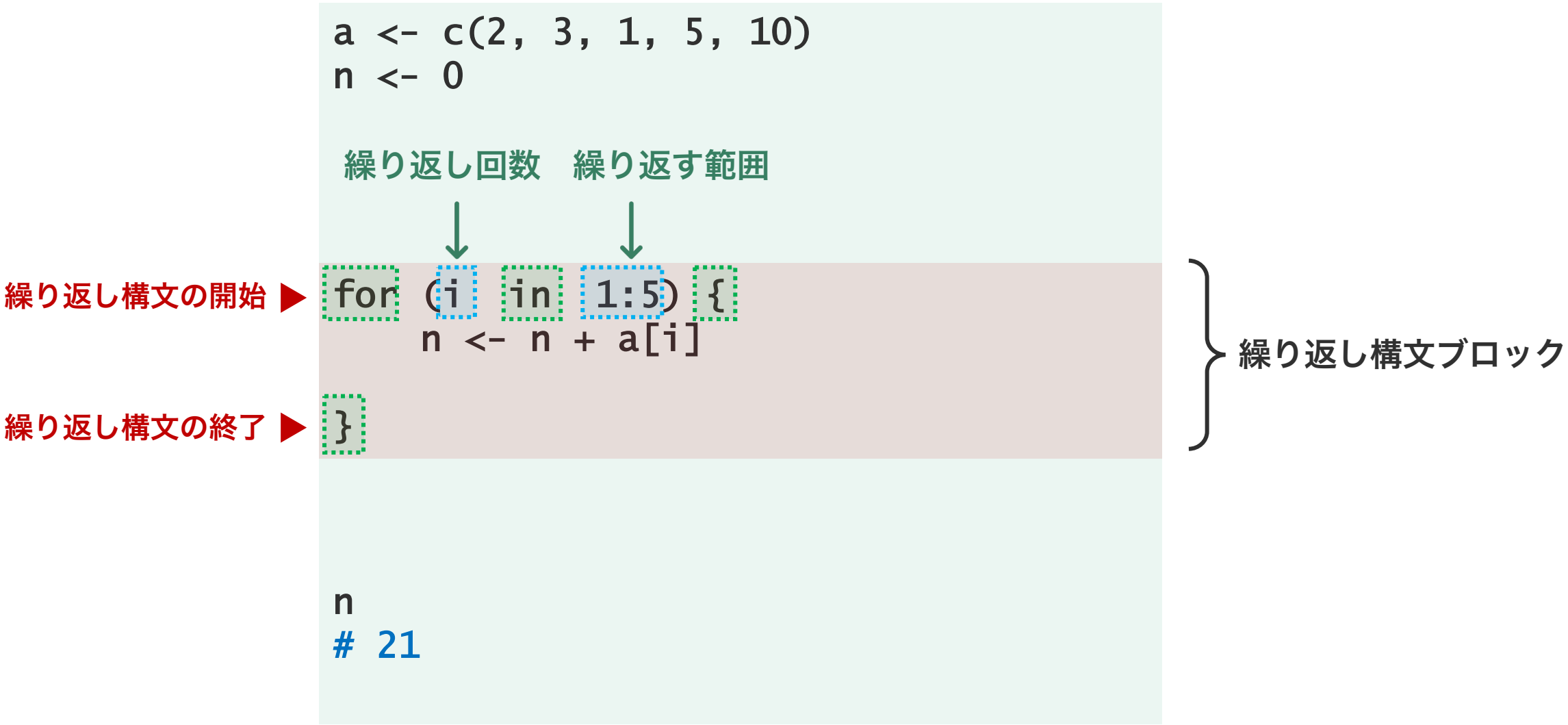
for 構文

繰り返し構文には while 構文と for 構文の 2 種類がある。このうち、for 構文は「n 回繰り返す」命令文である。for 文を使用するとき、繰り返し回数 n をはじめに指定する必要がある。

なお、R は統計計算用に開発されたプログラミング言語である。ベクトルや行列に対する高速演算できるような関数が多数用意されているが、for/while の繰り返し処理が遅いことが知られている。そのため、ベクトルや行列などに対して演算を行うときは、速度の遅い for/while 構文の使用をできるだけ控えること。

```
a <- c(2, 3, 1, 5, 10)
n <- 0

for (i in 1:5) {
    n <- n + a[i]
}
n
# 21



m <- 0

for (i in 1:length(a)) {
    m <- m + a[i]
}
m
# 21
```

for 構文


a <- c(2, 3, 1, 5, 10)
n <- 0
繰り返し回数
繰り返す範囲
繰り返し構文の開始
for (i in 1:5) {
n <- n + a[i]
繰り返し構文の終了
}
繰り返し構文ブロック
n
21

for 構文

```
a <- c(2, 3, 1, 5, 10)
n <- 0

for (x in a) {
    n <- n + x
}

n
# 21
```

一次変数

繰り返す対象

繰り返し構文の開始

繰り返し構文の終了

繰り返し構文ブロック

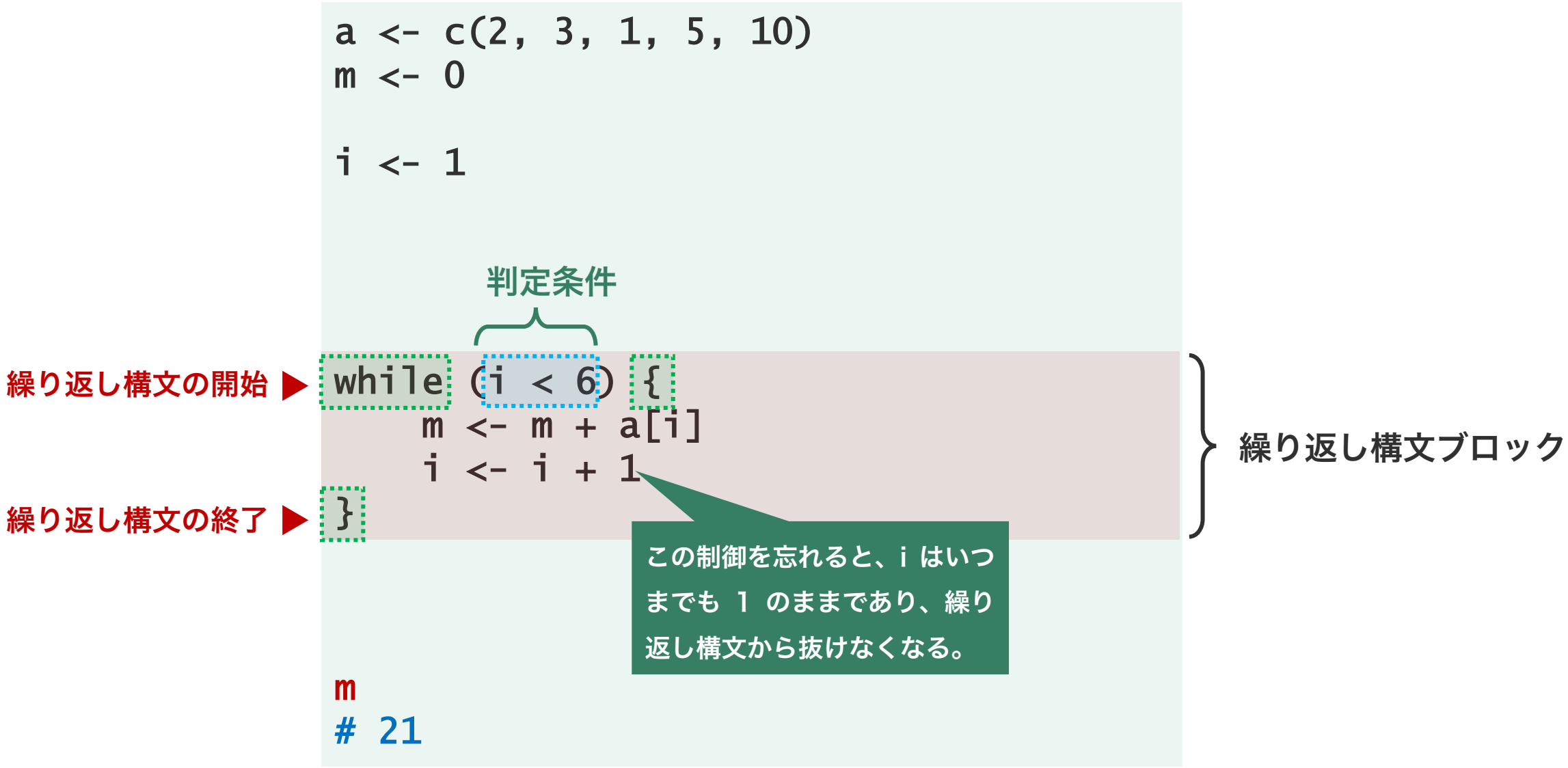
ベクトルの各要素に対して繰り返し処理を行う場合、繰り返す対象としてベクトルの変数名をそのまま与えることもできる。

while 構文

繰り返し構文には while 構文と for 構文の二種類がある。このうち、while 構文は、「与えた条件を満たす限り同じ処理を繰り返す」命令文である。例えば、「ベクトル a の末端に達するまで、a の各要素を n に足す」処理に利用できる。for 構文で書ける処理は while 構文でも書ける。右は、ベクトル a の要素の合計を計算する処理を、for 構文および while 構文で書いた例を示している。

```
a <- c(2, 3, 1, 5, 10)
n <- 0

for (i in 1:5) {
    n <- n + a[i]
}
n
# 21



m <- 0

i <- 1
while (i < 6) {
    m <- m + a[i]
    i <- i + 1
}
m
# 21
```

while 構文


a <- c(2, 3, 1, 5, 10)
m <- 0
i <- 1
判定条件
繰り返し構文の開始
while (i < 6) {
m <- m + a[i]
i <- i + 1
繰り返し構文の終了
}
繰り返し構文ブロック
この制御を忘れると、i はいつまでも 1 のままであり、繰り返し構文から抜けなくなる。
m
21

問題 S2-1

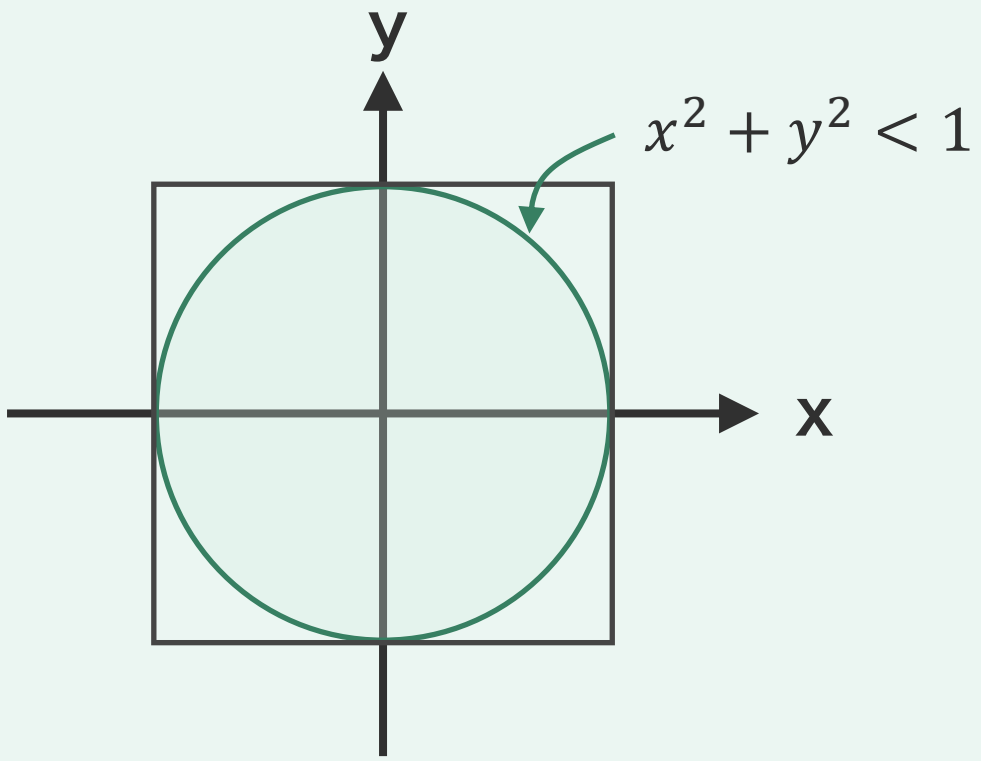
10 min

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

2. (1) の結果を利用して円周率（の近似値）を計算せよ。

3. (1)-(2) を 100 回繰り返して、100 個の円周率（の近似値）を計算し、それらの平均を求めよ。

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



問題 S2-2

以下のプログラムを、for 構文および while 構文を使わない形で書き換えよ。（例えば、apply, ifelse, which などの関数を利用するなど。）

```
a <- c(2, 5, 8, 7, 3, 4, 1)
b <- rep(NA, length = length(a))

for (i in 1:length(a)) {
  if (a[i] %% 2 == 1) {
    b[i] <- TRUE
  } else {
    b[i] <- FALSE
  }
}

x <- a[b]
x
```

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

for (i in 1:length(a)) {
  if (a[i] %% 2 == 0) {
    n <- n + 1
  }
}

n
```

問題 S2-3

以下のプログラムを、for 構文および while 構文を使わない形で書き換えよ。（例えば、apply, ifelse, which などの関数を利用するなど。）

```
a <- matrix(1:9, ncol = 3)
m <- rep(NA, length = ncol(a))

for (i in 1:ncol(a)) {
  m[i] <- mean(a[, i])
}

m
```

```
a <- matrix(1:12, ncol = 3)
m <- rep(0, length = ncol(a))

for (j in 1:ncol(a)) {
  for (i in 1:nrow(a)) {
    if (a[i, j] %% 2 == 0) {
      m[j] <- m[j] + a[i, j]
    }
  }
}
m
```

基本文法

- ◯ 条件構文
- ◯ 繰り返し構文
- ◯ 関数
- ◯ パッケージ

関数

同じ処理を繰り返すとき、その処理をまとめてパッケージ化することができる。このパッケージ化された処理群を関数という。まずは、関数を使わずに、あるベクトルの最大値と最小値の差を計算する例を見ていく。

```
a <- c(11, 2, 8, 7)



M <- max(a)
m <- min(a)
d <- M - m



d
```

関数

同じ処理を繰り返すとき、その処理をまとめてパッケージ化することができる。このパッケージ化された処理群を関数という。まずは、関数を使わずに、あるベクトルの最大値と最小値の差を計算する例を見ていく。

```
a <- c(11, 2, 8, 7)
b <- c(9, 3, 13, 6)




M <- max(b)
m <- min(b)
d <- M - m




d
```

関数

同じ処理を繰り返すとき、その処理をまとめてパッケージ化することができる。このパッケージ化された処理群を関数という。まずは、関数を使わずに、あるベクトルの最大値と最小値の差を計算する例を見ていく。

```
a <- c(11, 2, 8, 7)
b <- c(9, 3, 13, 6)
p <- c(7, 20, 8, 5)



M <- max(p)
m <- min(p)
d <- M - m



d
```

関数

同じ処理を繰り返すとき、その処理をまとめてパッケージ化することができる。このパッケージ化された処理群を関数という。まずは、関数を使わずに、あるベクトルの最大値と最小値の差を計算する例を見ていく。

```
a <- c(11, 2, 8, 7)
b <- c(9, 3, 13, 6)
p <- c(7, 20, 8, 5)
q <- c(12, 9, 2, 5)



M <- max(q)
m <- min(q)
d <- M - m




d
```

関数

これまでの例を見ると、計算対象 a, b, p, q は変化しているが、計算式（M, m, d を計算する式）は変化していない。そこで、変化している部分と変化していない部分を切り分けて、変化していない部分を function 構文でパッケージ化（関数化）していく。

```
a <- c(11, 2, 8, 7)
b <- c(9, 3, 13, 6)
p <- c(7, 20, 8, 5)
q <- c(12, 9, 2, 5)



M <- max(q)
m <- min(q)
d <- M - m



d
```

```
a <- c(11, 2, 8, 7)
b <- c(9, 3, 13, 6)
p <- c(7, 20, 8, 5)
q <- c(12, 9, 2, 5)


mmdiff <- function(x) {
  M <- max(x)
  m <- min(x)
  d <- M - m
  return(d)
}


mmdiff(a)
mmdiff(b)
mmdiff(p)
mmdiff(q)
```

関数

関数の定義の開始 ▶

関数の定義の終了 ▶

```
a <- c(11, 2, 8, 7)

mmdiff <- function(x) {

  M <- max(x)
  m <- min(x)
  d <- M - m

  return(d)
}



mmdiff(a)
```

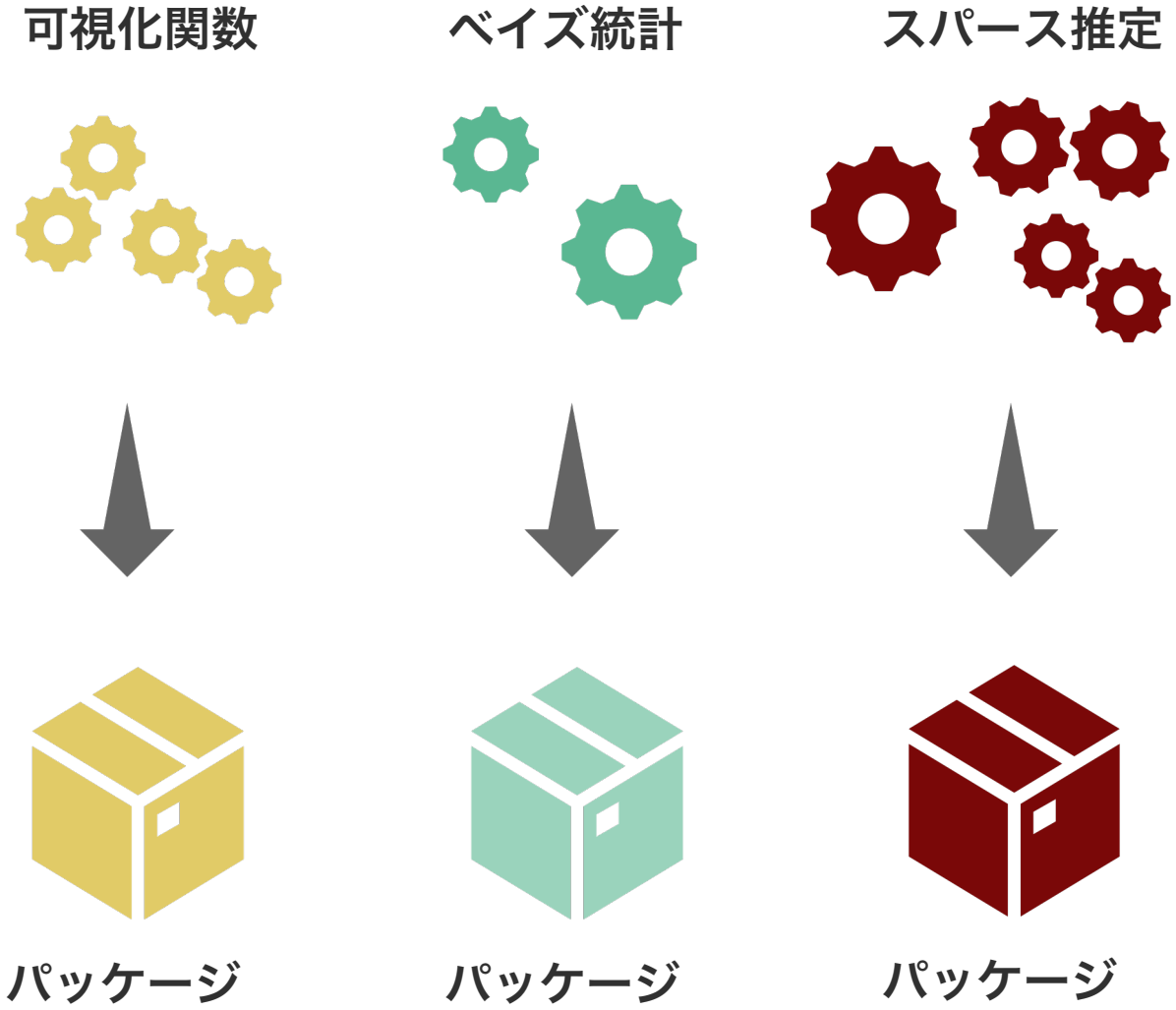
関数ブロック

基本文法

- ◯ 条件構文
- ◯ 繰り返し構文
- ◯ 関数
- ◯ パッケージ

パッケージ

R には便利な関数が多数開発されている。これらの関数は、ひと塊りにまとめられてインターネットから配布されている。これら複数の関数をひとつ塊にまとめたものをパッケージという。



パッケージ

CRAN

Available CRAN Packages By Name

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A3	Accurate, Adaptable, and Accessible Error Metrics for Predictive Models
aaSEA	Amino Acid Substitution Effect Analyser
AATtools	Reliability and Scoring Routines for the Approach-Avoidance Task
ABACUS	Apps Based Activities for Communicating and Understanding Statistics
abbyyR	Access to Abbyy Optical Character Recognition (OCR) API
abc	Tools for Approximate Bayesian Computation (ABC)
abc.data	Data Only: Tools for Approximate Bayesian Computation (ABC)
ABC.RAP	Array Based CpG Region Analysis Pipeline
abcADM	Fit Accumulated Damage Models and Estimate Reliability using ABC
ABCanalysis	Computed ABC Analysis
abcdeFBA	ABCDE_FBA: A-Biologist-Can-Do-Everything of Flux Balance Analysis with this package
ABCoptim	Implementation of Artificial Bee Colony (ABC) Optimization
ABCp2	Approximate Bayesian Computational Model for Estimating P2
abcrf	Approximate Bayesian Computation via Random Forests
abcrlda	Asymptotically Bias-Corrected Regularized Linear Discriminant Analysis
abctools	Tools for ABC Analyses
abd	The Analysis of Biological Data
abdiv	Alpha and Beta Diversity Measures
abe	Augmented Backward Elimination

16,850
packages

Bioconductor

Bioconductor
OPEN SOURCE SOFTWARE FOR BIOINFORMATICS

Search:

Home | Install | Help | Developers | About

About *Bioconductor*

Bioconductor provides tools for the analysis and comprehension of high-throughput genomic data. *Bioconductor* uses the R statistical programming language, and is open source and open development. It has two releases each year, and an active user community. *Bioconductor* is also available as an AMI (Amazon Machine Image) and Docker images.

News

- *Bioconductor* 3.12 is available.
- BiocEurope virtual conference registration and abstract submission open Decemeber 14-18, 2020.
- See our google calendar for events, conferences, meetings, forums, etc. Add

Install »

- Discover 1974 software packages available in *Bioconductor* release 3.12.

Get started with *Bioconductor*

- Install *Bioconductor*
- Get support
- Latest newsletter
- Follow us on twitter
- Install R

Learn »

Master *Bioconductor* tools

- Courses
- Support site
- Package vignettes
- Literature citations
- Common work flows
- FAQ
- Community resources
- Videos

Use »

Create bioinformatic solutions with *Bioconductor*

Develop »

Contribute to *Bioconductor*

- Developer resources

1,974
packages

パッケージのインストール

パッケージのインストールは install.packages 関数や BiocManager::install 関数を使用する。そのほか、GitHub リポジトリからパッケージをインストールする devtools::install_github 関数もある。一般的な統計解析であれば、install.packages 関数のみを把握していればよい。

```
# CRAN packages
install.packages('gplots')



# Bioconductor packages
if (!requireNamespace('BiocManager', quietly = TRUE)) {
  install.packages('BiocManager')
}
BiocManager::install('edgeR')



# GitHub packages
library(devtools)
devtools::install_github('jsun/TCC')
```

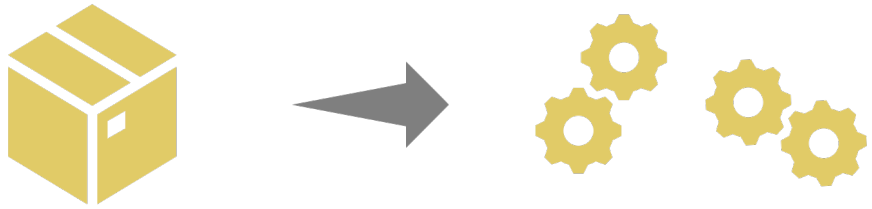
パッケージのインストール

R でデータ解析を行う上で次のような CRAN パッケージがよく使われる。予めインストールしておくとよい。

```
# CRAN packages
install.packages('survival')
install.packages('glmnet')
install.packages('lme4')
install.packages('nlme')
install.packages('gplots')
install.packages('rgl')
install.packages('openxlsx')
```

パッケージ

パッケージを使うとき、library または require 関数で予めパッケージの機能を呼び出してから、パッケージ中の関数を使用する。例えば、基礎的な数学演算用のパッケージ MASS の中の関数を使いたいときは、library(MASS) のようにしてパッケージを呼び出す。パッケージ中の関数を使用する場合は、パッケージ名に続けて :: を書き、そのあとに関数名を書く。ただし、パッケージ名および :: を省略して書くことが多い。



```
library(MASS)


x <- rnorm(1000)
MASS::truehist(x)
truehist(x)


detach(package:MASS)


truehist(x)
# Error in truehist(x) : could not find
function "truehist"


library(MASS)
truehist(x)
```

関数の使い方

R の標準関数やパッケージ中の関数には、使用方法などの情報が付け加えられている。関数に与える引数、戻り値やサンプルコードなどを確認するには help 関数を利用する。例えば、filter 関数の使い方を調べる場合は、右のようにする。

```
help(filter)
```

```
filter                  package:stats                  R Documentation
Linear Filtering on a Time Series
Description:
     Applies linear filtering to a univariate time series or to each
     series separately of a multivariate time series.
Usage:
     filter(x, filter, method = c("convolution", "recursive"),
            sides = 2, circular = FALSE, init)
Arguments:
       x: a univariate or multivariate time series.
  filter: a vector of filter coefficients in reverse time order (as for
          AR or MA coefficients).
Details:
     Missing values are allowed in ‘x’ but not in ‘filter’ (where they
     would lead to missing values everywhere in the output).
Examples:
     x <- 1:100
     filter(x, rep(1, 3))
```