

```
# (2024年版)
```

```
#####  
#####  
##### [BASIC] 基礎 (主にベクトル) #####  
#####  
#####
```

```
#-----  
# [BASIC.1] 変数と代入  
#-----
```

```
# コメントアウトは#を行頭につける
```

```
# 変数aに1を代入  
# イコールも矢印もどちらも使えます。  
a<-1  
a=1  
# Rでは矢印が一般的ですが、この授業ではイコール記法を採用します。
```

```
# 実行する行を選択し、「%RETURN」でコードを実行  
# コンソールに以下のように実行結果が表示されます  
# >[1] 1  
a  
#[1] 1
```

```
# ピリオドを変数の表記に使うことも可 (先頭は不可)  
A.a = 0.4; A.b = 0.8  
A.a; A.b #セミコロンを使うと一行に複数の文を並べることができます。  
#[1] 0.4  
#[1] 0.8
```

```
# 大文字と小文字は区別します  
name = -1; Name = -2  
name+1;Name+1  
#[1] 2  
#[1] 3
```

```
#-----  
# [BASIC.2] 型  
#-----
```

```
#https://qiita.com/maruman029/items/365a2abcdaaf99b720be
```

```
a = 1 #整数はデフォルトではdouble型として扱われます  
b = 0.2 #double型  
c = TRUE; C = T #logical型 (T、F表記も可能)  
d = 1L #整数型 (integer型) として扱いたい時はLをつける  
e = "Hello" #character型
```

```
# データ型はtypeofで調べる  
typeof(a);typeof(b);typeof(c);typeof(d);typeof(e)  
#[1] "double"  
#[1] "double"  
#[1] "logical"  
#[1] "integer"  
#[1] "character"
```

```
# 型変換にはas.xxxを使う  
a = 1; typeof(a)  
#[1] "double"
```

```

b = as.integer(a) #double型からinteger型に変換
b;typeof(b);
#[1] 1
#[1] "integer"
c = as.character(a) #double型からcharacter型に変換 (1は"1"に変換されます)
c;typeof(c)
#[1] "1"
#[1] "character"

# numericはintegerとdoubleを合わせた型 (mode関数で確認可能)
mode(a);mode(b)
#[1] "numeric"
#[1] "numeric"

#-----
# [BASIC.3] 一次元ベクトルの生成
#-----

#-----
##(3.1) 一次元ベクトルを生成する方法
#-----

# c(n1,n2,n3)
# cはcombineに由来

a=c(1,2,3)
#[1] 1 2 3
a=c(T,T,F)
#[1] T T F
a=c("T","T","F") #文字列の配列
#[1] "T" "T" "F"

# c(a:b)で一次元配列ベクトル (a,a+1,,b) を作成
a=c(101:105)
#[1] 101 102 103 104 105
a=c(9:1); # b<aのとき (a,a-1, .. b)
#[1] 9 8 7 6 5 4 3 2 1
a = c(-3:9999333)
#[1] -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21
#[26] 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46
# (以下略)
length(a) #ベクトルの長さはlength関数で取得可能
#[1] 9999337

# c(a,b)の引数が非整数の場合がある。
# 初項a、公差1で、b以下のものを集めたベクトルを返す
a=c(2.3:4)
#[1] 2.3 3.3

# 「:」記法の場合、cは省略可能
a=101:105
#[1] 101 102 103 104 105
a=9:1
#[1] 9 8 7 6 5 4 3 2 1
a=2.3:4
#[1] 2.3 3.3

a = seq(1,10,length=5) #1~10を5分割
#[1] 1.00 3.25 5.50 7.75 10.00
a= seq(1,10,by=2) #1から2ずつ増やして10を越える手前まで
#[1] 1 3 5 7 9

```

```

a = rep(1:3,times=3) #(1,2,3)を3回
#[1] 1 2 3 1 2 3 1 2 3
a = rep(1:3,length=5) #(1,2,3)を要素数5となるまで繰り返す
#[1] 1 2 3 1 2

# 無作為サンプリング関数 (sample) を使う
a = sample(1:6,1) #サイコロを1回振る
#[1] 5
#[1] 3
a = sample(1:6,6) #1~6から無作為サンプルを6回 (重複なし)
#[1] 4 6 5 1 3 2

a = sample(1:6,7) #エラー (母数よりサンプル数が多い)
a = sample(1:6,7,replace = TRUE) #replaceをTRUEにセットすると重複ありモードへ
# [1] 6 1 2 6 5 6 5

# 空のベクトルを作成
a=vector("integer") #integer型の空のベクトルを生成 (中身は空)
#integer(0)
a=vector("logical") #logical型の空のベクトルを生成 (中身は空)
#logical(0)
a=vector("integer",10) #引数2はベクトルの長さ (初期値は0)
#[1] 0 0 0 0 0 0 0 0 0 0
a=vector("logical",10) #logical型の場合、初期値はFALSE
#[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

#-----
##(3.2) アトミックベクトルの性質
#-----

# Rでは、全てのメンバが同じ型を持つタイプのベクトルを
# アトミックベクトルと呼ぶ。

# 後で示すように、
# アトミックベクトルは次元ベクトルとは限りません。

# アトミックベクトルかどうかを確認する
a1=2:5; is.atomic(a1)
#[1] TRUE

# リストはアトミックベクトルでは無い (次週学びます)
# (参考) リストは複数の型を扱います
a2=list(2:5,"R"); is.atomic(a2)
#[1] FALSE

# データフレームはアトミックベクトルでは無い (次週学びます)
a3=data.frame(c(1,2),c("KODAKA","KENRI")); is.atomic(a3)
#[1] FALSE

# (参考) is.vectorは、名前以外の属性を持つかを調べます。
is.vector(a1);is.vector(a2);is.vector(a3)
#[1] TRUE
#[1] TRUE
#[1] FALSE

# アトミックベクトルの型はtypeofで調べる。
a=c(T,T,F); b=c(101:105)
typeof(a); typeof(b)
#[1] "logical"
#[1] "integer"

```

```

# 異なる型を入れた場合、適当な型に型変換されます。
# 以下で、Tと102は文字列型 (character) に型変換されます。
a=c(T,102,"hello"); typeof(a)
#[1] "character"

a[1];a[2];a[3]
#[1] "TRUE"
#[1] "102"
#[1] "hello"

# 数値も要素1のベクトルです。
a=2;is.atomic(a)
#[1] TRUE
a[1]
#[1] 2

#-----
##(3.3) ベクトルの結合
#-----

a1 = c(101:105)
a2 = c(a1,201,202) #a1に2つの要素を追加
#[1] 101 102 103 104 105 201 202

# append関数を使って結合する
a = c(101:105)
b = c(301:305)
append(a,b) #aとbを結合
#[1] 101 102 103 104 105 301 302 303 304 305
append(a,b,after=2) #3番目の要素から挿入
#[1] 101 102 301 302 303 304 305 103 104 105

a1 = c(101:105)
a2 = c(201,a1,202) #a2の2番目の要素にa1を追加
#[1] 201 101 102 103 104 105 202

# この場合a1の要素数は7となり、
# 例えばa1の3番目要素はa2の4番目の要素となります。
a2[4]
#[1] 103

# 多次元配列の様な構造を持つことはできません。
# よって以下の様な書き方をしても、103とはなりません。
a2[2][3]
#[1] NA

# 要するに一次元ベクトルの要素に
# 別の一次元ベクトルを配置すると、
# 新しいサイズの一次元ベクトルが作られます。

a1 = c(101:105)
a2 = c(201,a1,202)
a3 = c(301,a2,302);
#[1] 301 201 101 102 103 104 105 202 302

#-----
##(3.4) ベクトルの要素抽出・置換
#-----

# n番目の要素を取り出す
a=101:105
a[3]
#[1] 103

```

```

# n番目の要素を書き換える
a=101:105; a[3] = 3; a;
#[1] 101 102 3 104 105

a=101:105
a[c(2,4)] #2番目と4番目の要素を取り出してベクトルにする
#[1] 102 104
a[c(-2,-4)] #2番目と4番目の要素を取り除く
#[1] 101 103 105
a[4:8] #範囲外はNAを要素とする
#[1] 104 105 NA NA NA

# 特定の条件を満たす要素のみを取り出す時は
# which()関数で、引数に条件を記述する
a=101:120
which(a-10==105) #10を引いて105となる要素の添字を取り出す
#[1] 15

# 添字が存在しない場合、空の整数ベクトルが返される
which(a-10==120) #10を引いて120となる要素の添字を取り出す
#integer(0)

a[which(a-10==105)] #aの要素のうち、10を引いて105となる要素を取り出す
#[1] 115
a[which(a-10==120)] #aの要素のうち、10を引いて120となる要素を取り出す
#integer(0) #存在しない

# 整数商は「%/%」、割った余りは「%%」
# 例えば、17わる3は、5余り2
17 %/% 3; 17 %% 3
#[1] 5
#[1] 2
# これを使うと、
a=101:125
a[which(a%%2==0)] #偶数の要素のみを取り出す
#[1] 102 104 106 108 110 112 114 116 118 120 122 124
a[which(a%%103<1)] #103で割った余りが1より小さい
#[1] 101 102

#-----
# [BASIC.4] ベクトルの演算
#-----

dice=1:6
#[1] 1 2 3 4 5 6
dice-1
#[1] 0 1 2 3 4 5
dice/2
#[1] 0.5 1.0 1.5 2.0 2.5 3.0
dice * dice
#[1] 1 4 9 16 25 36
dice %% 3 #3で割った余り
#[1] 1 2 0 1 2 0

# (リサイクル規則)
# ベクトルのサイズが異なる場合の演算は、
# 短い方のベクトルの要素が繰り返し使われます。
x = c(2,2,3,3,4,4)
y = c(0.1,0.2)
z = x+y

```

```
#[1] 2.1 2.2 3.1 3.2 4.1 4.2
z = x*y
#[1] 0.2 0.4 0.3 0.6 0.4 0.8
```

```
#-----
# [BASIC.5] ベクトルに適用される関数
#-----
```

```
vec = 1:6
sum(vec) #総和
#[1] 21
mean(vec) #平均
#[1] 3.5
round(mean(vec))#四捨五入
#[1] 4
```

```
# ランダムサンプリング (既出)
```

```
sample(1:4, 2)
#[1] 2 4
#[1] 4 1
```

```
# 引数を明示して関数を実行する
```

```
# (xとsizeは引数の名前として指定)
```

```
sample(x=1:4,size=2)
# 名前が無いもの (1:4) は、xとして扱う
# sampleの場合、xが引数のデフォルト値
sample(size=2,1:4)
```

```
# 関数の引数の名前 (および引数のデフォルト値 + 引数の順序) を調べる
```

```
args(sample)
#function (x, size, replace = FALSE, prob = NULL)
```

```
# replaceは同一の値のサンプリングを許可するか
```

```
sample(1:4, replace=TRUE)
#[1] 1 2 2 3
#[1] 4 3 3 2
```

```
# 名前がなければ、2番目の引数はsizeが期待されるためエラー
```

```
sample(1:4, TRUE)
#Error in sample.int(length(x), size, replace, prob) :
# invalid 'size' argument
```

```
# 誤った引数の名前はエラーとなる
```

```
sample(1:4,saizu=2)
#Error in sample(1:4, saizu = 2) : unused argument (saizu = 2)
```

```
#-----
# [BASIC.6] 配列
#-----
```

```
# アトミックベクトルを2行3列の配列に変換
```

```
a = array(1:6,dim=c(2,3))
#[,1] [,2] [,3]
#[1,] 1 3 5
#[2,] 2 4 6
```

```
is.array(a) #aは配列
```

```
#[1] TRUE
```

```
is.atomic(a) #aは変わらずアトミックベクトル
```

```
#[1] TRUE
```

```

length(a) #長さは6
#[1] 6

a[2,1] #2行1列目の要素
#[1] 2
a[2,3] #2行3列目の要素
#[1] 6

a[4] #a[2,2]と同じ
#[1] 4
a[5] #a[1,3]と同じ
#[1] 5

a[2,] #2行の要素群
#[1] 2 4 6
a[,3] #3列の要素群
#[1] 5 6

a[,c(1,3)] #1列目と3列目を切り出す (配列)
#[,1] [,2]
#[1,] 1 5
#[2,] 2 6

# コマンドの複製 (replicate)

a = replicate(3, 1+1) #1+1の結果を3回繰り返しベクトルへ
#[1] 2 2 2
a = replicate(10, sample(1:6,1)) #サイコロの無作為抽出を10回繰り返す
#[1] 4 3 4 6 3 3 5 2 2 3
is.array(a) #aは配列ではなく、単なる (一次元) ベクトル
#[1] FALSE

# 要素数n (>1) のベクトルを返す関数をm回replicateすると、
# 結果はn x mの配列となる
a = replicate(10, sample(1:6, size=2))
#[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#[1,] 2 5 4 2 2 1 6 3 1 6
#[2,] 1 1 6 6 1 2 2 1 2 2
is.array(a) #aは配列
#[1] TRUE
is.atomic(a) #aは変わらずアトムベクトル
#[1] TRUE
length(a) #長さは20 (2x10)
#[1] 20

#-----
# [BASIC.7] 繰り返し文・条件文
#-----

# 1:6に含まれる個々の要素をaとし、
# aを出力する (print関数)
for(a in 1:6){
  print(a)
}
#[1] 1
#[1] 2
#[1] 3
#[1] 4
#[1] 5
#[1] 6

```

```

# for(a in vec)のaは必ずしも使う必要はない
# 以下はサイコロの無作為抽出を5回繰り返したものの加算を計算したもの
sum = 0
for(a in 1:5){
  sum = sum + sample(1:6,1) #+=の記法はエラー（なぜ？）
  print(sum)
}
#[1] 2
#[1] 7
#[1] 10
#[1] 15
#[1] 18

# サイコロの総和が50以上となるまで加算
sum=0
while(sum<50){
  sum = sum + sample(1:6,1)
  print(sum)
}
#[1] 3
#[1] 5
#[1] 7
#中略
#[1] 35
#[1] 41
#[1] 45
#[1] 50

# repeat関数はwhile(true)に相当
# breakが実行されるまで永遠に実行する

sum = 0
repeat{
  sum = sum + sample(1:6,1)
  if(sum>50){
    break
  }
  print(sum)
}

#[1] 4
#[1] 10
#[1] 13
#中略
#[1] 41
#[1] 46
#[1] 49

#-----
# [BASIC.8] 関数定義（独自の関数をつくる）
#-----

# 2個のサイコロを振って、総和を得るプログラム
dice = 1:6
dice2 = sample(dice, size=2, replace = TRUE)
sum(dice2)

# これをroll()として関数化する
roll = function(){
  dice = 1:6
  dice2 = sample(dice, size=2, replace = TRUE)
  sum(dice2)
}

# 関数定義の最後の行が返り値となる（returnは使わないことに注意！！）
for(i in 1:3){

```



```

    ret = roll()
    print(ret)
}
#[1] 9
#[1] 10
#[1] 6

# n個のサイコロを振って、総和を得るプログラム
# 引数ありの関数をつくります。
roll.n = function(n){
  dice = 1:6
  dice2 = sample(dice, size=n, replace = TRUE)
  sum(dice2)
}

rolls=vector(mode="integer",length=1000)

result = vector("integer",10) #サイズ10の全ての要素がゼロのベクトル
result = vector(mode="integer",length=10) #同じです
for(i in 1:10){
  result[i] = roll.n(i)
}
result
#[1] 6 8 14 13 19 16 22 30 39 38 # (EXAMPLE)

#roll関数を10000回繰り返して、一次元ベクトルに格納
rolls = replicate(1000, roll())
#[1] 6 7 9 6 9 4 3 9 8 4 7 10 8 8 7 9 8 10 7 8 8 6
#[23] 4 12 5 5 12 7 10 6 9 5 5 5 11 4 12 7 8 10 7 11 7 6
#[45] 5 12 8 11 10 7 5 11 9 5 9 7 5 4 8 6 10 9 5 6 9 9
#...
#[991] 6 6 7 7 10 10 5 9 9 12

#for文で書き直しています（同じことです）。
rolls=vector("integer",1000) #integer型の空のベクトルを生成
for(i in 1:1000){
  rolls[i] = roll()
}

#ヒストグラムの計算（roll関数を使います）
imax = 10000
rolls = replicate(imax, roll())

hi=vector("integer",12) #中身が0の長さ12のベクトルをつくる
for(i in 1:imax){
  hi[rolls[i]] = hi[rolls[i]]+1
}
#[1] 0 28 53 85 93 140 180 146 112 89 54 20

# 棒グラフで可視化（barplot）
barplot(hi, main="Double dice",names.arg=1:12, ylab="Frequency")
# mainはタイトル
# names.argはx軸の値
# ylabはy軸のラベル

# hist関数を使うと、サンプルベクトルから自動でヒストグラムを生成できる
hist(rolls,breaks=0:13,xlim=c(0,12),ylim=c(0,2000))
# breaksは区切りのベクトル
# xlim/ylimはx/yの範囲

```