ミクロ政治データ分析実習

第5回 ベクトルとデータ型

そん じぇひょん
宋 財 泫
関西大学総合情報学部
2021/5/13 (updated: 2021-05-06)

データ型とは

データ型とは

- **データ型（Data Type）**
  - データ構造（Data Structure）とは別概念: 次週の内容
- Rにおけるデータの最小単位はベクトル（vector）

```
Object1 <- c(1, 3, 5, 7, 9, 11)
Object2 <- c("Kansai", "Kwansei Gakuin", "Doshisha", "Ritsumeikan")
Object3 <- c(TRUE, FALSE, FALSE)
Object4 <- "R"
```

- `Object1`: 長さ6の**数値型**ベクトル
- `Object2`: 長さ4の**文字型**ベクトル
- `Object3`: 長さ3の**論理型**ベクトル
- `Object4`: 長さ1の**文字型**ベクトル
  - 長さ1のベクトルは原子ベクトル（atomic vector）とも呼ばれる
- データ型: 数値型、文字型、論理型
- データ構造: ベクトル

データ型の確認

- `class(オブジェクト名)`関数

```
class(Object1) # 数値型 (numeric)
```

```
## [1] "numeric"
```

```
class(Object2) # 文字型 (character)
```

```
## [1] "character"
```

```
class(Object3) # 論理型 (logical)
```

```
## [1] "logical"
```

```
class(Object4) # 文字型 (character)
```

```
## [1] "character"
```

ベクトルについて

- Rにおけるデータの最小単位
- `class()`はデータ**構造**を調べる関数であるが、**オブジェクトがベクトルである場合、データ型を出力**する。
- ベクトルの長さは1以上、全て同じ**データ型**

```
# 一つのベクトルにnumericとcharacterが混在していると...
Object5 <- c("R", "Python", 3.5, "SPSS")
Object5 # 3.5が"で囲まれている（character型へ自動変換）
```

```
## [1] "R"      "Python" "3.5"    "SPSS"
```

```
class(Object5) # データ型を確認する
```

```
## [1] "character"
```

- 一つのベクトル内に複数のデータ型があると...
  - character > numeric > logical
  - 他にもいくつかのデータ型がある

全てはベクトル

行列の例

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    5   11   17   23
## [2,]    3    7   13   19   29
```

2行3列: 長さ1のnumeric型ベクトル（`13`）

1行: 長さ5のnumeric型ベクトル（`c(3, 7, 13, 19, 29)`）

5列: 長さ2のnumeric型ベクトル（`c(23, 29)`）

- 長さ1以上のnumeric型ベクトルを2次元に配置したデータ構造

全てはベクトル

データフレーム

最も広く使われる表形式データ構造

```
##   ID           Name  Pref Students
## 1  1         Kansai Osaka    30141
## 2  2 Kwansei Gakuin Hyogo    24569
## 3  3       Doshisha Kyoto    30602
## 4  4    Ritsumeikan Kyoto    35113
```

- 1列: 長さ4のnumeric型ベクトル
  - `c(1, 2, 3, 4)`
- 2列: 長さ4のcharacter型ベクトル
  - `c("Kansai", "Kwansei Gakuin", "Doshisha", "Ritsumeikan")`
- 3列: 長さ4のcharacter型ベクトル
  - `c("Osaka", "Hyogo", "Kyoto", "Kyoto")`
- 4列: 長さ4のnumeric型ベクトル
  - `c(30141, 24569, 30602, 35113)`
- データフレームは同じ長さのベクトルを横に並べたデータ構造

よく使うデータ型＆データ構造

よく使うデータ型

以下のリストはRが提供するデータ型の一部

- Logical
- Numeric
- Complex
- Character
- Factor
- Date
- NA
- NULL
- NaN
- Inf
- その他

(参考)データ構造

ベクトル以外のデータ構造は次週解説

- **ベクトル** (vector)
- 行列 (matrix)
- データフレーム (data frame)
- リスト (list)
- 配列 (array)

今週の内容

使用するデータ構造はベクトルのみ

- ベクトル以外のデータ構造については次週解説

取り上げるデータ型

- Logical
- Numeric & Complex
- Character
- Factor
- Date
- NA, NULL, NaN, Inf

Logical

Logical型とは

`TRUE`と`FALSE`のみで構成されたデータ型

- 第3回に登場

```
(2 + 3) == (4 + 1)
```

```
## [1] TRUE
```

データ型の確認

- ベクトルのデータ型を確認するときには`class()`関数を使用

```
Logical1 <- (2 + 3) == (4 + 1)
class(Logical1)
```

```
## [1] "logical"
```

```
# オブジェクトとして格納せずに、直接確認も可能
class((2 + 3) == (4 + 1))
```

```
## [1] "logical"
```

- Logical型か否かを判定するには`is.logical()`を使用

```
is.logical(FALSE)
```

```
## [1] TRUE
```

Logical型ベクトルの作成

`c()`関数で作成

```
Logical_Vec1 <- c(TRUE, FALSE, TRUE, TRUE, FALSE)
Logical_Vec1
```

```
## [1]  TRUE FALSE  TRUE  TRUE FALSE
```

`TRUE`は`T`、`FALSE`は`F`と略すことが出来るが**非推奨**

```
Logical_Vec2 <- c(T, F, T, T, F)
Logical_Vec2
```

```
## [1]  TRUE FALSE  TRUE  TRUE FALSE
```

Logical型ベクトル作成時の注意点

Logical型は`"`で囲まない。

- 一つでも`"`で囲むとLogical型でなく、Character型へ変換される

```
# 2つ目のFALSEを"で囲むと...
Logical_Vec3 <- c(TRUE, "FALSE", TRUE, TRUE, FALSE)
Logical_Vec3
```

```
## [1] "TRUE"  "FALSE" "TRUE"  "TRUE"  "FALSE"
```

```
class(Logical_Vec3)
```

```
## [1] "character"
```

Logical型ベクトルの使い方

- 直接`TRUE`や`FALSE`が格納されたベクトルを使う場面はほとんどない
- 何かの**演算の結果から返されるlogical型ベクトル**を使用

```
My_Vector1  <- c(89, 28, 93, 64, 6)
My_Logical1 <- My_Vector1 %% 2 == 1 # My_Vector1を2で割ったら余りが1か
My_Logical1
```

```
## [1]  TRUE FALSE  TRUE FALSE FALSE
```

```
class(My_Logical1)
```

```
## [1] "logical"
```

```
My_Vector1[My_Logical1] # My_Vector1から奇数のみ抽出
```

```
## [1] 89 93
```

```
My_Vector1[My_Vector1 %% 2 == 1] # 直接式を入れてもOK
```

```
## [1] 89 93
```

Numeric & Complex

Numeric型とComplex型

Numeric型: 数値型

```
Numeric_Vec1 <- c(2, 0, 0, 1, 3)
Numeric_Vec1
```

```
## [1] 2 0 0 1 3
```

```
# Numeric_Vec1のデータ型
class(Numeric_Vec1)
```

```
## [1] "numeric"
```

```
# Numeric_Vec1がNumeric型か否かを判定
is.numeric(Numeric_Vec1)
```

```
## [1] TRUE
```

Complex型: 複素数型

```
Complex_Vec1 <- c(1+3i, 3+2i, 2.5+7i
Complex_Vec1
```

```
## [1] 1.0+3i 3.0+2i 2.5+7i
```

```
# Complex_Vec1のデータ型
class(Complex_Vec1)
```

```
## [1] "complex"
```

```
# Complex_Vec1がComplex型か否かを判定
is.complex(Complex_Vec1)
```

```
## [1] TRUE
```

Numeric型の演算（１）

Case1: **同じ長さのベクトル**同士の演算

```
Numeric_Vec2 <- c(1, 2, 3, 4, 5)  # 長さ5のnumeric型ベクトル
Numeric_Vec3 <- c(11, 7, 5, 3, 2) # 長さ5のnumeric型ベクトル
```

```
Numeric_Vec2 + Numeric_Vec3 # c(1+11, 2+7, 3+5, 4+3, 5+2)
```

```
## [1] 12  9  8  7  7
```

```
Numeric_Vec2^Numeric_Vec3 # c(1^11, 2^7, 3^5, 4^3, 5^2)
```

```
## [1]   1 128 243  64  25
```

- 同じ長さのベクトル同士の演算の場合、**同じ位置の要素**同士で演算を行う。

Numeric型の演算（２）

Case2: **長さ2以上 (A)**と**長さ1 (B)**同士の演算

```
Numeric_Vec4 <- c(10) # 長さ1の場合、c()はなくてもOK
```

```
Numeric_Vec3 * Numeric_Vec4 # c(11*10, 7*10, 5*10, 3*10, 2*10)
```

```
## [1] 110  70  50  30  20
```

```
Numeric_Vec3 / Numeric_Vec4 # c(11/10, 7/10, 5/10, 3/10, 2/10)
```

```
## [1] 1.1 0.7 0.5 0.3 0.2
```

- **(A)のそれぞれ要素と(B)の要素**同士で演算を行う
  - `c(10)`が自動的に`c(10, 10, 10, 10, 10)`へ変換されたと考えてもOK

Numeric型の演算（2）

Case3: 長さ2以上と長さ2以上で長さが異なる場合

```
Numeric_Vec5 <- c(1, 2, 3)
```

```
Numeric_Vec3 * Numeric_Vec5 # c(11*1, 7*2, 5*3, 3*1, 2*2)
```

```
## [1] 11 14 15  3  4
```

```
Numeric_Vec3 / Numeric_Vec5 # c(11/1, 7/2, 5/3, 3/1, 2/2)
```

```
## [1] 11.000000  3.500000  1.666667  3.000000  1.000000
```

- より短いの要素がリサイクルされる
  - `c(1, 2, 3)`が自動的に`c(1, 2, 3, 1, 2)`へ変換されたと考えてもOK
  - 警告が表示される場合もあるが、演算には問題なし（長さが倍数になっていない場合など）
  - ベクトル・リサイクル（vector recycle）

Numeric型のタイプ（１）

Numeric型はDobule型（実数）とInteger型（整数）に分けられる。

- `typeof()`関数で確認可能（`class()`ではどれも`"numeric"`）

```
typeof(2)
```

```
## [1] "double"
```

```
typeof(4.5)
```

```
## [1] "double"
```

整数型にするためには`L`を付ける

```
typeof(2L)
```

```
## [1] "integer"
```

小数点付きの実数に`L`を付けると、強制的にDouble型に

```
typeof(4.5L)
```

```
## [1] "double"
```

Numeric型のタイプ（2）

Integer +/-/* Integer = Integer

```
typeof(3L + 5L) # Int + Int
```

```
## [1] "integer"
```

```
typeof(3L - 5L) # Int - Int
```

```
## [1] "integer"
```

```
typeof(3L * 5L) # Int * Int
```

```
## [1] "integer"
```

Integer / Integer = Double

```
typeof(3L / 5L) # Int / Int
```

```
## [1] "double"
```

演算に一つ以上のDouble型があると、結果はDouble型

```
typeof(3  + 5L) # Dbl + Int
```

```
## [1] "double"
```

```
typeof(3L - 5)  # Int - Dbl
```

```
## [1] "double"
```

```
typeof(3  * 5L) # Dbl * Int
```

```
## [1] "double"
```

```
typeof(3L / 5)  # Int / Dbl
```

```
## [1] "double"
```

Numeric型ベクトル作成時の注意点

Numeric型は`"`で囲まない。

- 一つでも`"`で囲むとNumeric型でなく、Character型へ変換される

```
# 4つ目のFALSEを"で囲むと...
Numeric_Vec6 <- c(38, 29, 10, "94", 51)
Numeric_Vec6
```

```
## [1] "38" "29" "10" "94" "51"
```

```
class(Numeric_Vec6)
```

```
## [1] "character"
```

Character & Factor

Character型

要素が`"`で囲まれたデータ型

- 文字列型

```
Char_Vec1 <- c("Kansai", "Kwansei-gakuin", "Doshisha", "Ritsmeikan")
Char_Vec1
```

```
## [1] "Kansai"         "Kwansei-gakuin" "Doshisha"       "Ritsmeikan"
```

```
class(Char_Vec1)
```

```
## [1] "character"
```

```
is.character(Char_Vec1)
```

```
## [1] TRUE
```

文字列の長さ

- `length()`は**ベクトルの長さ**を求める関数

```
length(Char_Vec1)
```

```
## [1] 4
```

- `nchar()`は**各要素の文字数**を求める関数

```
nchar(Char_Vec1)
```

```
## [1]  6 14  8 10
```

文字の結合: paste()

- `paste(Character型, Character型)`

ケース1: `Char_Vec1` の全要素の後に `"University"` を付ける

```
Char_Vec2 <- paste(Char_Vec1, "University")
Char_Vec2 # "University"の前に自動的にスペースが入る
```

```
## [1] "Kansai University"          "Kwansei-gakuin University"
## [3] "Doshisha University"        "Ritsmeikan University"
```

ケース2: `Char_Vec2` の全要素の前に `1`、`2`、...を付け、数字と大学名は `"."` で結合

- `sep =` で結合される要素間に入る文字を指定（デフォルトはスペース）

```
Char_Vec3 <- paste(1:4, Char_Vec2, sep = ".")
Char_Vec3
```

```
## [1] "1.Kansai University"          "2.Kwansei-gakuin University"
## [3] "3.Doshisha University"        "4.Ritsmeikan University"
```

文字の結合: paste0()

`paste()`は結合される文字列の間にスペースが自動的に入り、なくすためには`sep = ""`を指定する必要がある

- `paste0()`はスペース無しで結合する関数

ケース3: `Char_Vec1`の全要素の後に`"-Daigaku"`を付ける

```
Char_Vec4 <- paste0(Char_Vec1, "-Daigaku")
# "-Daigaku"の前にスペースがないことに注目
Char_Vec4
```

```
## [1] "Kansai-Daigaku"          "Kwansei-gakuin-Daigaku"
## [3] "Doshisha-Daigaku"        "Ritsmeikan-Daigaku"
```

Factor型

順序付きの文字列型 / ラベル付き数値型

- Character型だと、アルファベット順となる
- 図表を作成する際に重宝されるデータ型
  - 図表上の表示順番はFactor型でなくとアルファベット順となる
- 詳細はデータハンドリングおよび可視化で解説

Factor型の場合

Prefecture	Population
Hokkaido	5,383,579
Aomori	1,308,649
...	...
Kagoshima	1,648,752
Okinawa	1,434,138

Character型の場合

Prefecture	Population
Aichi	7,484,094
Akita	1,022,839
...	...
Yamaguchi	1,405,007
Yamanashi	835,165

Factor型の作成

既存のCharacter型ベクトルをFactor型に

```
Kankandoritsu <- c("Doshisha", "Kansai", "Kwansei-gakuin", "Ritsumeikan")
class(Kankandoritsu)
```

```
## [1] "character"
```

```
Kankandoritsu
```

```
## [1] "Doshisha"       "Kansai"         "Kwansei-gakuin" "Ritsumeikan"
```

```
Kankandoritsu <- factor(Kankandoritsu,
                        levels = c("Kansai", "Kwansei-gakuin",
                                   "Doshisha", "Ritsumeikan"))
class(Kankandoritsu)
```

```
## [1] "factor"
```

```
Kankandoritsu
```

```
## [1] Doshisha       Kansai         Kwansei-gakuin Ritsumeikan
## Levels: Kansai Kwansei-gakuin Doshisha Ritsumeikan
```

Factor型の詳細

- 図表作成時に頻繁に使われるため、詳細は第9回以降から

Date

Date型とは

日付、または日時を表すデータ型

- `Date`型: 日付（例: 2021-05-06）
- `POSIXct`型/`POSIXlt`型: 日時（例: 2021-05-06 17:12:54）
  - `POSIXct`型: 基準日 (1970年1月1日 00時00分00秒 = UNIX時間)からの符号付き経過秒数。パソコンにとって読みやすい（足し算、引き算などが可能）。
  - `POSIXlt`型: 日付、時間がそれぞれ格納されているデータ型。人間にとって読みやすい。
- 本講義（+後期の「マクロ政治データ分析実習」）では扱う機会がほぼない
  - 時系列分析に興味があれば`Date`型の理解は必須
  - 時系列データ処理に特化した{lubridate}パッケージもある(解説資料)

Date型ベクトルの作成

`as.Date()`関数を使用し、`Character`型から`Date`型へ変換

```
Date_Char_Vec1 <- c("1986-10-09", "2021-05-06")
Date_Vec1      <- as.Date(Date_Char_Vec1)
Date_Vec1
```

```
## [1] "1986-10-09" "2021-05-06"
```

```
class(Date_Vec1)
```

```
## [1] "Date"
```

`"1986-10-09"`も`"1986/10/9"`もRが自動的に`Date`型に変換

```
Date_Char_Vec2 <- c("1986/10/09", "2021/5/6")
Date_Vec2      <- as.Date(Date_Char_Vec2)
Date_Vec2
```

```
## [1] "1986-10-09" "2021-05-06"
```

特殊な形式の変換

`"1986年10月09日"`は変換できない

```
Date_Char_Vec3 <- c("1986年10月09日", "2021年05月06日")
as.Date(Date_Char_Vec3)
```

```
## Error in charToDate(x): character string is not in a standard unambiguous form
```

一般的な書き方でない場合は、直接形式を指定する必要がある

- 詳細はコンソール上で`?strptime`を入力

```
Date_Vec3 <- as.Date(Date_Char_Vec3, "%Y年%m月%d日")
Date_Vec3
```

```
## [1] "1986-10-09" "2021-05-06"
```

NA, NULL, NaN, Inf

NA, NULL, NaN, Inf

データ分析において頻繁に遭遇するのは`NA`

- `NA`: 何らかの値があるはずだが、欠損している状態
  - 「**欠損値**」とも呼ばれる
  - 多くのデータには欠損値が含まれているため、欠損値処理は非常に重要
- `NULL`: そもそも存在しない
- `NaN`: 計算不可（例: $0 \div 0$）
- `Inf`: 無限大（例: $10 \div 0$）

`NaN`と`Inf`のベクトルを作ることも可能であるが、使う機会はない

- 計算された結果として`NaN`と`Inf`の意味が分かれば問題なし

NA と NULL の違い

`NA` は要素扱いだが、`NULL` はカウントされない

```
NA_Vec   <- c(1, 2, 3, NA, 5, NA, 7)
NULL_Vec <- c(1, 2, 3, NULL, 5, NULL, 7)
length(NA_Vec)
```

```
## [1] 7
```

```
length(NULL_Vec)
```

```
## [1] 5
```

```
NA_Vec
```

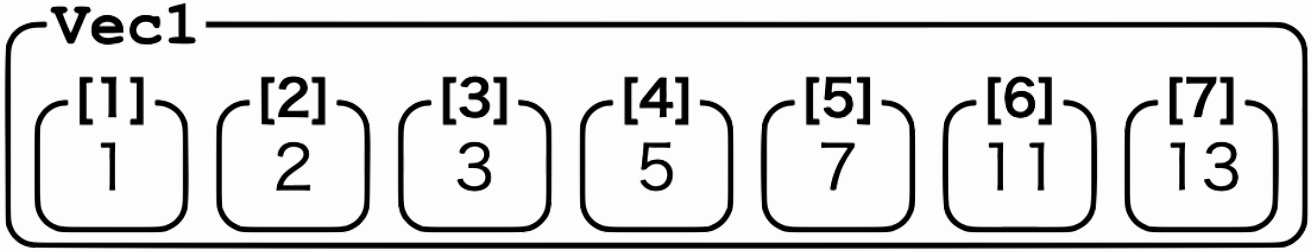
```
## [1]  1  2  3 NA  5 NA  7
```

```
NULL_Vec
```

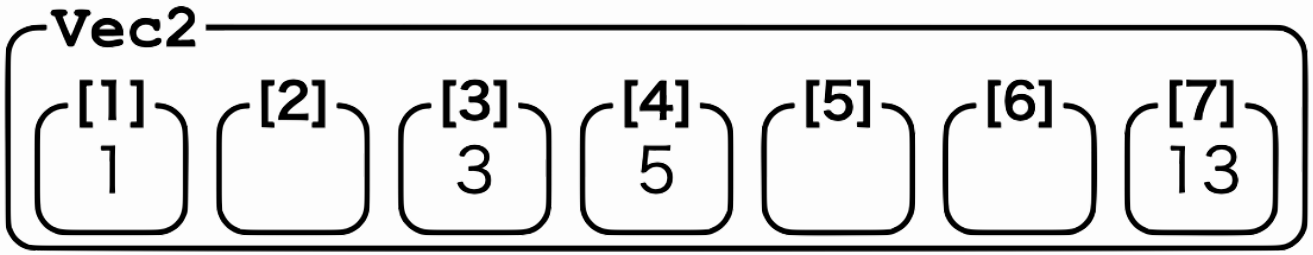
```
## [1] 1 2 3 5 7
```

NAとNULLの違い

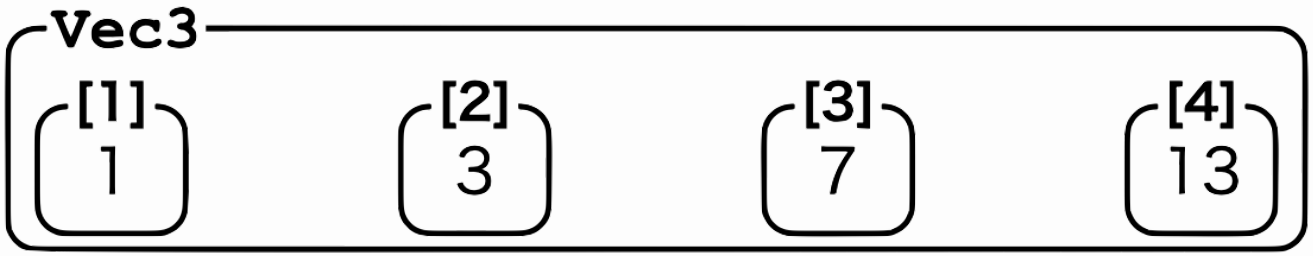
```
Vec1 <- c(1, 2, 3, 5, 7, 11, 13)
```




```
Vec2 <- c(1, NA, 3, 5, NA, NA, 13)
```




```
Vec3 <- c(1, NULL, 3, NULL, 7, NULL, 13)
```



欠損値を含むベクトルの計算

ベクトルに欠損値が含まれている場合、平均値（`mean()`）、標準偏差（`sd()`）などの計算ができないことに注意

- `NA`が含まれているベクトルの平均値、標準偏差、分散などは`NA`と出力される
- `length()`などは機能する

```
mean(NA_Vec)
```

```
## [1] NA
```

欠損値を含むベクトルの計算（1）

- **方法1:** データから欠損値を除外する
  - `!`演算子は否定を意味する（第3回参照）
  - ちなみに、`NA_Vec == NA`、`NA_Vec != NA`は使用不可

```
is.na(NA_Vec) # 各要素がNAか否かを判定
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
```

```
!is.na(NA_Vec) # 各要素がNAか否かの判定を反転
```

```
## [1]  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
```

```
NA_Vec[!is.na(NA_Vec)] # 欠損値でない要素のみ抽出
```

```
## [1] 1 2 3 5 7
```

```
mean(NA_Vec[!is.na(NA_Vec)])
```

```
## [1] 3.6
```

欠損値を含むベクトルの計算（2）

- **方法2:** 関数内に`na.rm = TRUE`を追加する
  - `na.rm`は`mean()`関数の仮引数（かりひきすう）であり、`TRUE`は仮引数と呼ぶ。合わせて引数（じつひきすう）（ひきすう）
  - 通常、関数には様々な引数が用意されている。Rコンソール上で`?関数名`を入力するとヘルプが読める（例: `?mean`）

```
mean(NA_Vec, na.rm = TRUE)
```

```
## [1] 3.6
```

ベクトル操作

ベクトルの長さ（復習）

`length(ベクトル名)`: ベクトルの長さを返す関数

```
My_Vector1 <- c(-6, 94, -4, 49, 27, 33, -94)
My_Vector2 <- c("Ramen", "Soba", "Udon", "Pasta")
length(My_Vector1)
```

```
## [1] 7
```

```
length(My_Vector2)
```

```
## [1] 4
```

ベクトル要素の抽出（復習）

`ベクトル名[n]`：ベクトルから`n`番目の要素を抽出

```
My_Vector1[6] # 6番目の要素を抽出
```

```
## [1] 33
```

```
My_Vector2[2:4] # 2:4はc(2, 3, 4)と同じ
```

```
## [1] "Soba"  "Udon"  "Pasta"
```

```
My_Vector1[c(1, 4)] # 抽出する番号のベクトルを使用
```

```
## [1] -6 49
```

```
My_Vector2[c(TRUE, FALSE, TRUE, TRUE)] # Logicalベクトルで抽出
```

```
## [1] "Ramen" "Udon"  "Pasta"
```

要素の置換

使い方: `ベクトル名[位置] <- 新しい要素`

ケース1: `My_Vector2`の4番目の要素を`"Kishimen"`に

```
My_Vector2[4] <- "Kishimen"
My_Vector2
```

```
## [1] "Ramen"    "Soba"     "Udon"     "Kishimen"
```

ケース2: `My_Vector1`の3〜5番目の要素を11、-5、-72に

```
My_Vector1[3:5] <- c(11, -5, -72)
My_Vector1
```

```
## [1]  -6  94  11  -5 -72  33 -94
```

ケース3: `My_Vector1`の1、4、5、7番目の要素に-1をかける

```
My_Vector1[c(1, 4, 5, 7)] <- My_Vector1[c(1, 4, 5, 7)] * -1
My_Vector1
```

```
## [1]  6 94 11  5 72 33 94
```

要素の追加

要素の置換と同じ方法

ケース1: `My_Vector2`の5、6番目の要素として`"Somen"`と`"Shirataki"`を追加

```
My_Vector2[5:6] <- c("Somen", "Shirataki")
My_Vector2
```

```
## [1] "Ramen"     "Soba"      "Udon"      "Kishimen"  "Somen"     "Shirataki"
```

ケース2: `My_Vector1`の2番目の要素を100に置換し、8〜10番目要素として13、10、0を追加

```
My_Vector1[c(2, 8:10)] <- c(100, 13, 10, 0)
My_Vector1
```

```
##  [1]   6 100  11   5  72  33  94  13  10   0
```

まとめ

今回の内容

よく分からない箇所は教科書を読み返す or 宋&TAに質問 (できれば、LMSの質問コーナーで)

- ベクトルの操作: 教科書第7章
- データ型: 教科書第9章

課題

R Markdownを利用すること

1. 今回講義用のプロジェクトを作成する。
2. 関大LMSから問題（`.Rmd`）とサンプル（`.html`）をダウンロードする
3. サンプル（`.html`）と同じ結果が得られるように、`.Rmd`ファイルを編集する。
4. 編集済みの`.Rmd`ファイルをLMSに提出（`.html`は不要）
5. **期限は2021年5月14日（講義日の翌日）の23時59分とする。**
6. 答案は次回の講義までに公開する。