ミクロ政治データ分析実習

第7回 Rプログラミングの基礎

そん じぇひょん
宋 財 泫
関西大学総合情報学部
2021/5/27 (updated: 2021-05-17)

R言語の基礎概念

オブジェクト

オブジェクト (object) : メモリに割り当てられた「何か」

- ベクトル (vector)、行列 (matrix)、データフレーム (data frame)、リスト (list)、関数 (function) など
- それぞれ固有の（=他のオブジェクトと重複しない）名前が付いている。
- 1から5までの自然数の数列を`my_vec1`という名前のオブジェクトとして格納

```
my_vec1 <- c(1, 2, 3, 4, 5) # my_vec1 <- 1:5  も同じ
```

- **Rに存在するあらゆるものはオブジェクトである** (Chambers, 2016)
  - Everything that exists in R is an object

```
my_vec1 * 2
```

- 以下の例の場合、`2`もオブジェクト
  - 計算が終わった瞬間、メモリから削除されるだけ
- 演算子`*`もオブジェクト

クラス

クラス (class): オブジェクトを特徴づける属性

- `class()` 関数で確認可能
- すべてのオブジェクトは何らかのクラスを持つ

```
class(my_vec1) # my_vec1 のクラス
```

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

```
class(2)       # 2 のクラス
```

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

```
class(`*`)     # * のクラス
```

```
## [1] "function"
```

```
class(class)   # class() のクラス
```

```
## [1] "function"
```

なぜクラスが重要か

- ある関数の引数（後述）には使用可能なクラスが指定されている。
- `?関数名` で確認可能
- `mean()` 関数の例
  - `mean()` に使用可能な引数は `x`、`trim`、`na.rm`
  - `x`: numeric型ベクトル、logical型ベクトルなどが使用可能
  - `na.rm`: 長さ1のlogical型ベクトル（a logical value）のみ使用可能

```
mean(c(1, 2, 3, NA, 5), na.rm = TRUE)
```

- 関数を使いこなすためには関数のヘルプを確認する
  - 必要な引数、返されるデータの構造、サンプルコードなど豊富な情報が載っている。

関数と引数

関数 (function): 入力されたデータを内部で決められた手順に従って処理し、その結果を返すもの

- Rで起こるあらゆることは関数の呼び出しである (Chambers, 2016)
  - Everything that happens in R is a function call。
- 使い方: `関数名(関数の入力となるオブジェクト)`
  - 例) `class(my_vec1)`、`sum(my_vec1)`
- 自分で関数を作成することも可能
- 関数には**引数（ひきすう）**が必要

引数

`sum()`関数の例

```
sum(c(1, 2, 3, NA, 5), na.rm = TRUE)
```

- 関数名は`sum`
- **仮引数 (parameter)**: `na.rm`
- **実引数 (argument)**: `c(1, 2, 3, NA, 5)`、`TRUE`
  - `c(1, 2, 3, NA, 5)`の仮引数名はない（ヘルプを見ると`...`と表示される）。このように仮引数名が存在しないケースもある。
- 第一引数の仮引数は省略するケースが多い
- 第二引数以降は仮引数名を明記（実引数がどの仮引数に対応するかを明示）
  - 関数によっては引数は数十個ある場合も
- 引数には既定値 (default value)が指定されているものもあり、省略可能。既定値がない引数はすべて指定する。
  - たとえば、`mean()`の`trim`引数の既定値は`0`（`?mean`参照）
  - Usageに`仮引数 = 実引数`と表記されている場合、既定値が存在する
  - Arguemntsにan optional...と書いていれば、既定値はないが、指定しなくても関数は動くことを意味する。

反復

反復の方法

- ベクトル化による反復
  - ベクトルの全要素に対して同じ動作を繰り返す
- `for()` 文による反復
  - 処理を繰り返し回数を指定し、その回数に達したら終了する
- `while()` 文による反復
  - 一定の条件を指定し、その条件が満たされるときに処理を終了する

反復処理の例 (ベクトル化)

`my_vec1`の各要素に10を足す

```
my_vec1 + 10
```

```
## [1] 11 12 13 14 15
```

反復処理の例 (for() 使用)

`my_vec1` の各要素に10を足す

```
for (i in 1:length(my_vec1)) {
  print(my_vec1[i] + 10)
}
```

```
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
```

反復処理の例 (while() 使用)

`my_vec1` の各要素に10を足す

```
i <- 1
while (i < 6) {
  print(my_vec1[i] + 10)
  i <- i + 1
}
```

```
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
```

forループの使い方

```
for（任意の変数名 in ベクトル）{
  処理内容
}
```

- 任意の変数名は何でも良いが、一般的に`i`を使用（indexの略）
- ベクトル長さ1以上のベクトル
- `i`にベクトルの要素が逐次的に代入され、`{}`内部の処理を行い、全要素の代入が終わったら処理が終了する。

forループの仕組み

`for()`構文の仕組み

```
for (i in 1:3) { # 1:3 は c(1, 2, 3) に書き換えてもOK
  print(i)
}
```

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

1. `i`にベクトル`1:3`の最初の要素を代入(`i <- 1`)
2. `print(i)`を実行
3. `{}`中身の処理が終わったら`i`にベクトル`1:3`内の次の要素を代入(`i <- 2`)
4. `print(i)`を実行
5. `{}`中身の処理が終わったら`i`にベクトル`1:3`内の次の要素を代入(`i <- 3`)
6. `print(i)`を実行
7. `{}`中身の処理が終わったら`i`にベクトル`1:3`内の次の要素を代入するが、`3`が最後の要素なので反復終了

forループ（文字列）

文字列ベクトルも使用可能

```
University <- c("Kansai", "Kwansei-gakuin", "Doshisha", "Ritsumeikan")
for (Univ in University) {
  print(Univ)
}
```

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

for文内の処理

`{}`内部で様々な処理が可能

```
# length(University)は4のため、1:4と同じ
# 1:length(University) は seq_along(University)に書き換えてもOK
# seq_along()はベクトルを自動的にインデックスを作成する関数
for (i in 1:length(University)) {
  x <- paste0(i, "番目の大学名: ", University[i])
  print(x)
}
```

```
## [1] "1番目の大学名: Kansai"
## [1] "2番目の大学名: Kwansei-gakuin"
## [1] "3番目の大学名: Doshisha"
## [1] "4番目の大学名: Ritsumeikan"
```

参考) `paste0()`関数: 引数の文字列、数字を空白なしで結合する関数

```
paste0("1番目の大学名: ", University[1])
```

```
## [1] "1番目の大学名: Kansai"
```

forループ（リスト構造の利用）

`my_list`の各要素の平均値を求める

```
my_list1 <- list(Math      = c(92, 76, 90, 74, 65),
                 English   = c(92, 75, 76, 79, 88),
                 Geography = c(90, 67, 72, 83, 75))
```

```
for (i in 1:length(my_list1)) {
  print(mean(my_list1[[i]]))
}
```

```
## [1] 79.4
## [1] 82
## [1] 77.4
```

`{}`の中に`paste0()`などを使うと`"Mean of Math Score: 79.4"`のように出力させることも可能

多重forループ

`for()`文の中に`for()`文を挿入

- 代表的な例として掛け算九九
- 多重forループの場合、インデックスが重ならないようにすること
    - 2重for文の場合、通常、`i`と`j`が使われる

```
for (i in 1:9) {
  for (j in 1:9) {
    # paste0()でなくpaste()を使うと、結合の際、スペースが入る
    print(paste(i, "*", j, "=", i * j))
  }
}
```

```
## [1] "1 * 1 = 1"
## [1] "1 * 2 = 2"
## [1] "1 * 3 = 3"
## [1] "1 * 4 = 4"
## [1] "1 * 5 = 5"
## [1] "1 * 6 = 6"
## [1] "1 * 7 = 7"
## [1] "1 * 8 = 8"
## [1] "1 * 9 = 9"
```

whileループの使い方

目的: 「ある条件が満たされる限り、反復し続ける」あるいは「ある条件が満たされるまで、反復し続ける」

- **目標は決まっている**が、その目標が達成されるまでは処理を**何回繰り返せばいいか分からない**場合に使用

```
while（条件）{
  条件が満たされた場合の処理内容
}
```

`while()`の例

- 総和が30になるまでサイコロを投げ続ける
  - 6のみが出るなら5回反復で問題ないが、1のみ出続けると30回反復する必要がある。
  - 目標: これまで出た目の総和が30以上になる
  - 反復回数: 5回〜30回（決まっていない）

簡単なwhileループ（例）

1から3まで出力

- 以下のコードは`for()`文がより効率的であることに注意

```
i <- 1
while (i <= 3) {
  print(i)
  i <- i + 1
}
```

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

簡単なwhileループ（解説）

1. `i`に1を代入
2. `while()`構文スタート
3. `i <= 3`は`TRUE`であるため（`1 <= 3`は`TRUE`）、`{}`内のコードを実行
   - `i`を出力（`print(i)`）
   - `i`に1を足して`i`に上書きする（`i <- i + 1`）
   - 2に戻る（この時点で`i`は2）
4. `i <= 3`は`TRUE`であるため（`2 <= 3`は`TRUE`）、`{}`内のコードを実行
   - `i`を出力（`print(i)`）
   - `i`に1を足して`i`に上書きする（`i <- i + 1`）
   - 2に戻る（この時点で`i`は3）
5. `i <= 3`は`TRUE`であるため（`3 <= 3`は`TRUE`）、`{}`内のコードを実行
   - `i`を出力（`print(i)`）
   - `i`に1を足して`i`に上書きする（`i <- i + 1`）
   - 2に戻る（この時点で`i`は4）
6. `i <= 3`は`FALSE`であるため（`4 <= 3`は`FALSE`）、`{}`内のコードを実行せずに反復終了

高度な使い方

任意の正の実数`x`の平方根を求める方法（Hero of Alexandriaの近似法）

1. 平方根を求める`x`、任意の初期値`g`、非常に小さい正の実数`e`を設定する。`g`は1、`e`のは0.001とする。
   - `e`が小さいほど精度が高くなる
2. `g`の2乗を計算し、`x - g^2`の絶対値を`gap`とする（`gap`の初期値は`Inf`とする）。
3. `gap`が`e`より小さい場合、`g`を`x`の平方根として出力する。
4. `gap`が`e`より大きい場合、`g + (x / g)`を新しい`g`とする。
5. 2へ戻る。

以上の例は、`gap`が`e`より小さくなるまで繰り返すため、何回の繰り返しが必要かが不明

5の平方根を求めてみる

Hero of Alexandriaの近似法の実装

```
x   <- 5      # 平方根を求める正の実数
g   <- 1      # 任意の初期値
e   <- 0.001 # 許容する誤差の範囲
gap <- Inf    # gapの初期値

while (gap > e) {
  gap <- abs(x - g^2)
  if (gap > e) {
    g <- (g + x / g) / 2
  } else {
    print(g)
  }
}
```

```
## [1] 2.236069
```

```
sqrt(5) # Rの内蔵関数で5の平方根を計算
```

```
## [1] 2.236068
```

条件分岐

条件分岐の方法

1. `if()`、`else if()`、`else()`使用
   - 最も汎用性の高い条件分岐
   - 3分岐以上にも対応
   - `for()`や`while()`、または自作関数と組み合わせて使う場合も多い
2. `ifelse()`または、`if_else()`使用
   - ベクトルの各要素に対して条件分岐を行う
   - ただし、条件が`TRUE`の場合と`FALSE`の場合、2分岐のみに限られる
3. `switch()`使用
   - 長さ1の文字列ベクトルのみ使用可能な条件分岐
   - 解説しない

if条件分岐の使い方

```
if（条件1）{
  条件1が満たされる場合の処理内容
} else if（条件2）{
  条件2が満たされる場合の処理内容
} else if（条件3）{
  条件3が満たされる場合の処理内容
} else {
  上記のすべての条件が満たされない場合の処理内容
}
```

- `if()`は必須
- `else()`は任意。ただし、最後に1回のみ
- `else if()`は何回でも使用可能
- `()`内に複数の条件を指定することも可能（`&`、`|`演算子を使用）

if条件分岐の例（1）

`name`の値によって異なるメッセージを出力

```
name <- "Kandai"

if (name == "Kandai") { # nameが"Kandai"なら...
  print("関西一の大学")
} else if (name == "Kangaku") { # nameが"Kangaku"なら...
  print("関学とは違うのだよ、関学とは！")
} else { # nameが"Kandai"でも、"Kangaku"でもないなら
  print("その他の大学")
}
```

```
## [1] "関西一の大学"
```

if条件分岐の例（2）

`name`の値によって異なるメッセージを出力

```
name <- "Kangaku"

if (name == "Kandai") {
  print("関西一の大学")
} else if (name == "Kangaku") {
  print("関学とは違うのだよ、関学とは！")
} else {
  print("その他の大学")
}
```

```
## [1] "関学とは違うのだよ、関学とは！"
```

if条件分岐の例（3）

`name`の値によって異なるメッセージを出力

```
name <- "Doshisha"

if (name == "Kandai") {
  print("関西一の大学")
} else if (name == "Kangaku") {
  print("関学とは違うのだよ、関学とは!")
} else {
  print("その他の大学")
}
```

```
## [1] "その他の大学"
```

if条件分岐の例（4）

`name`の値によって異なるメッセージを出力

```
name <- "Rits"

if (name == "Kandai") {
  print("関西一の大学")
} else if (name == "Kangaku") {
  print("関学とは違うのだよ、関学とは!")
} else {
  print("その他の大学")
}
```

```
## [1] "その他の大学"
```

ifelse()の使い方

ベクトルの全要素に対して条件分岐を行う

- 返される結果は元のベクトルと同じ長さのベクトル
- 条件が`TRUE`と`FALSE`のみに対応している
  - `ifelse()`の中に`ifelse()`を入れることで3分岐以上を指定することも可能だが、非推奨
- 同じ機能の関数として`if_else()` ({tidyverse}の読み込みが必要)

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

例) `scores`の要素が60以上なら`"Pass"`、未満なら`"Fail"`を返す条件分岐

```
scores <- c(58, 100, 81, 97, 71, 61, 47, 60, 73, 85)
pf <- ifelse(scores >= 60, "Pass", "Fail")
pf # scores と同じ長さのベクトルが返される
```

```
##  [1] "Fail" "Pass" "Pass" "Pass" "Pass" "Pass" "Fail" "Pass" "Pass" "Pass"
```

関数の作成

なぜ関数を作る必要があるのか

Rで起こるあらゆることは関数の呼び出しである (Chambers, 2016)

- Everything that happens in R is a function call

同じコードを数回使うケースが多い

- 変数だけを変えながら同じ分析を行う
- `if()`を使った条件分岐の例（4回）

if() 例題コードの関数化

```
UnivEval <- function(name) {              # 引数をnameとするUnivEval関数
  if (name == "Kandai") {                 # nameが"Kandai"なら...
    x <- "関西一の大学"                   # xに"関西一の大学"を代入
  } else if (name == "Kangaku") {         # nameが"Kangaku"なら...
    x <- "関学とは違うのだよ、関学とは!"
  } else {                                # その他のケースなら...
    x <- "その他の大学"
  }
  return(x)                               # xを返す
}

UnivEval("Kandai")
```

```
## [1] "関西一の大学"
```

```
UnivEval("Kangaku")
```

```
## [1] "関学とは違うのだよ、関学とは!"
```

```
UnivEval("Waseda")
```

```
## [1] "その他の大学"
```

自作関数の作り方

```
自作関数名 <- function(引数1, 引数2, ...) {
    処理内容
    ...

    return(結果として返すオブジェクト名)
}
```

- 引数は0個以上（引数が必要ない関数も作成可能）
- 処理内容の最後には返す結果オブジェクト名を明示
- 引数は関数内オブジェクトとして使用可能
    - 既定値を指定する場合は、`引数名 = 既定値`

自作関数の例 (引数なし)

`"ラーメン大好き！"`というメッセージを出力する`Noodle()`関数の作成

- 単にメッセージを出力するだけなので、引数は不要

```
Noodle <- function() {
  # 関数内オブジェクトxにメッセージを格納する
  # 関数内オブジェクトは関数の処理が終わったらメモリから削除される
  x <- "ラーメン大好き！"
  return(x) # オブジェクトxの内容を結果として返す
}
```

テスト

```
Noodle()
```

```
## [1] "ラーメン大好き！"
```

自作関数の例 (引数1つ)

`x`面体サイコロ投げ (`sample()`関数使用)

- 第一引数 (`x`): 抽出対象のベクトル
- 第二引数 (`size`): 抽出個数
- `replace`: `TRUE`の場合、復元抽出。`FALSE`(既定値)の場合、非復元抽出
  - 非復元抽出: 一回抽出された要素は抽出されない（サイコロの場合、`TRUE`に指定）

```
# 第一引数の要素から無作為に1つの要素を抽出
sample(c(1, 2, 3, 4, 5, 6), size = 1, replace = TRUE ) # 6面体サイコロの例
```

```
## [1] 2
```

```
sample(1:20, size = 1, replace = TRUE) # 20面体サイコロの例
```

```
## [1] 12
```

```
sample(1:20, size = 5, replace = TRUE) # 20面体サイコロを5回投げる例
```

```
## [1] 14 10 18 17  5
```

自作関数の例 (引数1つ)

```
Dice <- function(side = 6) {
  result <- sample(1:side, 1, replace = TRUE)
  return(result)
}
```

テスト

```
Dice()   # 6面体サイコロ
```

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

```
Dice(side = 20) # 20面体サイコロ（1回目）
```

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

```
Dice(side = 20) # 20面体サイコロ（2回目）
```

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

自作関数の例 (引数2つ)

`side`面体サイコロを`trials`回投げる関数

```
# 既定値は6面体 (side = 6)の1回投げ (trials = 1)
Dice <- function(side = 6, trials = 1) {
  result <- sample(1:side, trials, replace = TRUE)
  return(result)
}
```

テスト

```
Dice(trials = 10)    # 6面体サイコロ10回投げ
```

```
##  [1] 6 3 4 2 5 5 3 2 3 3
```

```
Dice(20, 3) # 20面体サイコロ3回投げ
```

```
## [1] 19  2  2
```

この関数は?

この関数が解釈できれば、自作関数の入門レベルは超えている（かも）

```
mySum <- function(x, na.rm = FALSE) {
  if (na.rm == TRUE) {
    x <- x[!is.na(x)]
  }

  Result <- 0

  for (i in x) {
      Result <- Result + i
  }

  return(Result)
}
```

まとめ

プログラミングは簡単だから難しい

チューリング完全な言語の2条件

- **データの読み書き**
- **条件分岐**
- データの読み書きと条件分岐を組み合わせると**反復処理**が可能

プログラミングが簡単な理由

データの読み書き、条件分岐、反復処理を覚えるだけで、パソコンで出来るすべてのことができる。

プログラミングが難しい理由

データの読み書き、条件分岐、反復処理だけですべてを処理しなければならない。

本講義における位置づけ

- 中級以上を目指す場合、Rプログラミングは必須スキル
  - コードの長さが劇的に短くなる
- ただし、今後、`ifelse()` (= `if_else()`)以外は登場しない（と思う）

本日の内容

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)

- Rプログラミングの基礎: 教科書 第11章
- 関数の自作: 教科書 第12章
- 本講義の内容より広範囲、かつ詳細に解説しているため、必ず読んでおくこと