第6回 関数とフロー制御

2023年2月27日

目 次

1 ここで学ぶ事

- オリジナルの関数を作成して複数の処理をまとめて実行できるようにする．
- 条件分岐や繰り返し処理を学び，複雑な処理を実装できるようにする．

2 関数

いろいろな処理を実現するために，これまで関数を使ってきた．例えば，csv ファイルを読み込む時には，`read.csv()` 関数を使い，データフレームを作成するには `data.frame()` 関数を使った．ここでは自分で関数を作成する方法を学ぶ．

2.1 関数の作成

関数を作成するには function() を使う．以下のコードは画面に「Hello World!」と表示する hello() 関数を作成している．対話環境（プロンプト）で複数行のコードを入力するとコードが 1 つの式として完結するまで行頭に + が付く．ファイルにコードを入力する際は行頭の + は入力しない．

```
> hello <- function() {          # 関数を作成
+     cat("Hello World\n")
+ }
> hello()                        # 関数の実行
Hello World
```

上の例のように，関数に行わせたい処理は function() の後の中括弧 { } 内に記述する．ただし関数内の処理が 1 文から成る場合，中括弧は省略できる．

関数に引数を渡す場合は function() の括弧 ( ) 内に引数を保持するための変数を設置する．以下の例は name にユーザから与えられた引数が代入される．

```
> hello <- function(name) {
+     cat ("こんにちは，", name, "\n", sep="")
+ }
> hello("皆さん!")
こんにちは，皆さん!
```

複数の引数を取ることもできる．

```
> add <- function (x, y) {
+     x + y
+ }
> add(10, 5)
[1] 15
```

正確には，引数を受け取る変数（add() 関数の例では x と y）を仮引数（もしくはパラメータ，パラメータ変数）と呼び，関数呼び出し時に実際に渡される値（add(10, 5) の例では 10 と 5）を実引数（もしくは単に引数）と呼ぶ．用語が増えると初学者は混乱するので，ここではこれ以降どちらも単に「引数」と呼ぶことにする．

2.2 関数の戻り値

関数を呼び出すと，関数内の最後の式の値が呼び出し側に返る．これを関数の「戻り値」とか「返り値」と呼ぶ．

```
> calc <- function (x, y) {
+     x * y
+     x - y
+     x + y     # この式の結果が calc の戻り値となる．
```

```
+ }
> calc (3, 4)
[1] 7
```

明示的に関数内から戻りたい場合は以下のように return() を使う.

```
> calc <- function (x, y) {
+     x * y
+     x - y
+     return(x + y)     # 明示的に関数から抜けて値を返す
+ }
> calc (1, 2)
[1] 3
```

return() は関数の最後に置く必要はない．以下の例では 2 番目の文が実行されたら関数から抜けるので最後の文は実行されない.

```
> calc <- function (x, y) {
+     x * y
+     return(x - y)   # ここで関数から抜けて x-y の値を返す
+     x + y           # これは実行されない
+ }
> calc (1, 2)
[1] -1
```

2.3 デフォルト値の設定

関数定義で引数のデフォルト値を設定しておくと，関数を呼び出す側は引数を省略できる．下の例では hello() 関数の定義時に name に与える引数のデフォルト値として文字列「"皆さん"」を指定している．hello() 関数を引数なしで呼び出した場合には，name に"皆さん"が設定される.

```
> hello <- function(name="皆さん") {          # name のデフォルト値を設定
+     cat ("こんにちは, ", name, "\n", sep="")
+ }
> hello()            # name にはデフォルト値の「皆さん」が設定される
こんにちは, 皆さん
> hello("匠くん!")   # name には「匠くん!」が設定される
こんにちは, 匠くん!
```

引数が複数ある場合，デフォルト値を設定する引数は最後にまとめて並べる．デフォルト値が設定された引数は省略される可能性があるので，引数の順番が変わらないように必須引数は前に置く必要があるからだ．以下の例では name と birthday にはデフォルト値が設定されていないため，前に置かれれている.

```
> greeting <- function(name, birthday, greet="こんにちは", to="皆さん") {
+     str <- paste(to, ", ", greet, ". ",
+                  "私の名前は", name, "です. ",
+                  "誕生日は", birthday, "です. ", sep="")
+     str
+ }
> greeting("市東亘", "4 月 1 日")       # greet と to を省略しデフォルト値を使う.
[1] "皆さん, こんにちは. 私の名前は市東亘です. 誕生日は 4 月 1 日です. "
```

上の関数定義の最後の行は str だけになっている．これは str 変数の値を関数の戻り値にするためだ．R が変数を評価した結果はその値になるので，str の値が関数の戻り値になる．もしこの行を置かないと，str への代入式が関数の最後の式となる．代入文は評価されても値を返さないので，greeting() 関数は値を返さない．

デフォルト値を変更してみよう．

```
> greeting("市東亘", "4 月 1 日", "おはようございます")          # 最後の to はデフォルト値
[1] "皆さん, おはようございます. 私の名前は市東亘です. 誕生日は 4 月 1 日です. "
> greeting("市東亘", "4 月 1 日", "おはようございます", "先生")
[1] "先生, おはようございます. 私の名前は市東亘です. 誕生日は 4 月 1 日です. "
```

greet に渡す引数を省略し以下のように関数を呼び出すと,「先生」は 3 番目の greet に渡されてしまう.

```
> greeting("市東亘", "4 月 1 日", "先生")
[1] "皆さん, 先生. 私の名前は市東亘です. 誕生日は 4 月 1 日です. "
```

引数を幾つか省略するときは必ず下のように名前付き引数を使って関数を呼び出すと間違いがない.

```
> greeting("市東亘", "4 月 1 日", to="先生")
[1] "先生, こんにちは. 私の名前は市東亘です. 誕生日は 4 月 1 日です. "
```

全ての引数に名前を付けて呼び出すと，後でプログラムを読み返した時に分かり易いだけでなく，引数の順番も気にする必要がない.

```
> greeting(greet="こんばんは", name="市東亘", to="先生", birthday="4 月 1 日")
[1] "先生, こんばんは. 私の名前は市東亘です. 誕生日は 4 月 1 日です. "
```

2.4 関数の面白い使い方

関数は変数に代入することができる．例えば先に定義した greeting() 関数を変数 g に代入すると，g() への呼び出しは greeting() の呼び出しと同じことになる.

```
> g <- greeting
> g(name="亘", birthday="4 月")
[1] "皆さん，こんにちは．私の名前は亘です．誕生日は 4 月です．"
```

関数を引数として別の関数に渡すこともできる．下の add() 関数は引数に渡された adder() 関数を使って x と y を「add」する関数を定義している．

```
> add <- function(x, y, adder) {
+     adder(x, y)
+ }
```

例えば，数字を「add」する関数 add.num() と，文字を「add」する関数 add.str() 関数を以下のように定義する．

```
> add.num <- function(x, y) {
+     x + y                    # 数字の足し算
+ }
> add.str <- function(x, y) {
+     paste(x, y)              # 文字列の足し算は文字列をつなげる
+ }
```

それぞれの関数を add() 関数に渡すことによって，数字や文字の「足し算」ができる．

```
> add(1, 5, add.num)
[1] 6
> add("a", "b", add.str)
[1] "a b"
```

このように関数を引数に渡すことによって適切な処理を実現する方法はよく用いられる．ちなみに，引数に渡す段階で関数を定義することもできる．

```
> add("a", "b", function(x, y) {paste(x, y, sep="+")})
[1] "a+b"
```

3　フロー制御

複雑な処理を一つの関数にまとめようとすると，条件に応じて処理を変えたり，同じ処理を繰り返す必要が生じる．ここではプログラムのフロー制御の方法を幾つか学ぶ．

3.1 条件分岐

3.1.1 if 文

条件が TRUE か FALSE で処理を変えたい場合は if 文を使う．以下は if 文を使って絶対値を返す関数の定義である．

```
> absolute.value <- function(x) {
+     if (x > 0) {
+         x             # x > 0 が TRUE の時，x を返す.
+     } else {
+         -x            # x > 0 が FALSE の時，マイナス x を返す.
+     }
+ }
> absolute.value(5)
[1] 5
> absolute.value(-9)
[1] 9
```

以下の定義のように，条件が TRUE の時に return(x) を使って x の値を返して関数から抜けるようにすれば FALSE 時の else 文を省略することができる．

```
> absolute.value <- function(x) {
+     if (x > 0) {
+         return(x)   # x > 0 が TRUE の時，x を返す.
+     }
+     -x              # x > 0 が FALSE の時，マイナス x を返す.
+ }
> absolute.value(-100)
[1] 100
> absolute.value(4)
[1] 4
```

if-else 文をつなげると複数条件に対応する処理を作成できる．以下は，90 点以上で A，80 点以上で B，60 点以上で C，それ以外は D の成績付けをする grade() 関数の定義である．

```
> grade <- function(score) {
+     if (score >= 90) {
+         "A"
+     } else if (score >= 80) {
+         "B"
+     } else if (score >= 60) {
+         "C"
+     } else {
+         "D"
+     }
+ }
> grade(100)
[1] "A"
```

```
> grade(34)
[1] "D"
> grade(65)
[1] "C"
> grade(83)
[1] "B"
```

上の grade() 関数の定義では，else と if の間の { } 括弧が省略されている．else の後の if 文 1 つが条件分岐先なので，1 つの文の時には括弧が省略できるからだ．分岐先を明示的に括弧でくくると以下のようになる．

```
> grade <- function(score) {
+     if (score >= 90) {
+         "A"
+     } else {
+         if (score >= 80) {
+             "B"
+         } else {
+             if (score >= 60) {
+                 "C"
+             } else {
+                 "D"
+             }
+         }
+     }
+ }
```

上の定義でもプログラムの内容は同じだが，階層が深くなり読みづらい．通常，複数の分岐をつなげる場合は，後続の if 文を括弧でくくらない．ちなみに grade() 関数の if 節と else 節内の文も全て 1 文なので，以下の様に全ての括弧を省略して書くことができる．

```
> grade <- function(score) {
+     if (score >= 90)
+         "A"
+     else if (score >= 80)
+         "B"
+     else if (score >= 60)
+         "C"
+     else
+         "D"
+ }
```

3.1.2 ifelse() 関数

if-else 型の条件分岐は ifelse() 関数を使うと簡潔に書ける．さらに ifelse() を使うと多くの場合，プログラムの実行速度が改善される．ifelse() の使い方は，

ifelse(条件テスト，真の時，偽の時)

である．

```
> absolute.value <- function(x) {   # 絶対値を返す関数，ifelse 版.
+     ifelse(x > 0, x, -x)
+ }
> grade <- function(score) {        # 成績付け関数，ifelse 版.
+     ifelse(score >= 90, "A",
+            ifelse(score >= 80, "B",
+                   ifelse(score >= 60, "C", "D")))
+ }
```

ifelse() は入れ子にすると見づらいので，単純な分岐の場合によく用いられる．複雑な分岐の場合も実行速度が遅い場合は書き換えて改善するか試してみると良い．

3.1.3 switch 関数

if-else 文は 2 つの分岐が基本だが，switch() 関数は複数の分岐を同時に扱える．使い方は，

switch(式，選択肢 1，選択肢 2，選択肢 3，...)

で呼び出すと，第 1 引数の「式」が評価され，その値にマッチする選択肢が選ばれる．選択肢は幾つでも良い．「式」の評価結果は整数か文字列でなければならない．整数ならその数番目の選択肢が実行され，文字列なら名前がマッチする選択肢が実行される．

```
> switch(3, "選択肢 1", "選択肢 2", "選択肢 3", "選択肢 4") # 3 番目の選択肢が実行される
[1] "選択肢 3"
> switch("b", a="選択肢 1", b="選択肢 2", c="選択肢 3")    # 選択肢 b が実行される．
[1] "選択肢 2"
```

3.2 繰り返し処理（ループ）

R には他のプログラミング言語でも使われる for ループと while ループが用意されている．しかし R では第 3.4 節で紹介する apply 系と呼ばれる方法でループを実装する場合が多い．

3.2.1 for ループ

for ループは以下の形式をとる．

for(i in ベクトル) ループ処理本体

ベクトルの要素を順番に変数 i に代入しループ処理本体を実行する．変数名は i である必要はなく何でも良い．ループ処理本体に複数の処理を書く場合は { } 括弧でくくる．

```
> z <- 0           # 変数 z を 0 で初期化
> for(i in 1:10) {
+     z <- z + i   # 1 から 10 までの和を求める
+ }
> z                # for ループ内で z が更新された
[1] 55
> for(n in c("a", "b", "c", "d"))  # ループ本体を括弧で囲まないケース.
+     cat(toupper(n), "\n")        # toupper() は大文字に変換する関数
A
B
C
D
```

for ループを入れ子にすると 2 重ループになる.

```
> for(i in 1:5) {
+     for(j in 1:5)
+         cat(i, "-", j, ", ", sep="")
+     cat("\n")
+ }
1-1, 1-2, 1-3, 1-4, 1-5,
2-1, 2-2, 2-3, 2-4, 2-5,
3-1, 3-2, 3-3, 3-4, 3-5,
4-1, 4-2, 4-3, 4-4, 4-5,
5-1, 5-2, 5-3, 5-4, 5-5,
```

3.2.2 while ループ

while ループは条件が真の間，処理を繰り返す.

```
> countdown <- function(n) {              # n から 0 までカウントダウンする関数
+     while (n != 0) {
+         cat(n, ", ", sep="")
+         n <- ifelse (n >= 0, n-1, n+1)  # n が正ならカウントダウン，負ならカウントアップ
+     }
+     cat("0\n")
+ }
> countdown(10)
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
> countdown(-5)
-5, -4, -3, -2, -1, 0
```

3.3 練習問題

(1) 前回の実習の (7) で，最富裕国と日本と最貧国の 3 つの国の GDP を比較表示させた．そこで，ppp と年度を引数に取ると，最富裕国，日本，最貧国の国名と一人当たり GDP を一覧表示する compare.gdp() 関数を作成せよ．例えば，2010 年度のデータに対する実行結果は以下のようにならなければならない．

```
> compare.gdp(ppp, 2010)
Qatar 117518.7
Japan 34986.99
Congo, Dem. Rep. 646.2954
```

(2) x と y の最大公約数を求める関数 gcd(x, y) を作成せよ．

3.4 apply 系の繰り返し処理について

for ループや while ループは他のプログラミング言語でもよく使われる一般的な方法である．しかし，R ではこれらのループ処理よりもを使うよりも apply 系と呼ばれる手法を使ってループ処理を行うことが推奨されている．apply 系の繰り返し処理は，リストやデータフレーム[1]に対して繰り返し処理を行いたいときに使うので，適切な利用ケースが現れたときに説明することにする．

[1] リストについては第 2 回の「4 データ構造とは?」で少し説明した．また第 3 回の「2 データフレームとは?」で説明した通りデータフレームはリストである．