18 データ分析プログラミング（2）

花澤楓 学籍番号: 2125242

2023/11/04

1　関数（function）の活用

プログラミングにおける関数とは、コードをひとまとめにしたもの。プログラミングの目的に応じて、適切な関数を作成することでコードの冗長性を防ぐことができる。また、関数名を「コメント」として使うことでより可読性が高まる。関数を用いてコードを分割することで、どの箇所でエラーが出ているかなど、非常にわかりやすくなるようなメリットがある。また、関数内で作成された変数は関数内でのローカル変数となるため、関数外のコードに影響を与えない。そのため、意図しない挙動をすることを防ぐことができる。

プログラミング言語には様々な人が作成した関数を使用することもできるが、自作で作成することももちろん可能である。R では、コブダグラス型関数の値を返す関数を以下のように作成する。

```
calculate_cobb_douglas <- function(K, L){
  alpha <- 1/3
  A <- 1000
  Y <- A*(K^alpha)*(L^(1-alpha))
  return (Y)
}

Y <- calculate_cobb_douglas(K=100, L=200)
Y
```

```
## [1] 158740.1
```

関数は、引数（argument）、戻り値（outocome）で構成される。

R 言語=object-based functional-style program で、R 上に存在するものはオブジェクトと認識される。関数もオブジェクトと認識されるため、関数を引数として受け取るような関数である「高階関数」といった便利なものも使用できる。例えば、あるリストにあるベクトルの値を 2 倍にするコードを考える。for 文を使用する場合は以下のようになる。

```r
# リストを作成
my_list <- list(a = c(1, 2, 3), b = c(4, 5, 6), c = c(7, 8, 9))

# リスト内の各ベクトルを 2 倍にする
for (i in 1:length(my_list)) {
  my_list[[i]] <- my_list[[i]] * 2
}
my_list
```

```
## $a
## [1] 2 4 6
##
## $b
## [1]  8 10 12
##
## $c
## [1] 14 16 18
```

高階関数である purrr::map を使用すると、以下の通り。

```r
my_list <- list(a = c(1, 2, 3), b = c(4, 5, 6), c = c(7, 8, 9))

my_list <- purrr::map(my_list, ~ .x * 2)
my_list
```

```
## $a
## [1] 2 4 6
##
## $b
## [1]  8 10 12
##
## $c
## [1] 14 16 18
```

2　データの整理・変換のための tidyverse パッケージ群の関数

dplyr には、あるデータセットに対する操作ができるコマンド（関数）が複数ある。例えば、以下の通り。

1. 列ごと：

- mutate
- select

2. 行ごと：filter, arrange
3. グループごと：group_by, summarize
4. データフレーム（df1, df2 とする）を結合させる：
   - left_join: by=... を key としてデータフレームを結合
   - bind_rows: データフレーム df1, df2 を単純に結合させる

これらのコマンドの使用例を以下に示す。

```
test_score <- read_csv(here("Peanuts-Data-Project","02_raw",
                            "test_score","data",
                            "James Street Elementary School Tests.csv"))
```

```
# mutate() と select() を使って新しい列を生成・抽出
test_score_1 <- test_score |> mutate(
  average_math_sep = mean(Math_Sept)) |>
    select(Student_ID, average_math_sep)
# wide 型のデータフレームに対して実行したのであまり意味がない
test_score_1
```

```
## # A tibble: 5 x 2
##   Student_ID average_math_sep
##        <dbl>            <dbl>
## 1        105             75.6
## 2        103             75.6
## 3        104             75.6
## 4        101             75.6
## 5        102             75.6
```

```
# fileter() と arrange() を使って行の抽出・並べ替えを行う
test_score_2 <- test_score |> filter(
  Math_Sept >= 80
)
test_score_2 # september の math score が 80 点以上なのは 1 人だった
```

```
## # A tibble: 1 x 17
##   Student_ID English_Sept Math_Sept English_Oct Math_Oct English_Nov Math_Nov
##        <dbl>        <dbl>     <dbl>       <dbl>    <dbl> <chr>          <dbl>
## 1        103           72        84          78       83 75                76
```

```
## # i 10 more variables: English_Dec <dbl>, Math_Dec <dbl>, English_Feb <dbl>,
## #   Math_Feb <dbl>, English_Mar <dbl>, Math_Mar <dbl>, English_Apr <dbl>,
## #   Math_Apr <dbl>, English_May <dbl>, Math_May <dbl>
```

また、パイプ演算子 %>%（or |>）を用いることで、コードの可読性をあげられる例を以下に示す。

1. 中間データを記述する場合

```
gdp_japan <- dplyr::filter(gdp_world_annual, country == "JPN")
gdp_japan2 <- dplyr::group_by(gdp_japan, decade)
gdp_japan_decade <- dplyr::summarize(gdp_japan2,
                                     avg_gdp = mean(gdp))
gdp_japan_decade2 <- dplyr::ungroup(gdp_japan_decade)
```

2. パイプ演算子を用いる場合

```
gdp_japan_decade <- gdp_world_annual %>%
                dplyr::filter(country == "JPN") %>%
                dplyr::group_by(decade) %>%
                dplyr::summarize(avg_gdp = mean(gdp)) %>%
                dplyr::ungroup()
```