Requirements for Atorus Research

Grant Weaver

2023-03-07

S3 Data Types

Fluent with basic S3 data types in R, such as lists, dataframes, and vectors (numeric, character, and logical).

Lists

Lists are a collection of objects of different data types.

```
list_data <- list("MLK", 31.7, FALSE)
```

In this example, "list_data" is a list containing three objects: a character string ("MLK"), a numeric value (31.7), and a logical value (FALSE).

Dataframes

Dataframes are a two-dimensional arrays or tables that can hold data of different data types.

```
demographics_df <- data.frame(names = c("Jamal", "Tyrone", "Malik"),
                              ages = c(29, 51, 73), married = c(TRUE, FALSE, TRUE))
```

In this example, "demographics_df" is a dataframe containing three columns: "names", "ages", and "married". Each column has a different data type: "names" is a character vector, "ages" is a numeric vector, and "married" is a logical vector.

Vectors

Vectors are one-dimensional arrays that can hold data of the same type.

```
my_numeric_vector <- c(1, 2, 3, 4, 5)
my_character_vector <- c("Ford", "Tesla", "Honda")
my_logical_vector <- c(TRUE, FALSE, TRUE, TRUE)
```

In each of these examples, the vector is creating used the "c()" function and contains elements of the same data type.

Data Wrangling

Fluent with data wrangling (such as filtering, grouping, summarizing) with dplyr and tidyr.

dplyr

"dplyr is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges."

Filtering

filter() allows you to select a subset of rows in a dataframe.

```r
s <- ("/Users/grantweaver/Desktop/Stats/R/Tennis/Data/atp_matches_2022.csv")

atp_singles_2022 <- read_csv(s)

gs_final <- atp_singles_2022 %>%
  filter(tourney_level == "G", round == "F") %>%
  select(tourney_name, winner_name, score, loser_name)

knitr::kable(gs_final, col.names = gsub("[_]", " ", names(gs_final)))
```

tourney name	winner name	score	loser name
Australian Open	Rafael Nadal	2-6 6-7(5) 6-4 6-4 7-5	Daniil Medvedev
Roland Garros	Rafael Nadal	6-3 6-3 6-0	Casper Ruud
Wimbledon	Novak Djokovic	4-6 6-3 6-4 7-6(3)	Nick Kyrgios
Us Open	Carlos Alcaraz	6-4 2-6 7-6(1) 6-3	Casper Ruud

In this example, I first read in the tennis data for 2022 season. Then, I used filter() to filter results for Grand Slams and the Final. Then I used select() to select a few columns and then made the table using knitr so it's easier to read, while also changing the names from "tourney_name" to "tourney name" for readability.

Grouping & Summarise

group_by() groups the data into groups and summarise() collapses a group into a single row.

```r
top_match_wins <- atp_singles_2022 %>%
  group_by(winner_name) %>%
  summarize(n = n()) %>%
  arrange(desc(n)) %>%
  head(5)

knitr::kable(top_match_wins, caption = "Most Match Wins in 2022",
             col.names = gsub("[_]", " ", names(top_match_wins)))
```

Table 2: Most Match Wins in 2022

winner name	n
Stefanos Tsitsipas	61
Felix Auger Aliassime	60
Carlos Alcaraz	57
Andrey Rublev	51
Casper Ruud	51

This returns the top 5 players by match wins in 2022. In this example, I group the data by "winner_name", then use the summarize() to see return the number of match wins by player. Then arrange the data by descending order and use the head() to select the top 5.

tidyr

There are three interrelated rules which make a dataset tidy:

- Each variable must have its own column.
- Each observation must have its own row.
- Each value must have its own cell.

```r
pollute <- data.frame(city = c("New York", "New York", "London", "London",
                               "Boston", "Boston"), size = c(
                                 "large", "small", "large", "small", "large",
                                 "small"), amount = c(23, 14, 22, 16, 121, 56))


knitr::kable(pollute)
```

city	size	amount
New York	large	23
New York	small	14
London	large	22
London	small	16
Boston	large	121
Boston	small	56

In this example, I first created an untidy dataframe. It is untidy because there are amount of large particles and amount of small particles within the same variable. In the following section I will tidyr the dataframe.

spread

The spread() generates multiple columns from two columns:

- Each unique value in the *key* column becomes a column name.
- Each value in the *value* column becomes a cell in the new columns.

```r
pollute_tidy <- spread(pollute, size, amount)

knitr::kable(pollute_tidy)
```

city	large	small
Boston	121	56
London	22	16
New York	23	14

Taking the "pollute" dataframe, I used spread() to make the data tidy. Large and small now have their own separate columns.

gather

Collapses multiple columns into two columns:

- A *key* column that contains the former column names.
- A *value* column that contains the former column cells.

```r
basketball <- data.frame(player=c('A', 'B', 'C', 'D'),
                 year1=c(12, 15, 19, 19),
                 year2=c(22, 29, 18, 12))

knitr::kable(basketball)
```

player	year1	year2
A	12	22
B	15	29
C	19	18
D	19	12

Here I create a dataframe that I will later apply the gather() on.

```r
knitr::kable(gather(basketball, key="year", value="points", 2:3))
```

player	year	points
A	year1	12
B	year1	15
C	year1	19
D	year1	19
A	year2	22

player	year	points
B	year2	29
C	year2	18
D	year2	12

I used gather() to create two columns called "year" and "points". Now, there is only one column for year

Functions

What is a function? A function is a set of statements organized together to perform a specific task. Functions allow you to automate common tasks in a more powerful and general way than copy-and-pasting.

Examples

Example 1

```r
# Prime Function
is_prime <- function(n) {
  if (n <= 1) {
    return(FALSE)
  }
  for (i in 2:(n-1)) {
    if (n %% i == 0) {
      return(FALSE)
    }
  }
  return(TRUE)
}

is_prime(11)
```

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

```r
is_prime(24)
```

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

The "is_prime" function takes a positive integer as input and returns "TRUE" if the number is prime, and "FALSE otherwise". The results show that "is_prime(11)" returned "TRUE" because 11 is a prime number and "is_prime(24)" returned "FALSE" because the number is divisible by more than just 1 and itself.

Example 2

```r
# Confidence Interval
mean_ci <- function(x, conf.level = 0.95) {
  n <- length(x)
  m <- mean(x)
  s <- sd(x)
  se <- s / sqrt(n)
  alpha <- 1 - conf.level
  z <- qnorm(1 - alpha / 2)
  ci <- z * se
  lower <- m - ci
  upper <- m + ci
  list(mean = m, lower = lower, upper = upper, conf.level = conf.level)
}


set.seed(123)
x <- sample(0:100, size = 10, replace = TRUE)

mean_ci(x)
```

```
## $mean
## [1] 48.2
##
## $lower
## [1] 31.17399
##
## $upper
## [1] 65.22601
##
## $conf.level
## [1] 0.95
```

"mean_ci" takes a numeric vector "x" and an optional confidence level as the input, and then returns a list containing the median and confidence interval for the data. It also uses a "set.seed()" to make sure the results are reproducible.

Converting existing non-functional code into functions.

Example 1

```r
# Non-functional code
x <- 5
y <- 10
z <- x + y
print(z)
```

```
## [1] 15
```

```r
# Functional code
sum_xy <- function(x, y) {
  z <- (x + y)
  print(z)
}
sum_xy(5,10)
```

```
## [1] 15
```

This is a simple function that adds together two numbers and returns the sum.

Example 2

```r
# Non-functional code
x <- c(1, 2, 3, 4, 5, 6, 101, 2018)
y <- c()

for(i in 1:length(x)) {
  if(x[i] %% 2 == 0) {
    y <- c(y, x[i])
  }
}

print(y)
```

```
## [1]    2    4    6 2018
```

```r
# Functional code
get_evens <- function(x) {
  y <- c()
  for(i in 1:length(x)) {
    if(x[i] %% 2 == 0) {
      y <- c(y, x[i])
    }
  }
  print(y)
}

get_evens(c(1, 2, 3, 4, 5, 6, 101, 2018))
```

```
## [1]    2    4    6 2018
```

"get_evens" takes a numeric vector and then returns which values are even.

Example 3

```r
# Non-functional code
n <- 5
fact <- 1
for (i in 1:n) {
  fact <- fact * i
}
print(fact)
```

```
## [1] 120
```

```r
# Factorial Function
factorial <- function(n) {
  if (!is.numeric(n) || !as.integer(n) == n || n < 0) {
    stop("n must be a non-negative integer")
  }
  fact <- 1
  for (i in 1:n) {
    fact <- fact * i
  }
  return(fact)
}

print(factorial(5))
```

```
## [1] 120
```

```r
print(factorial(5.5))
```

```
## Error in factorial(5.5): n must be a non-negative integer
```

This function returns the factorial for any integer. “factorial” first checks if “n” is a numeric value using the “is.numeric()” function. Then it checks if “n” is an integer value by converting it to an integer using the “as.integer()” function and checks if the result is equal to “n”. This ensures that the input is both numeric and an integer value.