Functional and Object-Oriented Programming

2023-10-17

Imperative vs. Declarative Programming

Programming paradigms refer to the philosophy or approach being taken to structure and write one's code. There are two overall paradigms: declarative programming and imperative programming. In the declarative approach, the code is written to say what the programmer wants to achieve, whereas in the imperative approach, the code is written to say how to obtain what the programmer wants. The declarative paradigm focuses on defining a sequence of operations that determine the desired output, leaving the implementation details to the language itself. There are no internal states, but rather everything is passed from one function to the next. Imperative programming, on the other hand, uses objects to encapsulate states and methods, and programs are constructed through interaction between objects. This style emphasizes the changes in state, rather than the application of functions. In general, imperative programming is considered easier to read.

Object-Oriented Programming

Object-oriented programming (OOP) follows the imperative programming paradigm. OOP provides the advantage of polymorphism, meaning that one symbol can represent various types, enhancing the reusability and extensibility of software. It separates software interface from implementation, allowing the programmer to focus on the interface and expected behavior without requiring detailed knowledge of the implementation details. Objects encapsulate both data and functionality; the former is specified by the fields and the latter by the methods. Often, OOP provides the property of inheritance, by which child classes inherit fields and methods from parent classes in a hierarchical structure. The structure of how different classes relate allows software systems to be more easily maintained and extended.

Object-Oriented Programming in R

R provides three different built-in models for OOP: S3, S4, and reference classes (there are other models that can be accessed through various CRAN packages). In general, S3 is a good choice for prioritizing simplicity, while reference classes is the best implemented model (S4 is intermediate in both simplicity and proper implementation, so it is typically a poorer choice).

S3 In S3, there is no formal class definition, which offers flexibility but sacrifices some implementation concerns. Programming in S3 relies heavily on conventions, rather than mandated integrity checks. To create an S3 object, it is advisable to write a constructor function that initializes the object elements and returns the object. Within this function, one first creates a list and then sets the class of that list using the `class()` function (the class is typically the name of the function). It is considered best practice to have a separate validator function to carry out argument validation, as well as a user-friendly helper function. The `str` function can be used to display the structure of the resulting object.

After creating an S3 object in this way, it is possible to create specialized versions of generic methods like `print` and `plot` that work specifically for your class. Then when these methods are run on an object of that class, R will execute the specific version, rather than the default. In order to implement a method for a new

S3 object, there needs to be an existing generic function by that name. In order to create a new existing generic function, one must first create that generic function and then implement it for the specific class in question. When implementing a generic function for a specific S3 object, one must use the same arguments as the existing generic function.

S4 S4 provides a more formal framework for OOP in R. In S4, everything must be defined explicitly, rather than by convention as in S3. This approach requires more effort from the programmer to correctly specify all elements, but it results in clearer implementation and enables built-in integrity checks to validate the correctness of the code. When creating an S4 object, the class must be declared formally using `setClass` and providing the class name and the fields for the class, known as `slots` in S4. After declaring the class and defining its fields, we then define the methods, starting with an initialization method called `initialize` that takes in an object and additional inputs. This method assigns all the slots in the class, and if any slots need to be computed, it computes them. You can set additional methods for the class, such as `print` and `plot` in a similar way to `initialize`. Similarly to in S3, if you want to set a method for the class that does not yet exist, you must first call `setGeneric` to make the new generic function available and then `setMethod` to implement that method for the specific class. Unlike in S3, you do not need to write a constructor function to create a new object of a class. Instead, you can use the function `new`.

Next, it is necessary to add getter and setter methods to the class. Getter and setter methods are functions that allow access to an object's attributes. Getter methods retrieve the value of a specific attribute (read-only access), while setter methods modify the value of an attribute (write access). For both methods, it is necessary to first run `setGeneric` and then `setMethod`, as above. Note that the setter in R uses `<-`, rather than `=`. By writing these methods, you can also ensure that the resulting object is valid, maintaining the integrity of the object. To add extra validation, you can implement a validator function for the class using the `setValidity` method to ensure that your class follows certain desired properties.

S4 can be used to model relationships between objects. Instance-level relationships refer to the relationship between individual objects, which is modeled by adding a slot to their respective classes. Inheritance relationships allow all objects of one class to inherit from another class, i.e. the child class has all the slots of the parent class. In S4, a child class can inherit from multiple parent classes, known as multiple inheritance. Inheritance also determines which methods are associated with a given class.

Reference classes Reference classes are the most properly implemented form of OOP in base R. They differ from S3 and S4 by implementing methods for a class as part of the class definition directly, rather than declaring them as generic functions and then implementing them for the class. Reference Class also allows for modify-in-place, unlike S3/S4, meaning that data inside an object can be modified by the methods of the object without creating a new copy of the object.

Since reference classes are the most rigorous implementation of OOP in R, I have chosen to create a class for linear models using reference classes, in contrast to the S3/S4 examples in the lecture notes. First, I call `setRefClass` (instead of `setClass` as in S4). Note that the fields are now called `fields`, rather than `slots`. I can directly define all of the methods as part of the class definition.

```
simple_lin_regression <- setRefClass(
  "simple_lin_regression",
  fields = list(
    response = "numeric",
    regressor = "numeric",
    estimate = "numeric"
  ),

  methods = list(
    # Initialize method
  initialize = function(response, regressor) {
```

```
    .self$response <- response
    .self$regressor <- regressor

    # Define design matrix
    n <- length(response)
    D <- matrix(c(rep(1,n), regressor), ncol = 2)

    # Compute the OLS estimate
    b <- solve(t(D) %*% D) %*% t(D) %*% response
    .self$estimate <- as.numeric(b)
  },

  # Print method
  print = function() {
    cat("head(x) =", head(.self$regressor), "\n")
    cat("head(y) =", head(.self$response), "\n\n")
    cat("Estimated regression: E[y|x] = b0 + b1 * x\n")
    cat("b0 = ", .self$estimate[1], " and b1 = ", .self$estimate[2], ".\n", sep = "")
  },

  # Plot method
  plot = function() {
    plot.default(x = .self$regressor,
         y = .self$response,
         xlab = expression(X),
         ylab = expression(Y)
         )
    abline(a = .self$estimate[1], b = .self$estimate[2])
  },

  # Residual analysis method
  residual_analysis = function() {
    predictions <- .self$estimate[1] + .self$estimate[2] * .self$regressor
    residuals = .self$response - predictions
    plot.default(x = predictions,
         y = residuals,
         xlab = expression(hat(Y)),
         ylab = expression(Y - hat(Y)))
  }

  )
)
```

Note that I need to specify `plot.default` within the `plot` and `residual_analysis` methods in order to make those methods work properly. I can test this class as follows:

```
x <- rnorm(50)
y <- 2 - x + rnorm(50)
slr <- simple_lin_regression$new(response = y, regressor = x)
str(slr)
```

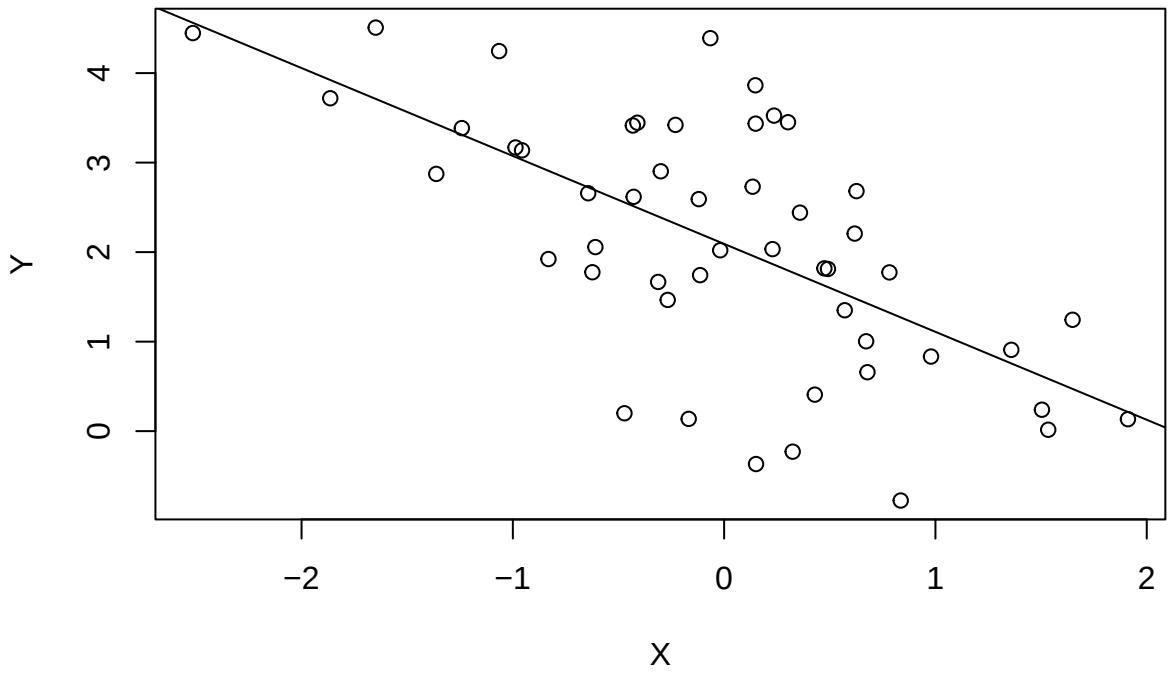
```
## Reference class 'simple_lin_regression' [package ".GlobalEnv"] with 3 fields
##  $ response : num [1:50] 2.73 1.92 1.74 4.45 3.44 ...
```

```
##  $ regressor: num [1:50] 0.135 -0.831 -0.114 -2.514 0.149 ...
##  $ estimate : num [1:2] 2.092 -0.982
##  and 18 methods, of which 4 are  possibly relevant:
##    initialize, plot, print, residual_analysis
```

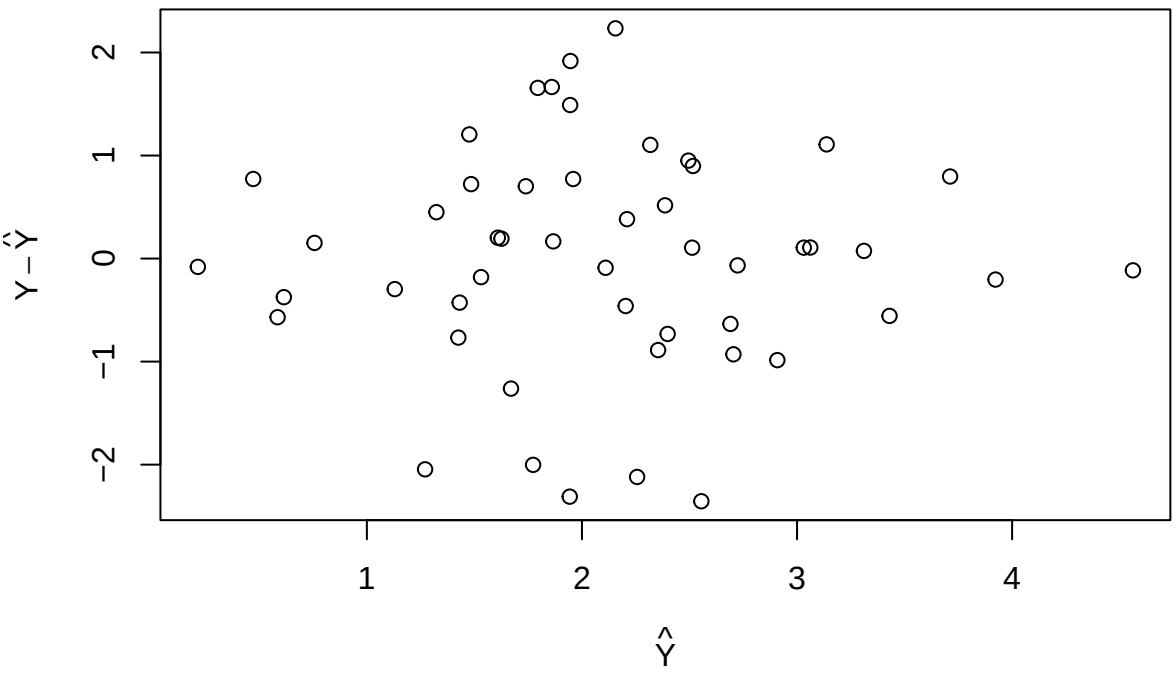
```
slr$print()
```

```
## head(x) = 0.1348914 -0.8314755 -0.1136578 -2.514442 0.1491165 -1.064771
## head(y) = 2.730902 1.922628 1.742733 4.447167 3.435274 4.245733
##
## Estimated regression: E[y|x] = b0 + b1 * x
## b0 = 2.091567 and b1 = -0.9823999.
```

```
slr$plot()
```




```
slr$residual_analysis()
```



Functional Programming

Functional programming, as opposed to OOP, follows the declarative paradigm. R supports many features of functional programming. For example, R supports first-class functions, meaning that functions can be arguments to other functions, can be returned by other functions, and can be stored in data structures, just like any other variable. Functional programming prioritizes the use of pure functions, i.e. functions that produce consistent outputs for the same inputs and avoid modifying the global program state. However, there are certain scenarios such as generating pseudo-random numbers or reading / writing data that necessitate impure functions.

Closures

A function is a closure when it is created by another function and has access to the variables defined within the environment of that function. For example, here is a closure for calculating the mean of a specific column:

```
make_column_mean_function <- function(data, column_name) {
  calculate_mean <- function(remove_na) {
    mean(data[[column_name]], na.rm = remove_na)
  }
  return(calculate_mean)
}

# Apply this closure to the penguin dataset
library(palmerpenguins)
```

```r
mean_flipper <- make_column_mean_function(penguins, "flipper_length_mm")
cat("Mean flipper length: ", mean_flipper(remove_na = TRUE))
```

```
## Mean flipper length:  200.9152
```

Lazy evaluation

R uses lazy evaluation for function arguments, meaning that arguments will only be evaluated when they are actually called, rather than as soon as they are encountered. Lazy evaluation improves performance and memory efficiency, but there can be some unexpected side-effects when using functions that return functions. To avoid these, you can use the `force` function within a function if you need to ensure that an argument is evaluated immediately.