Chapter_4

Henry Bourne

2022-10-24

Functional and Object-Oriented Programming

In this chapter we will go over how to write in both the functional and object oriented programming paradigms in R. First we will look at how to programme functionally.

Functional Programming

R supports many functional programming features, perhaps the most important is that is has **first-class functions**: a programming language has first-class functions if you can define a function, functions can be arguments to other functions, functions can be returned by functions and functions can be stored in data structures.

When functional programming it is good to avoid writing functions that modify the global programme state. A **pure function** is a function which given the same arguments returns the same output and has no side affects. Although it is unrealistic that all your functions will be pure, for example functions that generate pseudo-random numbers will not be pure.

In R we have that functions only have access to variables defined in their enviroment. If in the enviroment that a function is declared there are free variables (variables used in the function that haven't been asigned a value), then the enviroment binds the free variables and the function is a **closure**. So if there are free variables we get back a function and we can then pass to the function the arguments which we would like it to use for the free variables. For example:

```
f <- function(x){
  add_y <- function(y){
    x + y
  }
}
add_10 <- f(10)
add_10(5)
```

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

One thing to take especial note of is that R uses lazy evaluation. **Lazy evaluation** means that R doesn't evaluate an expression until it's value is needed. So, in the example above R doesn't evaluate the expression until we've given *add_10()* its argument and expect a value returned, ie. it doesn't evaluate the expression when we call f. This saves allot of computation, however, we must be careful as if we change variables along the way we must keep in mind that R will only evaluate when a value is needed (and therefore will use the most recently defined (current) values for variables).

Object-Oriented Programming (OOP)

Now we will look at how to do OOP in R. Note that in R this isn't very straightforward and there are actually multiple different ways in which we can programme in an object-oriented way in R. The key idea of OOP is **polymorphism** which essentially means that a single symbol may refer to different types. In

OOP we have that data and methods that operate on the data are bundled or **encapsulated** together in an object. This means that we can hide values or the state of a structured data object inside a class preventing direct access to them. The state of an object is defined by its **fields**, we can then perform operations on the state by defining **methods**. Many OOP implementations allow a class to **inherit** all the fields and methods from another class. We can use the **Unified Modelling Language (UML)** to graphically display the hierarchical structures of our classes (this is called a class diagram). We can also use OOP to create general purpose solutions to potential problems, allowing us to quickly perform a desired function without having to rewrite code, these general, reusable solutions are called design **patterns**.

In base R there are three ways of carrying out OOP. The first is S3 which is the least rigorous and can be thought of as functional OOP. Second is S4 which requires formal definitions. And lastly we have Reference Classes which implement properly encapsulated OOP, Reference Classes are also **mutable** (can be modified in-place). We start by describing S3...

S3

An **S3 object** is a base object with attribute class set to the class name. A **base object**