Unit 3: Functions

After reading this unit, students should

- understand the importance of function as a programming constructor and how it helps to reduce complexity and mitigate bugs.
- be aware of two different roles a programmer can play: the implementer and the client
- understand the concept of abstraction barrier as a wall between the client and the implementer, including in the context of a function.

Function as an Abstraction over Computation

Another important abstraction provided by a programming language is the *function* (or *procedure*). This abstraction allows programmers to group a set of instructions and give it a name. The named set of instructions may take one or more variables as input parameters, and return one or more values.

Like all other abstractions, defining functions allow us to think at a higher conceptual level. By composing functions at increasingly higher level of abstractions, we can build programs with increasing level of complexity.

Functions help us deal with complexity in a few ways:

- Functions allow programmers to compartmentalize computation and its effects. We can isolate the complexity to within its body: the intermediate variables exists only as local variables that has no effect outside of the function. A function only interacts with the rest of the code through its parameters and return value, and so, reduces the dependencies between variables to these well-defined interactions. Such compartmentalization reduces the complexity of code.
- Functions allow programmers to hide *how* a task is performed. The caller of the function only needs to worry about *what* the function does. By hiding the details of *how*, we gain two weapons against code complexity. First, we reduce the amount of information that we need to communicate among programmers. A fellow programmer only needs to read the documentation to understand what the parameters are for, what the return values are. There is no need for a fellow programmer to know about the intermediate variables or the internal computation used to implement the functions. Second, as the design and requirement evolve, the implementation of a

function may change. But, as long as the parameters and the return value of a function remains the same, the caller of the function does not have to update the code accordingly. Reducing the need to change as the software evolves reduces the chances of introducing bugs accordingly.

- Functions allows us to reduce repetition in our code through *code reuse*. If we have the same computation that we need to perform repeatedly on different *values*, we can construct these computations as functions, replacing the values with parameters, and pass in the values as arguments to the function. This approach reduces the amount of boiler-plate code and has two major benefits in reducing code complexity and bugs. First, it makes the code more succinct, and therefore easier to read and understand. Second, it reduces the number of places in our code that we need to modify as the software evolves, and therefore, decreases the chance of introducing new bugs.

Abstraction Barrier

We can imagine an *abstraction barrier* between the code that calls a function and the code that defines the function body. Above the barrier, the concern is about using the function to perform a task, while below the barrier, the concern is about *how* to perform the task.

While many of you are used to writing a program solo, in practice, you rarely write a program with contributions from only a single person. The abstraction barrier separates the role of the programmer into two: (i) an *implementer*, who provides the implementation of the function, and (ii) a *client*, which uses the function to perform the task. Part of the aim in CS2030/S is to switch your mindset into thinking in terms of these two roles.

The abstraction barrier thus enforces a *separation of concerns* between the two roles.

The concept of abstraction barrier applies not only to a function, but it can be applied to different levels of abstraction as well. We will see how it is used for a higher-level of abstraction in the next unit.