Chapter 1

# Subprograms

Writing programs to solve problems requires a solid understanding of the problem to be solved. In everyday problems, complexity may be a hurdle to software development. Simplifying approaches should be used to master and minimize this complexity. One such approach is "problem decomposition," which is the continuous decomposition of the problem into smaller ones for which *small software pieces* may be written. These pieces are then recomposed to form the whole solution.

When using a decomposition technique, many *software pieces* are often used, making it tedious to write them every time. This not only results in a longer development time, but it also makes maintenance more difficult because the code for these portions will be repeated across the software.

*Subprograms* are a good way to achieve such a decomposition. They are mini-versions of programs that can be invoked with input in order to solve small problems and return results to be used by other subprograms. In addition, subprograms offer a more powerful and elegant manner to solve problems by using recursion, but this course does not cover this feature.

## 1. Subprograms in the algorithmic language

Let's consider a program that displays a menu to executes some tasks in a bank information system. Assume there are three tasks: consult a customer account, debit an account, make a a transfer between two accounts and quitting the application. This can done with the following code:

algo

```
1  …
2  while true do
3      print(“C-Consult an account”)
4      print(“D-Debit an account”)
5      print(“T-Transfer from an account”)
6      print(“Q-Quit”)
7      answer=input(“Your choice”)
8      switch(answer)
9      begin
10         begin
11             case ‘C’:
12             begin
13                 // Code to consult an account
14                 end
15             case ‘T’:
16             begin
17                 // Code to transfer from an account
18                 end
19             …
20         end
21     end
```

At first sight, there is nothing wrong with this code. The program shows a menu prompting the user to select an action. Once this is done, the program invites the user to enter many inputs and performs various computations, all in the same area. As the number of menu items grows, the code becomes more messy. We will be obliged to build submenus to simplify the user interface.

It would be more interesting to separate the code for menu management from the rest of the program. This can be accomplished by assigning a name to these statements and using that name instead of all the statements. As a result, the statements will be written once and then utilized anywhere we want by just calling their names.

In the algorithmic language, we use the following syntax to do that:

algo

```
1  function menu():void
2  begin
3      print(“C-Consult an account”)
4      print(“D-Debit an account”)
5      print(“T-Transfer from an account”)
6      print(“Q-Quit”)
7  end
```

The first code becomes:

algo

```
1  …
2  while true do
3      menu()
4      answer=input(“Your choice”)
5      switch(answer)
6      begin
7          begin
8              case ‘C’:
9              begin
10                 // Code to consult an account
11                 end
12             case ‘T’:
13             begin
14                 // Code to transfer from an account
15                 end
16             …
17         end
18     end
```

A *function* is a mini-program that performs some processing. It is characterized by its name and can be called just by using this name with two parenthesis. A function may or may not compute a *return* value. In the latter, the keyword `void` is used. The code that calls `menu` is the *caller*, whereas the function itself is referred to as the *callee*.

A function is a mini-version of a *main* program in which variables can be declared. These variables are known as *local variables*. A variable declared in the main program is called a *global variable*. When a local variable and a global variable share the same name, the local variable prevail. This principle is widely used in computer science, and we refer to it as "locality takes over globality".

The previous function is a good start, but some menu-related code has yet to be integrated into the function. In fact, the function should prompt the user to input a choice and then indicate the user choice. To do this, we need:
- A local variable for storing the user's input results.
- The function *returns* its result (the output of the function). In this case, the function should specify the type of returned value and use statement `return`.

We get the following code:

algo
```
function menu():string
var s:string                              // this is the local variable
begin
    print("C-Consult an account")
    print("D-Debit an account")
    print("T-Transfer from an account")
    print("Q-Quit")
    s=input("Your choice")
    return s                              // this stops the function menu
end
…
answer=menu()                             // the main program
```

The call `menu()` executes the function `menu` and then returns the value that will be stored in the variable `answer`. Inside the function `menu`, the execution of `return` terminates the function and returns the value placed after `return`. In the algorithmic language, it is possible to return just one value at a time.

## 2. Arguments

The function `menu()` can communicate with the outer world through the return value. However, most of the time, the outer world would also want to communicate with the function by giving inputs. Let's modify the function to control if the program should display the menu item "Transfer from an account". This should be done by a boolean variable `transfer`:

algo
```
function menu(transfer:boolean):string
var s:string                              // this is the local variable
begin
    print("C-Consult an account")
    print("D-Debit an account")
    if transfer then
        print("T-Transfer from an account")
    print("Q-Quit")
    s=input("Your choice")
    return s                              // this stops the function menu
end
…
answer=menu(true)                         // the main program
```

The call of the function `menu` should indicate the value of the variable `transfer` (here `true`). This variable is called an *argument*, the function cannot be called without specifying the values of all arguments. It can be convenient, though inaccurate, to imagine an argument as a local variable whose value is initialized by the caller. A function can take any number of arguments of any type. However, the caller should always provide the correct number of arguments with the appropriate types.

To understand the usefulness of arguments, let's write a factorial function that will be called many times:

algo
```
1  algorithm factorials
2
3  function fact(n:integer):integer
4  var i,res:integer
5  begin
6      res=1
7      for i=2 to n do
8          res=res*i
9      return res
10 end
11
12 var i,m:integer
13 begin
14     m=integer(input("m= "))
15     for i=1 to m do
16         print("Factorial of",i,"is",fact(i))
17 end
```

It should be noted that the function is unaffected by the origin of its argument; it simply computes its output for a given n. This will certainly reduce the complexity of the code, but it is not optimal since the loop is executed every time. Let's remember that $n! = n * (n-1)!$, so why recompute the factorials of previous numbers each time? To solve this problem, we need the value of the last execution of `fact` to pass it to the next execution of the function. We might write the following code:

algo
```
1  algorithm factorials
2
3  function fact(n,prec:integer):integer
4  begin
5      prec=prec*n
6      return prec
7  end
8
9  var i,m,v:integer
10 begin
11     v=1 m=integer(input("Up to which value? "))
12     for i=1 to m do
13         print("Factorial of",i,"is",fact(i,v))
14 end
```

Unfortunately, this code does not work since the value of v will remain 1. This is because the function `fact` will just get a *copy* of v and not the variable itself. We say that the variable is passed *by value*. If a variable is passed by value, its current value does not change since the callee will work on a copy of the variable, and not the variable itself. Note that, in this case, the call can also be with an expression instead of a variable.

Another way to pass an argument is *by reference*. A variable passed by reference can be changed by the callee, while the caller will see any change made to it. Let's rewrite the previous code as:

algo
```
1  algorithm factorials
2
3  function fact(n:integer, var prec:integer):integer
4  begin
5      prec=prec*n
6      return prec
7  end
8
9  var i,m,v:integer
10 begin
11     v=1 m=integer(input("Up to which value? "))
12     for i=1 to m do
13         print("Factorial of",i,"is",fact(i,v))
14 end
```

References are indicated by the use of the reserved word `var` in this type of argument. The code now works, and the variable `v` is properly updated. To decide whether an argument should be passed by the rule is simple: if the argument won't be changed by the callee, pass it by value; otherwise, pass it by reference. However, there is an exception:

An array should always be passed by reference.

In fact, any data structure that may require a large amount of memory should be passed by reference. This is because copying may be a time-consuming and memory-intensive operation.

**Remark:** This problem can also be solved by global variables. Nevertheless, even if global variables cannot always be avoided in an imperative language, their usage should be limited as much as possible since they make programs more complex to understand and maintain.

## 3. Functions in Python

Python offers more potentialities to define and call functions. First, let's take a look on the syntax for defining a function in Python:

py
```
1 def f(arguments) -> type :
2     ...
```

`arguments` is a comma-separated list of variables (also known as parameters) and their types (keep in mind that typing is not mandatory in Python, although recommended). As an example, here is the function that shows the menu and returns the user's choice:

py
```
1 def menu(transfer:bool) -> str :
2     print("C-Consult an account")
3     print("D-Debit an account")
4     if transfer:print("T-Transfer from an account")
5     print("Q-Quit")
6     s:str=input("Your choice")
7     return s
```

This function will be called just by `menu(True)` . If the function is called without arguments or with more than one argument, then a runtime error will occur. Note that Python does not verify the type of the arguments, so a call `menu(1)` won't cause any error. In fact, this is a correct invocation of the function (why?).

In Python, calling functions can be a kind of fun. Assume we don't want to specify the value of the argument for the function `menu` each time, especially since the menu item "Transfer from an account" may be displayed in most cases. This can be done by:

```py
def menu(transfer:bool=True)
```

This syntax tells Python that the *default* value of the argument transfer is `True` . This means that if the user provides a value of this argument, the function will use it normally. However, if no argument is provided, the default value is used (in our example, `menu()` is equivalent to `menu(True)` ). This is known as a *default value*.

Global variables can also be used in Python. In fact, the keyword `global` allows defining a global variable that can be read from any function. However, in order to be updated, a global variable should also be declared in the function scope:

```py
y=...
...
def f():
    global y
    # now y can be updated and
    # update will be visible
```

## 3.1. Passing arguments by value or reference

Since Python has no `var` keyword, how can we pass arguments by reference? In fact, basic types cannot be passed by reference. Let's consider the following example:

```py
def f(a:int) -> None :
    a=4
...
a=5
print(a)
# f has no effect on the variable a in this scope
f(a)
print(a)
```

This program will print 5 twice, meaning that the update has been done inside `f` , and has been applied on a local variable. Therefore, integers, booleans, floats, and strings cannot be changed inside a function (any update will not be applied to the argument). In fact, all basic types are *immutable*.

In contrast, any other type (lists, dictionaries, classes) is always passed by reference. For example, modifying a cell in an array is visible outside of the function is visible from the outside:

```py
def f(a:list[int]) -> None:
    a[0]=1
...
arr=[0]*5
print(a) # [0,0,0,0,0]
f(a)
print(a) # [1,0,0,0,0]
```

## 3.2. Return values in Python

Since Python cannot pass integers by reference, we cannot directly implement the program that computes the sequence of factorials efficiently. We can solve the problem by using an array (though this is not a good solution in this case):

```py
def fact(n:int,prec:list[int]) -> int :
    prec[0]=n*prec[0]
    return prec[0]

v=[1]
m=int(input("m="))
for i in range(1,m+1):
    print(f"Factorial of {i} is {fact(i,v)}")
```

In this program, the function changes the value of the first cell of the array `list` . This update can be seen outside the function, therefore, the cells of the array `v` are updated after each call to the function `fact` .

There is another, more elegant solution to handle this situation (and many more). In fact, an argument with a basic type cannot be changed by the function (at least from the caller's perspective). The solution is that the function returns the new values of the arguments. Helpfully, Python functions can return any number of values as a tuple. Using unpacking, this will look as if the function returns many values. The previous example will hence be:

```py
def fact(n:int,prec:int) -> int :
    res=n*prec
    return res,res

v=1
m=int(input("m="))
for i in range(1,m+1):
    res,v=fact(i,v)
    print(f"Factorial of {i} is {res}")
```

## 3.3. Position and keyword arguments

When a Python function is called, its arguments can be named. Let's consider the function `fact` in the last program. The call `fact(i,v)` is equivalent to `fact(n=i,prec=v)` . In this call, we are formally expressing that the value of the argument named `n` (as declared in the function) is `i` and the value of the parameter `prec` is `v` . Now that the function knows the value of each of its parameters, it is possible to change the order of the arguments, such as `fact(prec=v,n=i)` . In the call `fact(i,v)` , both `i` and `v` are called *position-based* arguments. In contrast, in the call `fact(n=i,prec=v)` , each argument is called a *keyword-based* argument.

Combining position-based arguments along with default arguments leads to very flexible function definitions. In addition, position-based arguments can be used to call a function with arbitrary arguments (this won't be covered in this course).

However, Python forbids some combinations. For instance, when calling a function, once a keyword-argument is used, all following arguments should also be keyword-arguments.

## 3.4. Packing and unpacking arguments

When using position arguments, it is possible to use unpacking to call a function with values from an array. Let's consider a function `f(a:int,b:int,c:int)` . A normal call of `f` would be `f(1,2,3)` in which `a=1` , `b=2` and `c=3` . Thanks to unpacking, we can use `f` with values unpacked from an array. For instance, consider an array `arr=[1,2,3]` , the previous call to `f` is completely equivalent to `f(*arr)` .

Packing and unpacking can be further used to have a function with a variable number of arguments. As an example, let's consider a special implementation of the Python function `sum` that sums up all arguments that are passed:

```py
def my_sum(*a):
    s=0
    for v in a:
        s+=v
    return s
```

Here, we can see the argument `a` is a packed argument, i.e., that `my_sum` can be called with any number of arguments. Inside the function, the argument `a` is treated as an ordinary argument. With this syntax, we can call `my_sum(1,2)` to have the result 3, `my_sum(1,2,3)` to have the result 6, and `my_sum()` to have the result 0. Note that packing and unpacking in this case is generally incompatible with keyword arguments. However, it is possible to ordinary arguments as well as packed arguments in the same function. `print` is a typical example of a function using packing.

Consider a function `choose(...)` that receives at least one argument (call it `i` ) plus a variable number of arguments; then it should return the *i*-th argument for the list. For instance, `choose(2,0,2,4,5)` returns 4. If there is no *i*-th, an error is raised (the function can also use negative indices):

```py
def choose(i:int,*elts) -> int:
    if i<len(elts):
        return elts[i]
    else:
        raise IndexError("No enough arguments to choose from")
```

## 3.5. Internal functions

In this section, we will first rewrite the factorial program so that a function `compute_factorials` call the function `fact` :

```py
def compute_factorials(m):
    v=1
    for i in range(1,m+1):
        res,v=fact(i,v)
        print(f"Factorial of {i} is {res}")

def fact(n:int,prec:int) -> int :
    res=n*prec
    return res,res
...
compute_factorials(int(input("m=")))

```

This is a more interesting version of the program in which the main program just call functions. But, wait, there is more. Suppose that the function `fact` is just used inside `compute_factorials` , why may other functions invoke it? Python solves this issue by allowing internal functions. An internal function is declared inside another function and can only be inside it.

```py
def compute_factorials(m):
    def fact(n:int,prec:int) -> int :
        res=n*prec
        return res,res
    # function compute_factorial begin here
    v=1
    for i in range(1,m+1):
        res,v=fact(i,v)
        print(f"Factorial of {i} is {res}")
```

**Remark:** If a function `f` define an internal function `g` , then all arguments of `f` can be seen by `g` .