Modular programming

- Code development activity is less and less one individual task
- Code development projects are increasingly complex involving a large number of people working as a team
- Fast development is expected in this area that requires modify, develop and improve the code
- The increase capacity of computers has made the efficiency of programs (speed and resource management) no longer being the main objective of programming
- Today, the quality of programs depends more and more on the ease with which they are understood and the quickness with which they can be changed
- In the analysis of the problems, try to go from the general to the particular (technique known as "top - down") and whenever possible divide the problem into simpler sub-problems
- In the implementation of the programs use a modular organization starting from the particular to the general (technique known as "bottom - up") using small, independent and reusable modules

Subprograms / Functions

- Subprograms/functions are block of instructions grouped in units with the objective of performing specific tasks
- These tasks are performed within each subprograms in a way that is independent of the other units
- When Subprograms are called the instructions within the subprogram are executed
- When the subprogram ends values can be returned and the execution continues at the point from where the subprogram was called
- The communication between the units is done through sharing information associated with the variables listed as arguments of the subprograms

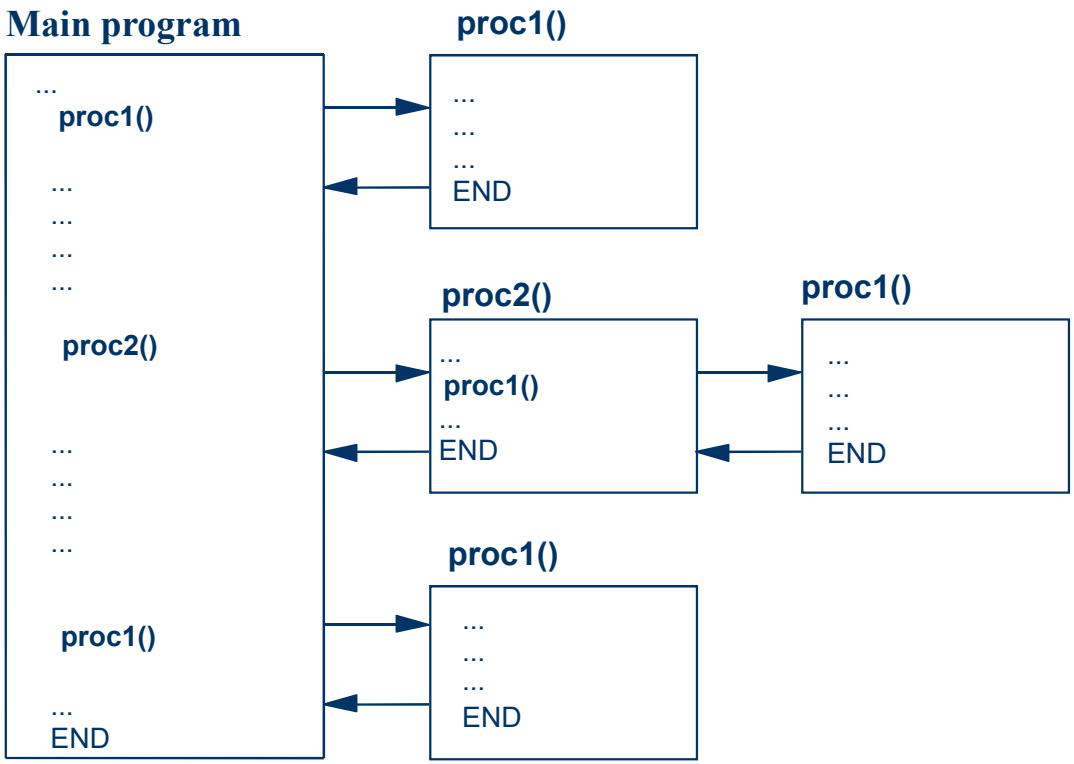
Subprograms / Functions

Advantages of using subprograms

- Blocks of repeated instructions can be grouped into subprograms thus avoiding unnecessary duplication of the same code and promoting its reuse
- A simpler approach to complex problems through the use of smaller units (subprograms) with less complexity
- Greater ease of change and maintenance of the code
- Subprograms can be grouped in Modules (code libraries) and be ready to use in new programs
- Modules are created in python files (.py) and to be used needs to be included using the statement: import mymodule

Subprograms / Functions

Subprogram calling scheme



Python Functions

- A Python function is defined by the name **def** followed by the function name and parenthesis. Inside parenthesis it can be defined a list of arguments separated by commas. Optionally, a function can return a value.

```
def my_function(arg1, arg2, …):
    …
    return value
```

Examples:

Function definition	**Function call**	**Output**
def greeting(): print("Good morning")	greeting()	Good morning

def greeting(name):
  greet_name = "Good morning " + name
  return greet_name
	my_name = "John"
	print(greeting(my_name))
		Good morning John

Function arguments

The communication with functions is done through the variables in the argument list.

There are the following types of arguments:

- Required arguments
- Keyword arguments
- Default arguments
- Variable number of arguments

Required arguments

- The correspondence between the variables that appear in the list of arguments of the function call and the parameters in the definition of the function is made by the position of these variables in the list and not by their name. That is, the first variable in the argument list of the function call corresponds to the first variable in the parameters list in the function definition, and so on.

```
def sum_values(e, f, g):
    h = 3 * e + 2 * f + g
    return h
```

```
a = 2
b = 3
c = 4
d = sum_values(a, b, c)
Print(d)
```

Function arguments

Keyword arguments

- The parameter names are used to identify the arguments in the list of the function call. The arguments can be written in any order in the function call list.

```
def sum_values(e, f, g):
    h = 3 * e + 2 * f + g
    return h
```

```
d = sum_values(f = 3, g = 4, e = 2)
Print(d)
```

Default arguments

- The parameters can have a default value using the assignment operator. If a parameter has a default value the corresponding argument in the function call is optional, assuming the default value if it is omitted.

```
def sum_values(e, f = 5):
    h = 3 * e + 2 * f
    return h
```

```
a = 2
d = sum_values(a)
print(d)
b = 3
d = sum_values(a, b)
print(d)
```

Function arguments

Variable number of arguments

- In the function definition an asterisk (*) can be used before the parameter name to denote an arbitrary number of arguments.

```
def sum_values(*values):
    h = 0
    for value in values:
        h = h + value
    return h
```

```
d = sum_values(2, 3)
Print(d)
d = sum_values(2, 3, 4)
Print(d)
```

Lambda function

A small inline function that can be used as an anonymous function inside another function

```
func = lambda a, b: a ** b
print(func(2,3))
```

```
list1 = [1,2,3,4,5,6]
square = map(lambda a: a ** 2, list1)
print(list(square))
```

The map() function executes a function for each item in an sequence

Variable scope

Local scope

- A variable defined within a function has local scope and can only be accessed from the point where it was defined until the end of the function. A local variable cannot be accessed outside the function where it was defined.

```
def myfunction():
    value = 5
    print(value)

myfunction()
print(value)        # Error
```

Variable scope

Global scope

- A variable defined outside any function has global scope and can be accessed from anywhere within the program. A global variable can be used by any function.

```
def myfunction():
    print(value)

value = 5
myfunction()
```

```
def myfunction():
    value = 3          #Local
    print(value)

value = 5
myfunction()
print(value)
```

```
def myfunction():
    global value
    value = 3          #Global
    print(value)

value = 5
myfunction()
print(value)
```