Tutorial III.I - Functions in Julia

Applied Optimization with Julia

Introduction

Imagine you have a helpful worker. Instead of explaining every little task to it each time, you can teach it specific jobs once, and then just ask it to do those jobs later. In programming, these “jobs” are called functions! This tutorial will show you how to create and use functions in Julia, making your code more organized and reusable.

Follow the instructions, input your code in the designated areas, and verify your implementations with `@assert` statements.

Section 1 - Creating and Calling Functions

Functions in Julia encapsulate reusable code and can be defined using the `function` keyword or shorthand syntax. After the keyword, you name the function and write the parameters in parentheses. Later, you can call the function by writing the name of the function followed by the parameters in parentheses.

Thus, think of a function like a recipe:

1. It has a name (like “say_something”)
2. It might need ingredients (our “parameters”)
3. It has steps to follow (the code inside the function)
4. It usually produces something (we call this the “return value”)

Let’s see some examples:

```
# A simple function to greet someone
function say_hello(name)
    return "Hello, $name!"
end

# Using our function
message = say_hello("Elio")
```

```
function multiply(a,b)
   a * b
end
multiply(10, 5)
```

```
50
```

The second function takes two parameters, multiplies them, and returns the result implicitly. In Julia, the last expression is automatically returned, making the `return` keyword optional. If you explicitly use the `return` keyword in the function, it will return the value immediately once the function encounters the keyword and stops the further execution of the function. That way, you can also use the keyword in conditional statements and use it to return a value based on a condition. For example:

```
function do_something(a,b)
   if a > b
      return a * b
   else
      return a + b
   end
end
println("The result of do_something(10, 5) is $(do_something(10, 5))")
println("The result of do_something(5, 10) is $(do_something(5, 10))")
```

```
The result of do_something(10, 5) is 50
```

```
The result of do_something(5, 10) is 15
```

> **Tip**
>
> Functions are like teaching a robot new skills:
> - The function name is like the skill name (e.g., “make_sandwich”)
> - Parameters are things the robot needs to do the job (e.g., bread, filling)
> - The code inside are the steps to follow
> - The return value is the finished product

Exercise 1.1 - Define and Test a Simple Addition Function

Define and test a simple addition function. Define a function `add` that takes two parameters and returns their sum.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert add(10, 5) == 15 "The sum computed is $(add(10, 5)) but should be 15."
println("The sum computed is $(add(10, 5)), wonderful!")
```

Section 2 - Scope within Functions

When you create variables inside a function, they're like secret ingredients - they only exist inside that function's "kitchen". We call this "local scope". Thus, variables declared inside a function are local to that function and are not accessible outside.

```
function bake_cake()
    secret_ingredient = "vanilla extract"
    println("Adding the secret ingredient: $secret_ingredient")
end
bake_cake()  # This works fine
```

```
Adding the secret ingredient: vanilla extract
```

```
# But this would cause an error:
# println(secret_ingredient)
```

If you want to access the variable outside of the function, you have to explicitly return it. You can do this by passing `return` in front of the variable you want to return from the function.

Exercise 2.1 - Return a Local Variable

Try to execute the following block of code. The objective is to understand how to return the `local_variable_one` from the function `scope_test`. Your task is to change the function, to return the value of `local_variable_one`.

```
# YOUR CHANGES BELOW
function scope_test()
    local_variable_one = 10
    local_variable_two = 20
end

# YOUR CHANGES ABOVE
# Test your function
@assert scope_test() == 10 "The value exported is $(scope_test())."
println("The value exported is $(scope_test()), you solved it!")
```

Exercise 2.2 - Define an Implicit Return Function

Define and test an implicit return function. Define a function `subtract` that takes two parameters and implicitly returns their difference. The implicit return feature makes your code cleaner and more concise.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert subtract(10, 5) == 5  "The difference computed is $(subtract(10, 5)) but should
 ↪  be 5."
println("The difference computed is $(subtract(10, 5)), perfect!")
```

Section 3 - Multiple Dispatch

Multiple dispatch in Julia allows defining function behavior based on argument types, promoting code reuse and clarity. It's a powerful feature for designing flexible and extensible functions. We first define a generic version and then provide specific implementations for different types:

```
# Generic operation for objects of all types.
function operation(a, b)
    "Generic operation for objects of type $(typeof(a)) and $(typeof(b))"
end

# The specific implementations are:
operation(a::Number, b::Number) = a + b        # Specific method for Number types.
operation(a::String, b::String) = string(a, b) # Specific method for String types.

# Test with different types of arguments.
result1 = operation(10, 20)
println(result1)
result2 = operation("Hello, ", "World!")
println(result2)
result3 = operation("Hello, ", 20)
println(result3)
```

```
30
Hello, World!
Generic operation for objects of type String and Int64
```

Exercise 3.1 - Match Results to Assertions

Match the results from the previous example to the correct assertions:

```
# YOUR CHANGES BELOW
@assert result2 == 30 "result1 should be the sum of two numbers"
@assert result3 == "Hello, World!" "result2 should be the concatenation of two strings"
@assert result1 == "Generic operation for objects of type String and Int64" "result3
 ↪  should use the generic operation"
println("You solved it, the assertions are now correct!")
```

> **Tip**
>
> **Hint**: Look at the types of arguments used in each `operation` call and match them to the appropriate method.

Conclusion

Great work! You've just completed the tutorial on functions in Julia. You now have a first understanding of how to create, use, and understand functions in Julia. Continue to the next file to learn more.

Solutions

You will likely find solutions to most exercises online. However, I strongly encourage you to work on these exercises independently without searching explicitly for the exact answers to the exercises. Understanding someone else's solution is very different from developing your own. Use the lecture notes and try to solve the exercises on your own. This approach will significantly enhance your learning and problem-solving skills.

Remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities. If you encounter difficulties, review the lecture materials, experiment with different approaches, and don't hesitate to ask for clarification during class discussions.

Later, you will find the solutions to these exercises online in the associated GitHub repository, but we will also quickly go over them in next week's tutorial. To access the solutions, click on the Github button on the lower right and search for the folder with today's lecture and tutorial. Alternatively, you can ask ChatGPT or Claude to explain them to you. But please remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities.