Tutorial III.I - Functions in Julia

Applied Optimization with Julia

Functions in Julia

This interactive Julia script is a comprehensive guide to understanding functions in Julia. Dive into the creation and usage of functions, grasp the concept of scope within functions, explore returning values, use anonymous functions, and leverage multiple dispatch for versatile function behavior. 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. For example:

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

This 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.

Exercise 1.1

Define and Test a Simple Addition Function. Define a function `add` that takes two parameters and returns their sum.

```
# Define and Test a Simple Addition Function. Define a function `add` that takes two
 ↪  parameters and returns their sum.
# YOUR CODE BELOW

# Test your function
@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

Variables declared inside a function are local to that function and are not accessible outside. 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

Try to execeude 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`.

```
# Try to execeude 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 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.

```
# 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 function
@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: Anonymous Functions

Anonymous functions in Julia are unnamed functions, useful for concise and short operations. They are particularly useful for operations that are passed as arguments to higher-order functions or used for short, one-off computations. Syntax for anonymous functions can be either of the following:

```
add_two = (a,b) -> a + b
add_two(a,b) = a + b
```

Exercise 3.1

Create an anonymous function `multiply` that multiplies two numbers.

```
# Create an anonymous function 'multiply' that multiplies two numbers.
# YOUR CODE BELOW

```

```julia
# Test your function
@assert multiply(10, 5) == 50  "The result is $(multiply(10, 5)) but should be 50."
println("Great job! You created an anonymous function that multiplies two numbers.")
```

Section 4: 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:

```julia
# 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)               # Numeric addition.
result2 = operation("Hello, ", "World!")  # String concatenation.
result3 = operation("Hello, ", 20)        # Generic implementation.
```

```
"Generic operation for objects of type String and Int64"
```

Exercise 4.1

Choose the result that should be asserted in the following to equal the expected value on the right side of the conditional statement.

```julia
# Choose the result that should be asserted in the following to equal the expected value
 ↪  on the right side of the conditional statement. Hint: The answer is easy, you just
 ↪  have to change the comparisons.

# YOUR CHANGES BELOW
@assert result1 == "Hello, World!"
@assert result3 == 30
@assert result2 == "Generic operation for objects of type String and Int64"
println("You solved it, the order is now correct!")
```

> Tip
>
> **Hint**: The answer is easy, you just have to change the comparisons.

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.