Tutorial II.IV - Loops

Energy System Optimization with Julia

Introduction

Imagine you need to do the same task many times - like sending emails to 100 people or counting from 1 to 10. Instead of writing the same code over and over, we use loops! Loops are like having a helpful assistant who can repeat tasks for you.

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

Section 1 - For Loops

A `for` loop is like giving instructions to that assistant: "For each item in this list, do this task." For example:

Iterating over a range (1 to 3):

```
for i in 1:3
    println(i)
end
```

```
1
2
3
```

This prints `1`, `2`, and `3`.

Iterating over an array:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits
    println(fruit)
end
```

```
apple
banana
cherry
```

This prints each fruit in the `fruits` array.

The `break` statement can be utilized to exit the loop based on a condition. To check some condition, we can use `if` statements. For example:

```
loop_number = 0
for x in 1:10
    loop_number = x
    println(loop_number)
    if loop_number == 4
        break
    end
end
```

```
1
2
3
4
```

This exits the loop in iteration `4`, as the condition `loop_number == 4` is true here.

We can also chain `if` statements. For example:

```
loop_number = 0
for x in 1:10
    loop_number = x
    if loop_number <= 2
        println(loop_number)
    elseif loop_number == 3
        println("We reached 3!")
    else
        break
    end
end
```

```
1
2
We reached 3!
```

This prints `1`, then `2`, then `We reached 3!`. Afterwards the loop ends, as the `break` statement kicks in.

> **Tip**
>
> Think of a `for` loop like a recipe:
> 1. Start with a collection of items (like numbers 1-10 or a shopping list)
> 2. For each item, follow the instructions inside the loop
> 3. When done with one item, move to the next
> 4. Stop when you've used all items (or when told to break)

Exercise 1.1 - Sum the Numbers from 1 to 5

Sum the numbers from `1` to `5` in a loop. The next lines initialize `sum_numbers` to `0`. The sum you compute should accumulate in this variable.

```
sum_numbers = 0
# YOUR CODE BELOW
```

```
# Test your answer
@assert sum_numbers == 15
println("Sum of numbers from 1 to 5: ", sum_numbers)
```

Exercise 1.2 - Sum Only the Even Numbers from 1 to 10

Sum only the even numbers from `1` to `10`. Again, we initialize a variable `sum_evens` to `0`. The sum you compute should accumulate in this variable.

```
sum_evens = 0
# YOUR CODE BELOW
```

```
# Test your answer
@assert sum_evens == 30
println("Sum of even numbers from 1 to 10: ", sum_evens)
```

> Tip
>
> Hint: You can use the modulo operator `%` to check if a number is even. The modulo operator returns the remainder of the division of two numbers. If a number is divisible by another number, the remainder is `0`. The following example checks if `4` is even: `4 % 2 == 0`.

Exercise 1.3 - Exit the Loop if the Current Fruit is Banana

Iterate over each fruit in the `fruits` array, store the current fruit in `current_fruit`, and exit the loop if `current_fruit` is `banana`. The next lines initialize the `fruits` array and `current_fruit` variable.

```
fruits = ["apple", "banana", "cherry"]
current_fruit = "None"
# YOUR CODE BELOW
```

```
# Test your answer
@assert current_fruit == "banana"
println("The current fruit is: ", current_fruit)
```

Section 2 - While Loops for Conditional Execution

A `while` loop is like giving instructions to your assistant: "Keep doing this task as long as this condition is true." They're particularly useful when the number of iterations is dynamic or unknown in advance. For example:

```
# Keep subtracting 1 until we get below 5
number = 10
while number >= 5
    println("Number is: $number")
    number = number - 1
end
println("Final number: $number")
```

```
Number is: 10
Number is: 9
Number is: 8
Number is: 7
Number is: 6
Number is: 5
Final number: 4
```

Real-world examples:

```
# Keep playing game while player has lives
lives = 3
while lives > 0
    println("Playing game... Lives left: $lives")
    lives = lives - 1
end
println("Game Over!")
```

```
Playing game... Lives left: 3
Playing game... Lives left: 2
Playing game... Lives left: 1
Game Over!
```

```
# Keep filling water bucket until full
current_liters = 0
bucket_size = 5
while true
    println("Adding 1 liter...")
    current_liters = current_liters + 1
```

```
    if current_liters == bucket_size
        break
    end
end
println("Bucket is full!")
```

```
Adding 1 liter...
Adding 1 liter...
Adding 1 liter...
Adding 1 liter...
Adding 1 liter...
Bucket is full!
```

> Tip
>
> Think of a `while` loop like these everyday situations:
> - Keep studying WHILE you don't understand the topic
> - Keep adding ingredients WHILE the recipe isn't complete
> - Keep saving money WHILE you don't have enough

Exercise 2.1 - Subtract from 10 in Increments of 1 Until the Result is Less Than 3

Subtract from `10` in increments of `1` until the result is less than `3`. The next line initializes `current_value` to `10`. The result should be in this variable.

```
current_value = 10
# YOUR CODE BELOW
```

```
# Test your answer
@assert current_value == 2
println("The first value smaller than 3 is: ", current_value)
```

Exercise 2.2 - Find the First Multiple of 7 Greater Than 50

Find the first multiple of `7` greater than `50` using an indefinite loop. The next line initializes `first_multiple_of_7` to `0`. The first multiple should be in this variable.

```
first_multiple_of_7 = 0
# YOUR CODE BELOW
```

```
# Test your answer
@assert first_multiple_of_7 == 56
println("First multiple of 7 greater than 50: ", first_multiple_of_7)
```

> Tip
>
> 'while true … end' constructs an infinite loop. You can exit the loop using a 'break' statement if a condition is met.

Section 3 - Nested Loops

Nested loops are loops inside other loops. Think of it like organizing your closet: - First loop: Go through each shelf - Second loop: For each shelf, go through each item

```
# Checking sizes and colors of t-shirts
sizes = ["Small", "Medium", "Large"]
colors = ["Red", "Blue"]

for size in sizes
    for color in colors
        println("Found $color t-shirt in size $size")
    end
end
```

```
Found Red t-shirt in size Small
Found Blue t-shirt in size Small
Found Red t-shirt in size Medium
Found Blue t-shirt in size Medium
Found Red t-shirt in size Large
Found Blue t-shirt in size Large
```

Exercise 3.1 - Compute the Product of Each Pair of Elements from Two Arrays

Compute the product of **each pair of elements** from two arrays. For example, the product of two numbers `a` and `b` is `a * b`. The next lines initialize `numbers1`, `numbers2` arrays, and the `products` array to store your results.

```
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]
products = []
# YOUR CODE BELOW
```

```
# Test your answer
@assert products == [4, 5, 6, 8, 10, 12, 12, 15, 18]
println("Products of each pair from two arrays: ", products)
```

> Tip
>
> Remember, you can use push!() to append elements to an array.

Section 4 - List Comprehensions

List comprehensions provide a concise way to create lists based on existing lists. They can often replace for loops and are considered more "Julia-like". Basic syntax:

```
[expression for item in collection if condition]
```

For example:

```
# The long way
squares = []
for n in 1:5
    push!(squares, n^2)
end

# The short way (list comprehension)
squares = [n^2 for n in 1:5]
```

Both create `[1, 4, 9, 16, 25]`, but the second way is more concise!

Exercise 4.1 - Create a List of Even Numbers

Create a list of even numbers from 1 to 10 using a list comprehension.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert even_numbers == [2, 4, 6, 8, 10]
println("Even numbers from 1 to 10: ", even_numbers)
```

Conclusion

Great work! You've successfully navigated through the basics of loops in Julia. You've seen for and while loops, tackled iterable structure, and worked on nested loops. 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.