Tutorial I.II - Vectors and Matrices

Optimization with Julia

Introduction

This interactive Julia script introduces the basics of vectors, matrices, and tuples.

- A vector is like a single row in a spreadsheet or a shopping list - it stores items in a line
- A matrix is like a spreadsheet - it has rows and columns
- A tuple is like a sealed package containing different types of items

Understanding these data structures and how to manipulate them is crucial in Julia. Follow the instructions, write your code in the designated code blocks, and execute the corresponding code cell.

> **Note**
>
> If a cell is marked with `YOUR CODE BELOW`, you are expected to write your code in that cell.

Section 1 - Vectors

Vectors in Julia are one-dimensional arrays used to store sequences of elements. They're powerful for numerical operations and data handling. A vector is the simplest way to store a list of items. Think of it as a row of boxes, where each box can hold a number or other type of data. Create vectors with square brackets, separate elements with commas.

```
# Creating a simple vector (list) of numbers
grades = [95, 87, 91, 78, 88]
```

```
5-element Vector{Int64}:
 95
 87
 91
 78
 88
```

```
# Creating a vector of names
students = ["Mike", "Yola", "Elio"]
```

```
3-element Vector{String}:
 "Mike"
 "Yola"
 "Elio"
```

Vectors are mutable, and you can:

- Add items to the end (like adding to a shopping list): `push!(grades, 82)`
- Remove the last item (like crossing off the last item): `pop!(grades)`
- Remove the first item (like crossing off the first item): `popfirst!(grades)`
- Look at specific items using their position number: `grades[1]` gives you the first grade
- Access a range of items: `grades[1:3]` gives you the first three grades

> **Tip**
>
> In Julia, we start counting positions from 1, not 0. So `grades[1]` gives you the first grade!

Use '?' in the REPL for function details.

Exercise 1.1 - Create a Vector

Create a vector 'fib' with the first five Fibonacci numbers: `1, 1, 2, 3, 5`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert fib == [1, 1, 2, 3, 5]
println("The 'fib' vector: ", fib)
```

Exercise 1.2 - Append to a Vector

Append the number 8 to the `fib` vector.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert fib == [1, 1, 2, 3, 5, 8]
println("The 'fib' vector after appending 8: ", fib)
```

Exercise 1.3 - Remove the First Element

Remove the first element of the `fib` vector.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert fib == [1, 2, 3, 5, 8]
println("The 'fib' vector after removing the first element: ", fib)
```

Exercise 1.4 - Access and Save the First Three Elements

Access and save the first three elements of `fib`, to `first_three_elements`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert first_three_elements == fib[1:3]
println("The first three elements of the 'fib' vector: ", first_three_elements)
```

Section 2 - Matrices

A matrix in Julia is a 2D array, great for linear algebra and data representation. Create matrices with square brackets, separate elements with spaces, rows with semicolons. Access elements with square brackets (e.g., `matrix[2,2]`). You can add or substract matrices of the same dimensions element-wise if you add or substract them.

For example:

```
matrix1 = [2 2; 3 3] + [1 2; 3 4]
matrix2 = [2 2; 3 3] - [1 2; 3 4]
println("matrix1 is $matrix1.")
println("matrix2 is $matrix2.")
```

```
matrix1 is [3 4; 6 7].
matrix2 is [1 0; 0 -1].
```

If you want to perform multiplications, things are a bit different, as matrix multiplications are not performed element-wise!

```
matrix3 = [2 2; 3 3] * [1 2; 3 4]   # Not element-wise
println("matrix3 is $matrix3.")
```

```
matrix3 is [8 12; 12 18].
```

But we can change this by using broadcasting!

> **Tip**
>
> Julia's broadcasting feature allows you to apply functions element-wise to arrays of different sizes. This is denoted by adding a dot (.) before the operator or function. It is extremely useful for performing operations on arrays without the need to loop through each element.

```
matrix4 = [1 2; 3 4]
matrix4 = matrix4 .+ 1
println("matrix4 is $matrix4.")
```

```
matrix4 is [2 3; 4 5].
```

We can also use this to perform element-wise multiplications, as shown below:

```
matrix5 = [2 2; 3 3] .* [1 2; 3 4]  # Element-wise
println("matrix5 is $matrix5.")
```

```
matrix5 is [2 4; 9 12].
```

Exercise 2.1 - Create a Matrix

Create a 2x3 matrix `my_matrix` with the values: `1 2 3; 4 5 6`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert my_matrix == [1 2 3; 4 5 6]
println("The 'my_matrix':\n", my_matrix)
```

Exercise 2.2 - Change the 3rd Column of the 2nd Row

Change the 3rd column of the 2nd row to `17` by accessing and changing the element.

```
# YOUR CODE BELOW

```

```
# Test your answer
@assert my_matrix == [1 2 3; 4 5 17]
println("The 'my_matrix' after modification:\n", my_matrix)
```

Exercise 2.3 - Perform Matrix Addition

Perform matrix addition with `my_matrix` and `another_matrix`. Call the resulting matrix `added_matrices`.

```
another_matrix = [10 20 30; 40 50 60]
# YOUR CODE BELOW
```

```
# Test your answer
@assert added_matrices == [11 22 33; 44 55 77]
println("Result of adding 'my_matrix' and 'another_matrix':\n", added_matrices)
```

Exercise 2.4 - Add 10 to Each Element

Add `10` to each element in `added_matrices`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert added_matrices == [21 32 43; 54 65 87]
println("Result of adding 10 to each element in 'added_matrices':\n", added_matrices)
```

Section 3: Tuples

A tuple is like a sealed package - once you create it, you can't change what's inside. That's why we call it "immutable". It's perfect for grouping related items that shouldn't change. For example:

```
# A person's basic info that won't change
person = ("Elio Smith", 18, "Hamburg")

# RGB color values
red = (255, 0, 0)
```

Think of tuples as permanent labels - once you write them, they can't be changed. This is useful when you want to make sure data stays exactly as you set it.

Exercise 3.1 - Create a Tuple

Create a tuple `my_tuple` with three elements: `4.0`, your matrix `added_matrices`, and `"Hi there!"`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert my_tuple == (4.0, added_matrices, "Hi there!")
println("The 'my_tuple': ", my_tuple)
```

Exercise 3.2 - Access the Second Element

Access the second element of `my_tuple`, store it in `second_element`.

```
# YOUR CODE BELOW
```

```
# Test your answer
@assert second_element == [21 32 43; 54 65 87]
println("The second element of 'my_tuple':\n", second_element)
```

Conclusion

Well done! You've completed the tutorial on Vectors, Matrices, and Tuples. You've learned to create, manipulate, and interact with these fundamental data structures. Experiment with the code, try different operations, and see how Julia behaves. Continue to the next file to learn more.

Solutions

You will find the solutions to all exercises online here in the `solutions` folders for each part. 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.