Assignment 1

Daniel Friis-Hasché, rcb933

27/09/2024

1 Vectors

1.1 Questions

1.1.1 a) What type do you use to represent a 2D vector in F#? Why? (2 points)

In my program, I use floats encapsulated in a tuple, to represent the coordinates of the vectors. I do this to more accurately represent the vectors coordinates, as integers would round off some digits and therefore not correctly show the answers created by the function. Floats more accurately represent the different values and coordinates of vectors, and tuples show which numbers corresponds to which vector.

1.1.2 b) What names do you give to these three functions? Why? (2 points)

For all the functions in my vector.fsx program, I decided on names that would easily describe their purpose in the program. The names I have chosen are **vectorLength**, **vectorAdd** & **vectorScale**.

1.1.3 c) Write the specification of the three functions. (2 points)

The three functions take input in the form of vector coordinates, modify them and outputs new coordinates for the different vectors. The in- and outputs of the differenct functions vary, given that some functions use multiple vectors where others only use a single.

1.1.4 d) Give two examples (in F#) for each of the three vector functions. (2 points)

For the **vectorLength function**:

```
let vectorLength (a:float, b:float) =
    sqrt(a**2 + b**2)

printfn "The length of vector 1 is (%.3f) \nThe length of vector 2 is (%.3f)\n" (
    vectorLength (1, 1)) (vectorLength (2, 4))

// Output:
"The length of vector 1 is (1.414)"
"The length of vector 2 is (4.472)"
```

The **vectorLength** function take as input, vector coordinates, as arguments named a & b, raise both of them to the power of 2, adds them together, and takes the square root of the results. The output here will be a single number (which I have chosen to represent in floats, as it is more precise that way) that shows the length of the coordinate.

For the **vectorAdd function**:

```
let vectorAdd (a:float, b:float) (c:float, d:float) =
    (a + c), (d + b)

printfn "Vector 1 & 2 added results in %A \nVector 3 & 4 added results in %A" (
    vectorAdd (10.0, 5.0) (-2.0, 8.0)) (vectorAdd(6.0, 8.0) (13.0,20.0))

```

```
// Output
"Vector 1 & 2 added results in (8.0, 13.0)"
"Vector 3 & 4 added results in (19.0, 28.0)"
```

The **vectorAdd** function take as inputs, vector coordinates for a single vector, as well as an argument for scaling the vectors coordinates. The arguments are named *a*, *b*, & *scaleValue*. The function multiplies both coordinates of the vector, with the given scaleValue, and outputs a new vector with the scaled coordinates.

For the **vectorScale function**:

```
let vectorScale (a: float, b:float) (scaleValue:float) =
    (a * scaleValue), (b * scaleValue)

printfn "The scale product of the two vectors given, is %A \n" (vectorScale(2.0,
    3.0) (2.0))

// Output
"The scale product of the two vectors given, is (4.0, 6.0)"
```

The **vectorScale** function take as inputs, vector coordinates for two vectors, as arguments named *a*, *b*, *c* & *d*, adds all first coordinates together in the form of a tuple, and in another tuple add together both second coordinates. The outputs here is are new vectors based on the previous coordinates.

1.1.5 e) Give the F# code for these functions and explain in detail how you design them. (20 points)

```
let vectorLength (a:float, b:float) =
    sqrt(a**2 + b**2)

printfn "\nThe length of vector 1 is (%.3f) \nThe length of vector 2 is (%.3f)\n" (
    vectorLength (1, 1)) (vectorLength (2, 4))

// Output
"The length of vector 1 is (1.414)"
"The length of vector 2 is (4.472)"

let vectorAdd (a:float, b:float) (c:float, d:float) =
    (a + c), (d + b)

printfn "Vector 1 & 2 added results in %A \nVector 3 & 4 added results in %A" (
    vectorAdd (10.0, 5.0) (-2.0, 8.0)) (vectorAdd(6.0, 8.0) (13.0,20.0))

// Output
"Vector 1 & 2 added results in (8.0, 13.0)"
"Vector 3 & 4 added results in (19.0, 28.0)"

let vectorScale (a: float, b:float) (scaleValue:float) =
    (a * scaleValue), (b * scaleValue)

printfn "The scale product of the two vectors given, is %A \n" (vectorScale(2.0,
    3.0) (2.0))

// Output
"The scale product of the two vectors given, is (4.0, 6.0)"
```

The entire script has 3 different functions, where each of the functions manipulate given arguments (mostly vector coordinates, sometimes some other arguments used to modify the coordinates), and outputs new coordinates (or length of a coordinate) in the terminal where the program are executed. I have chosen to make the functions in this way, to keep as much of the input in the print statements in order to keep it simple and readable for other people reading through my code. The chosen variables are mostly taken from the mathematical world, but for simplicity's sake, changed a bit (e.g. instead of x1, x2, I used a, b, c & d instead). I have also chosen to separate the declaration and execution part

of the functions into separate lines, again to ensure readability.

1.1.6 f) Explain how your code would change if you were to represent 3D vector instead of 2D vectors? (2 points)

All functions in my code would break, if I were to input a third argument in the print statement without changing the functions themselves. Most functions (except for the vectorScale funciton) are "hard coded" to only use the formula given, and not look for whether or not a third dimension has been added or not.
If I were to make a *perfect* script, I would probably bind the entire thing in if-statements, and execute different functions depending on the amount of arguments given.

2 Survey

answerSorter function to look through a specific question, but in doing so I have hard coded the amount of items in the tuple that encapsulates the student's ID and answers. This means that adding multiple questions, removing questions or changing in the order in the tuple would result in either breaking the code entirely or outputting an incorrect result (e.g. getting the UUID's as answers to the questions)

2.1 Questions

2.1.1 a) Describe two possible ways to represent the survey answers in F#. Also, give F# code with two examples that illustrate the representations you have chosen. (10 points)

To represent survey answers, we would have to use a list of some form. In the list, there would have to be separations for each participant, so it's easier to match the participants with their answers, and navigate through all answers bound to said participant. The two ways I would do this, is to either put a list within a list, e.g.:

```
    let list1 =
    [
        [0;1;2;3]
        [1;1;2;3]
        [2;1;2;3]
    ]
```

Where you could have the first index in the list be the ID/name of the participants, and the rest of the indexes be their answers. The other way could be to have tuples within a list. The upside to doing this would be the ability to combine different types (e.g. names and answers being strings and integers) in the same list. In other languages this would also be possible to do in a single list, but in F# we would have to do it with tuples. It could look like this:

```
    let list2 =
    [
        ("ID1";1;2;3)
        ("ID2";1;2;3)
        ("ID2";1;2;3)
    ]
```

You could also flip it if the participants' answers were in the form of strings.

In my script I have chosen to use tuples in a list, to easier represent the ID's of the participants. It looks like this:

```
let survey =
[
    // 0(ID), 1, 2, 3, 4, 5
    ("UUID1", 4, 4, 3, 3, 1)
    ("UUID2", 2, 3, 1, 4, 3)
    ("UUID3", 1, 1, 2, 1, 4)
    ("UUID4", 2, 1, 4, 3, 2)
    ("UUID5", 1, 4, 3, 4, 2)
    ("UUID6", 4, 2, 2, 1, 3)
    ("UUID7", 1, 3, 2, 1, 4)
    ("UUID8", 2, 4, 3, 2, 1)
    ("UUID9", 3, 3, 1, 2, 4)
]
```

2.1.2 b) Design a function that counts the number of students that have given a specific answer to a given question. For example: how many students have given answer 2 to question 3? Explain your design using Ken's method, include your F# code and describe it (10 points)

Both question **b)** & **d)** have been answered here, because they use the same function.

To get the participants that have answered the same to a specific question in the survey, we should make a function that filters through the given answers, to see the participants who gave the same answer. The function should then make a new list, that only shows the ID's of the participants which have given the chosen answer. Lastly, we would want a print statement to show the participants' ID's in the terminal, for the end user to see.
The name of the function should explain what the function does, so it's easy to spot. I have chosen to use `answerSorter` as the name.
If two participants answered "1" in question 2, the function should find the ID's of those students, and display them in a new list, which should then be printed out. Here the input is given in the form of the original survey, which is in list format (that incapsulates tuples), the question we would like to look through and what answer we want to check for. The output here is both a list with the ID's of the participants, and the length of the new list with ID's.

The code I have made looks like this:

```
let answerSorter =
    survey
    |> List.filter (fun (_, _, answer2, _, _, _) -> answer2 = 1)
    |> List.map (fun (uuid, _, _, _, _, _) -> uuid)


printfn "\nThe students that answered question 2 with 1 as their answer \nare: %A,
    which totals to: %A students\n" answerSorter answerSorter.Length

// Output
"The students that answered question 2 with 1 as their answer are: ["UUID3"; "UUID4"
    ], which totals to: 2 students"
```

The function, *answerSorter* pulls the original survey, listed earlier in the assignment. It then applies a pipeline with the filter function, which is set to look through the tuples of the survey, and check all answers in question two (with index 2 (ID, question1, question2, ...)). If the answers given match 1, the filter pipeline will make a new list only containing all tuples where the filter function is true (e.g. if the answer is 1, a boolean for the given tuple would be set to true).
After the filter has been applied, a new pipeline with List.map is run, which takes the now filtered list, and make a new list only containing the first element of the tuples, which is the ID's of the participants. Lastly, a print statement is run to both print out the ID's, as well as the length of list with ID's.

2.1.3 c) Design a function that returns the percentage of student answers for all answers to a given question. Explain your design using Ken's method, include your F# code and describe it (15 points)

We want a function that can look through the list we had in the previous example, and see which participants gave a specific answer to a specific question. The function should then filter through those answers and show the end user (in this case us) who answered with the same.
The naming in the function should be easy, and readable for anyone who hasn't had a play in making the code. The names should therefore short, but thoroughly describe what their given part does (e.g. variables, arguments or functions).
The function should be recursive take input in the form of the original list containing survey answers grouped. It should also include a form of start value, $\boldsymbol{n}$, which we can manipulate in order to iterate over each answer given (we want to do this in order to show all answers given, and not a single one). The function should then iterate over each given answer, take the length of how many answered with the given answer, and call itself with the value $\boldsymbol{n+1}$. The function should run until it reaches the end of the list, and should always start at 0. At each iteration, the length of the given answers should be divided by the length of the entire survey, and then multiplied with 100, to get the result in percentage. The output should finally be each result printed out in the terminal for the end user to read.
Here is the code I have made:

```
let groupedAnswers =
    survey
    |> List.groupBy (fun (_, _, answer2, _, _, _) -> answer2)
```

```
4
5  let rec percentageSorter n =
6      match n with
7      | _ when n >= groupedAnswers.Length -> exit 0
8      | _ when n < 0 -> percentageSorter(0)
9      | _ ->
10     printfn "Percentage of students who answered %A, is %.3f %%" (fst groupedAnswers
           [n]) (float (snd groupedAnswers[n]).Length/float survey.Length*100.0)
11     percentageSorter (n+1)
12
13
14 printfn "%A" (percentageSorter 0)
15
16 // Output
17 "Percentage of students who answered 4, is 33.333 %"
18 "Percentage of students who answered 3, is 33.333 %"
19 "Percentage of students who answered 1, is 22.222 %"
20 "Percentage of students who answered 2, is 11.111 %"
```

The function combs through the survey (in list that encapsulates 9 tuples), and groups each answer by using List.groupBy to an unnamed function that specifically looks through question 2, and groups each tuple by their given answer. The list consist of an ID (which I called UUID, not for any specific reason, it is however what is used for Minecraft Usernames, which was the first thing that popped into my mind) in string format, and answers to each question in integer format. After I have grouped all answers, I comb through it using a recursive function with input ***n***. The function matches the value n, with different cases, and checks to see whether

- n is bigger than or equal (¿=) to the length of the grouped answers (in this case 4, since answers could be 1, 2, 3 or 4). If the case is true, and n is out of bounds, the program will exit with exitcode 0 (a random exitcode I chose).
- If n is less than 0, the function will run again, this time however with input of `n = 1`, to ensure the script will not break if n is too low.
- If nothing matches the two other cases, the program will print out a string that calls the recursive function's first index in the tuple of answers, here with the current value of n - which corresponds to which answer it should look through. It also calls the function again, but here with the second index of the tuple, which corresponds to all answers for the specific answer (e.g. the first is answer [n], the second is all the answers given with that answer, ([n], [n], [n], ...)). It then takes the length of the list containing the answers, divide it with the length of the original survey list, and multiply it by 100, to get the percentage. Lastly, the recursive function is run again, here with `n+1`, to iterate to the next answer.
  Note that the answers in the groupedAnswers variable are sorted from most common occurrence to least common occurrence

2.2 Concepts and syntax

2.2.1 a) What is a function (5 points)?

A function is an operation, either a computation or an action, that takes some form of input and modifies it. Generally speaking, a function executes whatever is put inside it, be it some mathematical operations, printing statements or even callings to other functions.

2.2.2 b) What is type inference (5 points)?

Type inference is when a type is bound to a form of value, e.g. an integer to the value 4, or a boolean to the "value" *true*. F# automatically does this, but it can also be manually done.

2.2.3 c) What are the advantages of functional programming based on what you have seen during the exercises (10 points)?

I personally think the biggest advantage of the functional programming paradigm, over ex. the imperative programming paradigm, is the knowledge you gain about data-types & how to use them. In F#

you "punished" if you use a wrong datatype, since you have to be 100% in-the-know about which types are given as input and used throughout the function, which forces you to learn how to use data-types, and constantly checks on you to see if you are doing it correctly. This also results in the developer having a more in-depth understanding of why they get the output they get.
I also personally believe pipelines are a major advantage over the other paradigms, but that might just be because I haven't tried it in other programming languages.
Generally data-types, higher order functions and pipelines are some of the biggest advantages.