4 Tuples

Key concept:

1. Tuple is a good data-structure for many things:

   (a) To represents terms that goes together, e.g. 2-D coordinates, Year-Month, etc.

   (b) For testing out ideas quickly. (use other data-structures after testing)

2. You can directly extract the content of a tuple, and use underscore "_" to ignore any part of the tuple that you don't need.

3. `List.concat` is hidden inside some other useful functions.

4.1 Tuples

A 2D-coordinate may look like this:

```
let point1 = (1.0, 2.0)
let point2 = (3.0, 4.0)
```

Hover your mouse on top of these two objects. Notice that the signature is `float * float`. So these points have two coordinates, each of them are `float` or `double`

```
let DistanceFromOrigin point =
    let (x,y) = point          // Data extraction process!
    sqrt (x ** 2.0 + y ** 2.0)

let distance1 = DistanceFromOrigin point1
let distance2 = DistanceFromOrigin point2
```

Output:

```
// val distance1 : float = 2.236067977
// val distance2 : float = 5.0
```

Notice that we have an extraction process `let (x,y) = point` that helps us extract the contents of `point` (and save the contents into the variables `x,y`). In fact, we can directly do the extraction process in the function definition:

```
let DistanceFromOrigin2 (x,y) =
    sqrt (x ** 2.0 + y ** 2.0)
```

Tuples of Different Type

We can mix tuples of different type (compared to list, which cannot contain elements of different types).

```
let mixedTuple1 = (1.0, "HELLO")
let mixedTuple2 = (1, "Hello", true)
```

If you hover your mouse on top of these, you will see that:

- The first tuple has signature `float * string`
- The second tuple has signature `int * string * bool`

As before, we can extract the contents of the tuple using `let`.

```
let (extractedDecimal, extractedString) = mixedTuple1
let (a,b,c) = mixedTuple2
```

Output:

```
// val extractedString : string = "HELLO"
// val extractedDecimal : float = 1.0

// val c : bool = true
// val b : string = "Hello"
// val a : int = 1
```

If you only want to extract part of a tuple, you can use the underline "_" to ignore any part of the tuple that you don't need.

```
let personalInfo = ("John", 21, 170.0)

let (extractedName,_,_) = personalInfo
// val extractedName: string = "John"
```

Example

You are given data about the number of student in each class. The data is saved in a `List<string * int>`. e.g. in the first list, Class A has 50 students, Class B has 40 students, etc.

```
let studentList =
    [("A",50); ("B", 40); ("C", 45); ("D", 48)]
let studentList2 =
    [("A", 40); ("B", 30); ("C", 20); ("D", 25); ("E", 29);
    ("F", 50)]
```

The following function helps to find the total number of students in those school:

```
let TotalStudent (studentList: List<string * int>) =
    studentList
    |> List.map (fun classInfo ->
        let (_,numStudent) = classInfo
        numStudent
    )
    |> List.sum

let totalStudent1 = TotalStudent studentList
```

Output:

```
// val totalStudent1 : int = 183
```

Of course, we can directly do the extraction process in the function definition:

```
let TotalStudentVersion2 (studentList: List<string * int>)
   =
    studentList
    |> List.map (fun (_,numStudent) -> numStudent)
    |> List.sum

let totalStudent2 = TotalStudentVersion2 studentList
```

Output:

```
// val totalStudent2 : int = 194
```

Exercise

You are given data about how each student score in a class. e.g. In this class, Ali scored 85.0 points, Baba scored 95.0 points, etc.

```
let classScore1 =
    [("Ali", 85.0); ("Baba", 95.0); ("Charlie", 87.0); ("Dan", 92.0); ("Emily", 96.0); ("Fiona", 92.0)]
```

Write a function that accepts a list of names with their scores, and return the class average.

```
let ClassAverage (scores: List<string * double>) =







    // Implement your function here.
    // Hint: List.map and List.average
```

Example

A country currently wants to implement a new tax system:

- `COMMON`: 5% tax
- `IMPORTS`: 10% tax
- `ALCOHOL`: 20% tax

A supermarket wants currently saves the data in a `List<string * double * string>`, where the first `string` is the product, the `double` is the original price before tax, and the last `string` is the product code. e.g.

```
let productList1 =
    [("Bread", 2.40, "COMMON");
     ("Beer", 10.20, "ALCOHOL");
     ("Swiss Chocolate", 8.20, "IMPORTS");
     ("Rice", 20.50, "COMMON");
     ("Red Wine", 30.00, "ALCOHOL");
     ("Australian Beef", 18.50, "IMPORTS")]
```

The following code will help calculate the total price after tax:

```
let TotalAfterTax (productList: List<string * double * string> ) =
    productList
    |> List.map (fun tuple ->
        let (_,priceBeforeTax,productType) = tuple
        // Data Extraction above!

        if productType = "COMMON" then
            1.05 * priceBeforeTax
        else if productType = "ALCOHOL" then
            1.20 * priceBeforeTax
        else
            1.10 * priceBeforeTax
    )
    |> List.sum

let totalPrice = TotalAfterTax productList1
printfn "The final price after tax is: %.2f" totalPrice
```

Output:

```
The final price after tax is: 101.66
```

Again, we can move the extraction process into the function definition:

```
let TotalAfterTaxVersion2 (productList: List<string * double * string> ) =
    productList
    |> List.map (fun (_,priceBeforeTax,productType) ->
        if productType = "COMMON" then
            1.05 * priceBeforeTax
        else if productType = "ALCOHOL" then
            1.20 * priceBeforeTax
        else
            1.10 * priceBeforeTax
    )
    |> List.sum
```

Notice that the values are extracted immediately after the `fun` keyword.

Exercise

A clothing store is planning to do a discount sale:

- `CLEARANCE`: 50% off.
- `SHIRT`: 30% off.
- `JEANS`: 20% off.

You are given a `List<string * double>` that represents an item's product code and their original price. e.g. the customer below bought a clearance item, two shirts and two jeans.

```
let listOfClothes =
    [ ("CLEARANCE", 70.0); ("SHIRT", 20.0); ("SHIRT", 40.0); ("JEANS", 55.0); ("JEANS", 79.9)]
```

Write a function that takes a list of items and their original price, and return the total price after discount.

```
let TotalAfterDiscount (priceList: List<string * double>) =










    // Implement your function here.
```

The expected final price after discount is $184.92

4.2 All Pairs

We have the List.allPairs function in F# 4.1. (If you are using an earlier version of F#, then you may need to implement the function yourself using other functions, see the next section of this guide).

```
let allPairs1 = List.allPairs [1;2;3] ["A";"B"]
```

Output:

```
// val allPairs1 : (int * string) list =
// [(1, "A"); (1, "B");
//       (2, "A"); (2, "B");
//           (3, "A"); (3, "B")]
```

Example

Given two lists S_1 and S_2, we want to find the sum of all products $a \times b$, where $a \in S_1, b \in S_2$.

```
let SumOfAllPairProducts list1 list2 =
    List.allPairs list1 list2
    |> List.map (fun (x,y) -> x * y)
    |> List.sum

let list1 = [1;2;3]
let list2 = [5;6]

let result2 = SumOfAllPairProducts list1 list2
// val result2 : int = 66
```

We can also verify mathematically:

$$\sum_{x\in S_1}\sum_{y\in S_2} x\cdot y = \sum_{x\in S_1}\left[x\cdot\left(\sum_{y\in S_2} y\right)\right] = \left(\sum_{y\in S_2} y\right)\cdot\left(\sum_{x\in S_1} x\right)$$

$$(1+2+3)\cdot(5+6) = 6\times 11 = 66$$

Exercise (Euler Project Question 9)

https://projecteuler.net/problem=9

Find the only Pythagorean triplet a, b, c that satisfy:

$$a < b < c, \qquad a + b + c = 1000, \qquad a^2 + b^2 = c^2$$

Hints:

For $1 \leq a \leq 1000, 1 \leq b \leq 1000$, let $c = 1000 - a - b$. Then select (a, b) such that:

$$c > 0 \qquad a^2 + b^2 = c^2$$

```
let FindPythagoreanTriple =
    List.allPairs [1 .. 1000] [1 .. 1000]
    // |> List.filter (fun (a,b) ->
    //         let c = ......
    //         ..............)
```

Expected answer: $a = 200, b = 375$, and so $c = 1000 - 200 - 375 = 425$. And so $a \times b \times c = 31875000$.

You can submit your answer online for personal achievement/accomplishment.

Remark: The pipe-forward operator `|>` can only pipe forward one object/item. It cannot pipe-forward two items. And so, the following code will not work:

```
let FindPythagoreanTriple =
    [1 .. 1000] [1 .. 1000]
    |> List.allPairs    // Error! Cannot pipe two objects.
    |> List.filter (fun (a,b) ->
           let c = ......
           ..............)
```

Exercise (Euler Project Question 4)

https://projecteuler.net/problem=4

A palindromic number reads the same from left-to-right or right-to-left.

The largest palindromic number made from the product of two 2-digit numbers is $9009 = 91 \times 99$.

Find the largest palindrome made from the product of two 3-digit numbers.

You can use the following `IsPalindrome` function that is already implemented for you. You do not need to re-implement it.

```
let ReverseString (xString: string) =
    new string (xString.ToCharArray() |> Array.rev)

let IsPalindrome (x:int) =
    let xString = x |> string
    (ReverseString xString) = xString

let palindromeResult1 = IsPalindrome 1234
let palindromeResult2 = IsPalindrome 16761
// val palindromeResult1 : bool = false
// val palindromeResult2 : bool = true
```

Find the largest palindrome number which is a product of two 3-digit numbers $a \times b$, where $100 \leq a \leq 999$, and $100 \leq b \leq 999$

```
let findProductPalindrome =
    List.allPairs [100 .. 999] [100 .. 999]
    // |> List.map (fun (a,b) -> ..............)
    // |> ..............



    failwith "NOT YET IMPLEMENTED!"
```

Expected answer: 906609

Again, you can submit your answer online for personal achievement/accomplishment.

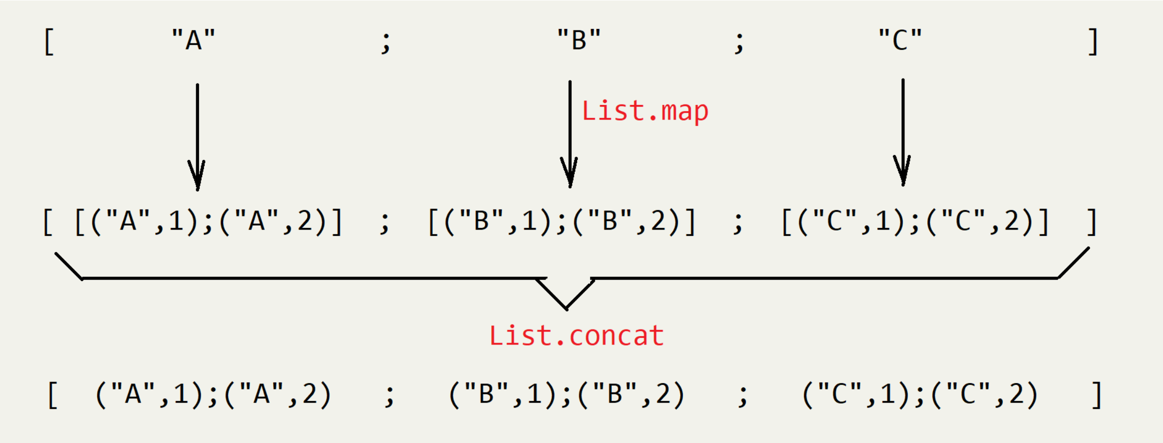
4.3 (Optional) Implement AllPairs yourself

`List.allPairs` is a function available in F# 4.1. It may not be available in earlier versions. However, you can implement this yourself: Let's say that:

```
let list1 = ["A";"B";"C"]
let list2 = [1;2]

let expectedResult = List.allPairs list1 list2
// [("A",1);("A",2); ("B",1);("B",2); ("C",1);("C",2)]
```

Hint:



["A" ; "B" ; "C"]
List.map
[[("A",1);("A",2)] ; [("B",1);("B",2)] ; [("C",1);("C",2)]]
List.concat
[("A",1);("A",2) ; ("B",1);("B",2) ; ("C",1);("C",2)]

```
// self-defined version
let AllPairs list1 list2 =
    list1
    |> List.map (fun x ->
        ......
        ......)
    |> List.concat
```

Notice that there are two layers of `List.map`, the outer layer converts `["A";"B";"C"]` to the huge nested `List<List<_>>`, and the inner layer that converts `[1;2]` to `[("B",1);("B",2)]`

`List.concat`

Conceptually/theoretically, `List.concat` is much more interesting than `List.allPairs`.