```
    // See below for other list functions

let addedToList = 1 :: aList // :: adds an element to the front of the list, so this evaluates to [1; 2; 3; 4; 5]
let concatTwoLists = [0; 1] @ addedToList // @ concatenates two lists together, so this evaluates to [0; 1; 1; 2; 3; 4; 5]
let aListOfRange = [1..10] // .. defines an inclusive range within the list, so this evaluates to [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
let aListFromSeqExp = [for i in 1..10 -> i * i] // lists can be defined with sequence expressions as well, as seen here this creates a list of squares of integers 1 to 10, which evaluates to [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
let aListFromListComprehension = [for i in 1..10 do yield i * i] // basically the same as above but achieved via list comprehension

// --------- ARRAY ----------
    // arrays are mutable and considered more efficient
    // ordered collection of elements of the same type
    // arrays are wrapped with [||] square brackets with bars and ; semicolon delimited

let anArray = [|"a"; "b"|]

// ---------- SEQUENCE ----------
    // infinite sequence of elements
    // basically an enumerator
    // sequences are wrapped with {} curly braces and ; semicolon delimited and accompanied with seq
    // yield is used to generate sequence values where required

let aSequence = seq {yield "a"; yield "b"}

// ---------- TUPLE ----------
    // collecton of elements of any data type
    // tuples are anonymous by default
    // , comma delimited
    // can be unpacked with pattern matching similar to Rust

let aTuple = 1,2
let anotherTuple "a", 2, true

let x,y = aTuple // this unpacks the tuple and sets the value of x = 1, y = 2
```

Functions

```
// ---------- FUNCTION ----------
    // let defines a function, which is defined similar to a variable (since we're in functional land where every statement is an expression that evaluates to a value including functions)
    // no brackets and implicit return of last expression similar to Haskell
    // functions are first class entitites and can be chained to create powerful constructs
    // let {FUNCTION NAME} {FUNCTION PARAMETERS} = {FUNCTION PROCEDURES which are implicitly returned}
    // multiline functions can be defined with indents
    // () brackets can be used to define function precedence
    // |> pipe operator also available to pipe the output of one operation to another, this is very common in F#
    // anonymous functions (lambdas) can be defined with the fun keyword
    // modules group functions together (indentation necessary for each nested module)

let square x = x * x // note that parameter and return values are effectively non-distinguishable
square 3 // evaluates to the value of 9

let add x y = x + y // implicit return of the x + y calculated value
add 2 3 // runs the function, evaluates to the value of 5

// MULTILINE FUNCTION
    // mainly used for greater readibility

let evens list =
    let isEven x = x % 2 = 0 // defining the sub function isEven within the multiline function evens so that it can only be referenced within the evens function, also the first = is an assignment of an expression, the second = is an equality check
    List.filter isEven list // built-in function List.filter then called on each value of the list parameter received by evens function based on the conditional check laid out in the defined sub function isEven

evens [1; 2; 3; 4; 5] // this runs the above function as expected

// FUNCTION PRECEDENCE
    // () define function precedence, and help readibility

let sumOfSquaresTo100 = List.sum ( List.map square [1..100] ) // here, the brackets specify the contents of List.map is to be called on the in t list defined for range 1-100, mapping the square function on each value, the returned value then being passed to List.sum

// PIPING
    // |> lets you pipe

let sumOfSquaresTo100WithPipes = [1..100] |> List.map square |> List.sum // this does the same thing as above
```