General

- Curly braces are NOT used to delimit blocks of code. Instead, indentation is used (like Python).
- Whitespace is used to separate parameters rather than commas.

Other keywords:

- "let" is used instead of "var"
- "let" is also used for defining functions
- "type" is used instead of "class", "enum", etc

Symbols

- "=" is used instead of "=="
- "<>" is used instead of "!="
- "not" is used instead of "!"
- In parameters, commas are replaced by whitespace
- In non-parameter usage (eg lists), commas are replaced by semicolons in most places.

Functions

```
// define a function
let printName myName =
    printfn "my name is %s" myName

// call the function
printName "Scott"

// define a function
let add x y =
    x + y    // no return needed
```

Piping "|>" is the pipe symbol

```
// piping passes the left side to the LAST parameter
"Scott" |> printName

// piping passes the left side to the LAST parameter
2 |> add 1

// piping can be used to connect a sequence of actions
add 1 2
|> add 3
|> printfn "1 + 2 + 3 = %i"
```

Type annotations

Not normally needed but might be helpful when you are a beginner.

How to add type annotations to a function:

```
let addInt (x:int) (y:int) :int =   //last one is the return type
    x + y

let addString (x:string) (y:string) :string =
    x + y

let intToString (x:int) :string =
    x.ToString()

// printing
printfn "an int: %i | a string: %s | a float: %g | a bool: %b |
an F# type: %A" 1 "hello" 3.14 true [1;2;3]
```

Pattern matching

```
let matchInt i =
    match i with
    | 1 -> printfn "One"
    | 2 -> printfn "Two"
    | _ -> printfn "other"  // "_" is a wildcard
```

Function signatures

Function signatures are like this

```
//    paramType -> returnType                // one param fn
//    paramType -> paramType -> returnType   // two param fn

// try it
let add1 x = x + 1        // val add1 : x:int -> int
let plus x y  = x + y     // val plus : x:int -> y:int -> int
```

The unit type

The "unit" type is like void, sort of. It is written "()" and means no output or no input

```
// E.g. print signatures are like this
//    paramType -> unit   // a one param fn returning nothing

// try it
printfn "hello %s"        // string -> unit
```

Generic types

Generic types are written 'a, 'b etc. Equivalent to <T> <U> in C#

```
// try it
let same x = x            // val same : x:'a -> 'a
```

Basic types: Tuples (pairs,triples)

```
let myPair = 1,2      // pair
let myTriple = 1,2,3  // triple

// How many parameters does this function have?
let tupleExample1 (x,y,z) = x + y + z
// How many parameters does this function have?
let tupleExample2 x y z = x + y + z

// Can I call the functions like this?
tupleExample1 1 2 3
tupleExample2 (1,2,3)
```

Basic types: Records

"{" is used for defining record types and constructing record values.

```
type MyRecordType = {a:int; b:string}
let myRecordValue = {a=1; b="hello"}

// you can copy all the fields but some like this
let cloneMyRecordValue = {myRecordValue with b="goodbye"}
```

Basic types: Choices (aka Discriminated Union)

```
type MyChoices = Choice1 | Choice2
let myChoice1 = Choice1
let myChoice2 = Choice2

type MyChoiceWithData =
    | Choice0WithNoData
    | Choice1WithIntData of int
    | Choice2WithStringData of string

// To create a choice, use the case pattern as a constructor
let myChoice0WithNonData = Choice0WithNoData
let myChoice1WithData = Choice1WithIntData 42
let myChoice2WithData = Choice2WithStringData "hello"
```

```
// Pattern matching for choices. To extract one of the choices,
// use the case pattern as a "deconstructor"
match myChoice1WithData with
    | Choice0WithNoData  ->
        printfn "no extra data"
    | Choice1WithIntData anInt ->
        printfn "an int %i" anInt
    | Choice2WithStringData aString ->
        printfn "a string %s" aString
```

Basic types: Lists

```
let myList = [1;2;3]      // square brackets
let myList2 = 0 :: myList // prepend with "::"

// NOTE: needs "rec" keyword for recursion
let rec loopThroughList aList =

    match aList with
        | [] ->           // match empty list
            printfn "List is empty. Stopping."

        | first::rest ->  // match first element and rest of list
            printfn "processing element %i" first
            loopThroughList rest

loopThroughList myList2
```

Helpful methods in the "List" module

```
myList |> List.rev

// "map" loop with one parameter lambda that returns a new value
myList |> List.map (fun x -> x + 1)

// e.g. collect uppercase versions
["Alice"; "Bob"; "Carol"] |> List.map (fun s -> s.ToUpper())

// "iter" loop with one parameter lambda that returns unit
myList |> List.iter (fun x -> printfn "x=%i" x)

// e.g. given a print fun with ONE parameter that returns unit
let printHello = printfn "Hello %s"       // string -> unit

// then you can use List.iter like this:
["Alice"; "Bob"; "Carol"] |> List.iter printHello
```