F# Basic Syntax

- Curly braces are NOT used to delimit blocks of code. Instead, indentation is used (like Python).
- Most common mistake is that commas are not used in the usual way:
  - To separate parameters, whitespace is used
  - In list literals and records, semicolons are used.

Declaring Types

Built-in types include:

```
string, int, float, decimal, bool, System.DateTime
```

Use the "type" keyword to define custom types. Names should be PascalCase.

```
type MyThing = …
```

Type annotations

A single colon ":" is used for type annotations.

```
let x = "abc"              // prints "val x : string"
let add1 (x:int) = x + 1 // typed parameter in a function
```

Defining tuples (pairs, triples, etc)

```
int * string         // A Pair. "*" indicates tuple types
int * string * bool  // A Triple.
```

Type aliases

```
type Name = string
type Triple = int * string * bool
```

Record types

{ } are used for defining record types

```
type MyRecord = {A:int; B:string} // semicolon if on same line
type MyRecord = {
  MyInt:int        // No semicolon needed
  MyString:string  // Field names are PascalCase
  }
```

Choice types (aka Discriminated Unions, Sum Types)

| is used for defining choice types

```
type Color = Red | Blue | Green  // enum style on one line
type MyChoice =                 // more complex, with 3 choices
  | Choice0WithNoData          // no associated data
  | Choice1WithIntData of int // use "of" to associate data
  | Choice2WithStringData of string
```

Generic types

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

Built-in types using generics are

```
'a option // also can be written Option<'a>
'a list   // fixed size list
'a seq    // lazy generator (IEnumerable)
```

Function types

Function types have arrows between each parameter and return type

```
int -> string -> bool  // pass int & string, return a bool
```

A function signature is documented this way

```
String.length   // signature is string -> int
List.contains   // signature is 'a -> 'a list -> bool
```

You can define named aliases for function types

```
type IntToString = int -> string
type MakePair<'a> = 'a -> 'a * 'a
```

Declaring Values

"`let`" is used instead of "`var`" or "`val`"

```
let x = 1
let x : int = 1 // with type annotation
```

Value names should be `camelCase`

Printing values

"`printfn`" writes a string (C# equivalent: `Console.WriteLine`)

```
printfn "%i %s %f %b" 1 "abc" 3.14 true // int,string,float,bool
printfn "%A" [1;2;3] // %A for lists and custom types
```

"`sprintf`" constructs a string (C# equivalent: `String.Format`)

```
let str = sprintf "%i %s %f %b" 1 "abc" 3.14 true
```

Working with Functions

"`let`" is used for declaring both values *and* functions

```
let printName myName =
   printfn "my name is %s" myName
```

The last expression in a function is the return value

```
let add x y =
    x + y // no return needed
```

When calling a function, no parentheses are needed

```
printName "Scott"
```

Also, whitespace is used to separate parameters, not commas. Watch out for using tuples by mistake when calling a function.

```
let result = add 1 2  // CORRECT. Spaces not commas
let result = add(1,2) // WRONG!
```

(You *can* pass tuples into a function, just don't do it by accident!)

Longer functions are indented. You can also define local values and functions inside a function.

```
let bigFunction x y =
    let z = x + y      // define new local value
    let add1 i = i + 1 // define new local function
    let w = add1 x     // call new local function
    z + w              // final return value
```

Type annotations for functions

Often not needed but can be helpful to fix compilation errors.

```
let doSomething (x:int) (y:string) :bool =  ...
//               ^parens ^for parameters
//                                 ^no parens for return type
```

Anonymous functions (lambdas)

Lambdas use the "`fun`" keyword

```
let myList = [1;2;3]                 // create a list
List.map (fun i -> i + 1) myList     // transform each element
List.filter (fun i -> i >2 ) myList  // filter the list
```

To define a function of a specific function type, use a lambda

```
let intToString : IntToString =  // specify type
   fun i -> i.ToString()         // implementation
```

Piping

"|>" is the pipe symbol. Piping passes the left side to the LAST parameter

```
"Scott" |> printName
2 |> add 1
```

Piping is commonly used to chain a series of actions

```
add 1 2
|> add 3                         // 2nd param comes from pipe
|> printfn "1 + 2 + 3 = %i"      // %i param comes from pipe
```

```
[1;2;3]
|> List.map (fun i -> i + 1)     // list param comes from pipe
|> List.filter (fun i -> i > 2)  // list param comes from pipe
```

Generic functions

Functions are inferred as generic automatically

```
let makePair x = (x,x)           // val makePair: x:'a -> 'a * 'a
```

Pattern matching

Used instead of a switch statement

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

Each case handler looks a bit like a lambda function

The unit type

The "unit" type and value represents no input or output (like void, sort of). The type is "`unit`" and the value is written "`()`"

```
let sayHello() = printfn "hello"
// signature is unit -> unit
```

```
let sayHello str = printfn "hello %s" str
// signature is string -> unit
```

Working with data types

Using Tuples

Commas are used to construct tuples

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

You can deconstruct in the same way

```
let x,y = myPair      // x=1, y=2
let x,y,z = myTriple  // x=1, y=2 etc
```

Using Records

Constructing records is similar to defining them

```
type MyRecordType = {A:int; B:string} // define
let myRecord = {A=1; B="hello"}       // construct
```

Records are immutable. If you want to "modify" them you have to copy all the fields and then update some of them using "with"

```
let myRecord2 = {myRecordValue with B="goodbye"}
```

To access data in a record, use standard dot-notation

```
let a = myRecord.A
```

Using Choices

```
type MyChoice =                 // define with three cases
  | Choice0WithNoData           // no associated data
  | Choice1WithIntData of int // use "of" to associate data
  | Choice2WithStringData of string
```

To construct a choice, use a specific case name as a constructor

```
let myChoice0 = Choice0WithNoData
let myChoice1 = Choice1WithIntData 42
let myChoice2= Choice2WithStringData "hello"
```

Pattern matching for choices

To extract one of the choices, use the case pattern as a "deconstructor"

```
match myChoice2 with
| Choice0WithNoData -> printfn "no extra data"
| Choice1WithIntData anInt -> printfn "an int %i" anInt
| Choice2WithStringData aString -> printfn "a string %s" aString
```

Each case handler looks a bit like a lambda function

Things to watch out for when you are used to other languages!

- "=" is used instead of "=="
- "<>" is used instead of "!="
- "not" is used instead of "!"
- No commas in function parameters (use spaces)
- No commas in lists or records (use semicolons)

Organizing code with Modules

Modules are used to group code (types and functions) together

```
module MyModule =
    type MyRecord = {A:int}
    let addTwo x = x + 2
```

A module qualifier can be added to a type or function

```
let myRecord : MyModule.MyRecord = { A = 123}
MyModule.addTwo 40
```

A module can also be “opened” (same as "using" in C#)

```
open MyModule
addTwo 40    // MyRecord & addTwo are now in scope directly
```

Working with Lists

Square brackets "[ ]" are for list literals. Double colon "::" is the list prepend operator

```
let bc = ["b"; "c"]  // note semicolon!
let abc = "a" :: bc
```

Pattern matching for lists

You can pattern match lists using :: and []. Here’s an example

```
let rec loopThroughList aList = // "rec" keyword for recursion
  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  // repeat with smaller list
```

Useful methods in the "List" module

For all the available functions search the internet for "Choosing between F# collection functions” and “F# List module”

`[1..10]` creates a list of numbers

`for..in..do` iterates over the elements with a body that returns unit

```
for x in myList do
  printfn "x=%i" x
```

`List.map` returns a new value for each element (LINQ equivalent: Select)

```
 myList |> List.map (fun x -> x + 1)
```

`List.filter` returns a filtered list (LINQ: Where)

```
 myList |> List.filter (fun x -> x > 42)
```

`List.choose` filters and maps in one step

```
 myList |> List.choose(fun x -> if x > 42 then Some(x+1) else None)
```

`List.collect` collapses lists (LINQ: SelectMany)

```
 myList |> List.collect (fun x -> [x; x+1; x+2] )
```

`List.exists` and `List.contains` check for membership

```
 myList |> List.exists (fun x -> x > 42)   // any items > 42?
 myList |> List.contains 43                // a specific item?
```

Working with Options

```
let x = Some 42  // assign the Some case
let y = None     // assign the None case
```

To deconstruct an option using pattern matching

```
let test anOption =
   match anOption with
   | Some x -> printfn "Option is Some %A" x
   | None -> printfn "Option is None"
```

To work with options without pattern matching

```
anOption |> Option.map (fun x -> x + 1)
anOption |> Option.defaultValue 42
anOption |> Option.bind (fun x -> if x < 0 then None else Some x)
```