Fsharp

F# is a general purpose functional/OO programming language. It's free and open source, and runs on Linux, Mac, Windows and more.

It has a powerful type system that traps many errors at compile time, but it uses type inference so that it reads more like a dynamic language.

The syntax of F# is different from C-style languages:

- 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.

If you want to try out the code below, you can go to tryfsharp.org and paste it into an interactive REPL.

```
// single line comments use a double slash
(* multi line comments use (* . . . *) pair

-end of multi line comment- *)

// ================================================
// Basic Syntax
// ================================================

// ------ "Variables" (but not really) ------
// The "let" keyword defines an (immutable) value
let myInt = 5
let myFloat = 3.14
let myString = "hello"           // note that no types needed

// ------ Lists ------
let twoToFive = [2; 3; 4; 5]        // Square brackets create a list with
                                 // semicolon delimiters.
let oneToFive = 1 :: twoToFive   // :: creates list with new 1st element
// The result is [1; 2; 3; 4; 5]
let zeroToFive = [0; 1] @ twoToFive   // @ concats two lists

// IMPORTANT: commas are never used as delimiters, only semicolons!

// ------ Functions ------
// The "let" keyword also defines a named function.
let square x = x * x          // Note that no parens are used.
square 3                      // Now run the function. Again, no parens.

let add x y = x + y           // don't use add (x,y)! It means something
                              // completely different.
add 2 3                       // Now run the function.

// to define a multiline function, just use indents. No semicolons needed.
let evens list =
   let isEven x = x % 2 = 0     // Define "isEven" as a sub function
   List.filter isEven list    // List.filter is a library function
                              // with two parameters: a boolean function
                              // and a list to work on

evens oneToFive               // Now run the function
```

```fsharp
// You can use parens to clarify precedence. In this example,
// do "map" first, with two args, then do "sum" on the result.
// Without the parens, "List.map" would be passed as an arg to List.sum
let sumOfSquaresTo100 =
   List.sum ( List.map square [1..100] )

// You can pipe the output of one operation to the next using "|>"
// Piping data around is very common in F#, similar to UNIX pipes.

// Here is the same sumOfSquares function written using pipes
let sumOfSquaresTo100piped =
   [1..100] |> List.map square |> List.sum  // "square" was defined earlier

// you can define lambdas (anonymous functions) using the "fun" keyword
let sumOfSquaresTo100withFun =
   [1..100] |> List.map (fun x -> x * x) |> List.sum

// In F# there is no "return" keyword. A function always
// returns the value of the last expression used.

// ------ Pattern Matching ------
// Match..with.. is a supercharged case/switch statement.
let simplePatternMatch =
   let x = "a"
   match x with
    | "a" -> printfn "x is a"
    | "b" -> printfn "x is b"
    | _ -> printfn "x is something else"   // underscore matches anything

// F# doesn't allow nulls by default -- you must use an Option type
// and then pattern match.
// Some(..) and None are roughly analogous to Nullable wrappers
let validValue = Some(99)
let invalidValue = None

// In this example, match..with matches the "Some" and the "None",
// and also unpacks the value in the "Some" at the same time.
let optionPatternMatch input =
   match input with
    | Some i -> printfn "input is an int=%d" i
    | None -> printfn "input is missing"

optionPatternMatch validValue
optionPatternMatch invalidValue

// ------ Printing ------
// The printf/printfn functions are similar to the
// Console.Write/WriteLine functions in C#.
printfn "Printing an int %i, a float %f, a bool %b" 1 2.0 true
printfn "A string %s, and something generic %A" "hello" [1; 2; 3; 4]

// There are also sprintf/sprintfn functions for formatting data
// into a string, similar to String.Format in C#.
```

```
// ================================================
// More on functions
// ================================================

// F# is a true functional language -- functions are first
// class entities and can be combined easily to make powerful
// constructs

// Modules are used to group functions together
// Indentation is needed for each nested module.
module FunctionExamples =

    // define a simple adding function
    let add x y = x + y

    // basic usage of a function
    let a = add 1 2
    printfn "1 + 2 = %i" a

    // partial application to "bake in" parameters
    let add42 = add 42
    let b = add42 1
    printfn "42 + 1 = %i" b

    // composition to combine functions
    let add1 = add 1
    let add2 = add 2
    let add3 = add1 >> add2
    let c = add3 7
    printfn "3 + 7 = %i" c

    // higher order functions
    [1..10] |> List.map add3 |> printfn "new list is %A"

    // lists of functions, and more
    let add6 = [add1; add2; add3] |> List.reduce (>>)
    let d = add6 7
    printfn "1 + 2 + 3 + 7 = %i" d

// ================================================
// Lists and collection
// ================================================

// There are three types of ordered collection:
// * Lists are most basic immutable collection.
// * Arrays are mutable and more efficient when needed.
// * Sequences are lazy and infinite (e.g. an enumerator).
//
// Other collections include immutable maps and sets
// plus all the standard .NET collections

module ListExamples =
```

```
    // lists use square brackets
    let list1 = ["a"; "b"]
    let list2 = "c" :: list1    // :: is prepending
    let list3 = list1 @ list2   // @ is concat

    // list comprehensions (aka generators)
    let squares = [for i in 1..10 do yield i * i]

    // prime number generator
    let rec sieve = function
        | (p::xs) -> p :: sieve [ for x in xs do if x % p > 0 then yield x ]
        | []      -> []
    let primes = sieve [2..50]
    printfn "%A" primes

    // pattern matching for lists
    let listMatcher aList =
        match aList with
        | [] -> printfn "the list is empty"
        | [first] -> printfn "the list has one element %A " first
        | [first; second] -> printfn "list is %A and %A" first second
        | _ -> printfn "the list has more than two elements"

    listMatcher [1; 2; 3; 4]
    listMatcher [1; 2]
    listMatcher [1]
    listMatcher []

    // recursion using lists
    let rec sum aList =
        match aList with
        | [] -> 0
        | x::xs -> x + sum xs
    sum [1..10]

    // -----------------------------------------
    // Standard library functions
    // -----------------------------------------

    // map
    let add3 x = x + 3
    [1..10] |> List.map add3

    // filter
    let even x = x % 2 = 0
    [1..10] |> List.filter even

    // many more -- see documentation

module ArrayExamples =

    // arrays use square brackets with bar
    let array1 = [| "a"; "b" |]
    let first = array1.[0]        // indexed access using dot
```

```
    // pattern matching for arrays is same as for lists
    let arrayMatcher aList =
        match aList with
        | [| |] -> printfn "the array is empty"
        | [| first |] -> printfn "the array has one element %A " first
        | [| first; second |] -> printfn "array is %A and %A" first second
        | _ -> printfn "the array has more than two elements"

    arrayMatcher [| 1; 2; 3; 4 |]

    // Standard library functions just as for List

    [| 1..10 |]
    |> Array.map (fun i -> i + 3)
    |> Array.filter (fun i -> i % 2 = 0)
    |> Array.iter (printfn "value is %i. ")


module SequenceExamples =

    // sequences use curly braces
    let seq1 = seq { yield "a"; yield "b" }

    // sequences can use yield and
    // can contain subsequences
    let strange = seq {
        // "yield" adds one element
        yield 1; yield 2;

        // "yield!" adds a whole subsequence
        yield! [5..10]
        yield! seq {
            for i in 1..10 do
              if i % 2 = 0 then yield i }}
    // test
    strange |> Seq.toList


    // Sequences can be created using "unfold"
    // Here's the fibonacci series
    let fib = Seq.unfold (fun (fst,snd) ->
        Some(fst + snd, (snd, fst + snd))) (0,1)

    // test
    let fib10 = fib |> Seq.take 10 |> Seq.toList
    printf "first 10 fibs are %A" fib10


// ================================================
// Data Types
// ================================================

module DataTypeExamples =
```

```
// All data is immutable by default

// Tuples are quick 'n easy anonymous types
// -- Use a comma to create a tuple
let twoTuple = 1, 2
let threeTuple = "a", 2, true

// Pattern match to unpack
let x, y = twoTuple  // sets x = 1, y = 2

// ------------------------------------
// Record types have named fields
// ------------------------------------

// Use "type" with curly braces to define a record type
type Person = {First:string; Last:string}

// Use "let" with curly braces to create a record
let person1 = {First="John"; Last="Doe"}

// Pattern match to unpack
let {First = first} = person1    // sets first="John"

// ------------------------------------
// Union types (aka variants) have a set of choices
// Only case can be valid at a time.
// ------------------------------------

// Use "type" with bar/pipe to define a union type
type Temp =
    | DegreesC of float
    | DegreesF of float

// Use one of the cases to create one
let temp1 = DegreesF 98.6
let temp2 = DegreesC 37.0

// Pattern match on all cases to unpack
let printTemp = function
   | DegreesC t -> printfn "%f degC" t
   | DegreesF t -> printfn "%f degF" t

printTemp temp1
printTemp temp2

// ------------------------------------
// Recursive types
// ------------------------------------

// Types can be combined recursively in complex ways
// without having to create subclasses
type Employee =
  | Worker of Person
```

```fsharp
  | Manager of Employee list

let jdoe = {First="John"; Last="Doe"}
let worker = Worker jdoe

// ------------------------------------
// Modeling with types
// ------------------------------------

// Union types are great for modeling state without using flags
type EmailAddress =
    | ValidEmailAddress of string
    | InvalidEmailAddress of string

let trySendEmail email =
    match email with // use pattern matching
    | ValidEmailAddress address -> ()   // send
    | InvalidEmailAddress address -> () // dont send

// The combination of union types and record types together
// provide a great foundation for domain driven design.
// You can create hundreds of little types that accurately
// reflect the domain.

type CartItem = { ProductCode: string; Qty: int }
type Payment = Payment of float
type ActiveCartData = { UnpaidItems: CartItem list }
type PaidCartData = { PaidItems: CartItem list; Payment: Payment}

type ShoppingCart =
    | EmptyCart  // no data
    | ActiveCart of ActiveCartData
    | PaidCart of PaidCartData

// ------------------------------------
// Built in behavior for types
// ------------------------------------

// Core types have useful "out-of-the-box" behavior, no coding needed.
// * Immutability
// * Pretty printing when debugging
// * Equality and comparison
// * Serialization

// Pretty printing using %A
printfn "twoTuple=%A,\nPerson=%A,\nTemp=%A,\nEmployee=%A"
         twoTuple person1 temp1 worker

// Equality and comparison built in.
// Here's an example with cards.
type Suit = Club | Diamond | Spade | Heart
type Rank = Two | Three | Four | Five | Six | Seven | Eight
            | Nine | Ten | Jack | Queen | King | Ace
```

```
    let hand = [ Club, Ace; Heart, Three; Heart, Ace;
                 Spade, Jack; Diamond, Two; Diamond, Ace ]

    // sorting
    List.sort hand |> printfn "sorted hand is (low to high) %A"
    List.max hand |> printfn "high card is %A"
    List.min hand |> printfn "low card is %A"


// ================================================
// Active patterns
// ================================================

module ActivePatternExamples =

    // F# has a special type of pattern matching called "active patterns"
    // where the pattern can be parsed or detected dynamically.

    // "banana clips" are the syntax for active patterns

    // for example, define an "active" pattern to match character types...
    let (|Digit|Letter|Whitespace|Other|) ch =
       if System.Char.IsDigit(ch) then Digit
       else if System.Char.IsLetter(ch) then Letter
       else if System.Char.IsWhiteSpace(ch) then Whitespace
       else Other

    // ... and then use it to make parsing logic much clearer
    let printChar ch =
      match ch with
      | Digit -> printfn "%c is a Digit" ch
      | Letter -> printfn "%c is a Letter" ch
      | Whitespace -> printfn "%c is a Whitespace" ch
      | _ -> printfn "%c is something else" ch

    // print a list
    ['a'; 'b'; '1'; ' '; '-'; 'c'] |> List.iter printChar

    // -----------------------------------
    // FizzBuzz using active patterns
    // -----------------------------------

    // You can create partial matching patterns as well
    // Just use underscore in the defintion, and return Some if matched.
    let (|MultOf3|_|) i = if i % 3 = 0 then Some MultOf3 else None
    let (|MultOf5|_|) i = if i % 5 = 0 then Some MultOf5 else None

    // the main function
    let fizzBuzz i =
      match i with
      | MultOf3 & MultOf5 -> printf "FizzBuzz, "
      | MultOf3 -> printf "Fizz, "
      | MultOf5 -> printf "Buzz, "
      | _ -> printf "%i, " i
```

```
    // test
    [1..20] |> List.iter fizzBuzz

// ================================================
// Conciseness
// ================================================

module AlgorithmExamples =

    // F# has a high signal/noise ratio, so code reads
    // almost like the actual algorithm

    // ------ Example: define sumOfSquares function ------
    let sumOfSquares n =
       [1..n]              // 1) take all the numbers from 1 to n
       |> List.map square  // 2) square each one
       |> List.sum         // 3) sum the results

    // test
    sumOfSquares 100 |> printfn "Sum of squares = %A"

    // ------ Example: define a sort function ------
    let rec sort list =
       match list with
       // If the list is empty
       | [] ->
            []                            // return an empty list
       // If the list is not empty
       | firstElem::otherElements ->      // take the first element
            let smallerElements =         // extract the smaller elements
                otherElements             // from the remaining ones
                |> List.filter (fun e -> e < firstElem)
                |> sort                   // and sort them
            let largerElements =          // extract the larger ones
                otherElements             // from the remaining ones
                |> List.filter (fun e -> e >= firstElem)
                |> sort                   // and sort them
            // Combine the 3 parts into a new list and return it
            List.concat [smallerElements; [firstElem]; largerElements]

    // test
    sort [1; 5; 23; 18; 9; 1; 3] |> printfn "Sorted = %A"

// ================================================
// Asynchronous Code
// ================================================

module AsyncExample =

    // F# has built-in features to help with async code
    // without encountering the "pyramid of doom"
    //
    // The following example downloads a set of web pages in parallel.
```

```
    open System.Net
    open System
    open System.IO
    open Microsoft.FSharp.Control.CommonExtensions

    // Fetch the contents of a URL asynchronously
    let fetchUrlAsync url =
        async {   // "async" keyword and curly braces
                  // creates an "async" object
            let req = WebRequest.Create(Uri(url))
            use! resp = req.AsyncGetResponse()
                // use! is async assignment
            use stream = resp.GetResponseStream()
                // "use" triggers automatic close()
                // on resource at end of scope
            use reader = new IO.StreamReader(stream)
            let html = reader.ReadToEnd()
            printfn "finished downloading %s" url
            }

    // a list of sites to fetch
    let sites = ["http://www.bing.com";
                 "http://www.google.com";
                 "http://www.microsoft.com";
                 "http://www.amazon.com";
                 "http://www.yahoo.com"]

    // do it
    sites
    |> List.map fetchUrlAsync  // make a list of async tasks
    |> Async.Parallel          // set up the tasks to run in parallel
    |> Async.RunSynchronously  // start them off

// ================================================
// .NET compatibility
// ================================================

module NetCompatibilityExamples =

    // F# can do almost everything C# can do, and it integrates
    // seamlessly with .NET or Mono libraries.

    // ------- work with existing library functions  -------

    let (i1success, i1) = System.Int32.TryParse("123");
    if i1success then printfn "parsed as %i" i1 else printfn "parse failed"

    // ------- Implement interfaces on the fly! -------

    // create a new object that implements IDisposable
    let makeResource name =
       { new System.IDisposable
         with member this.Dispose() = printfn "%s disposed" name }
```

```
let useAndDisposeResources =
    use r1 = makeResource "first resource"
    printfn "using first resource"
    for i in [1..3] do
        let resourceName = sprintf "\tinner resource %d" i
        use temp = makeResource resourceName
        printfn "\tdo something with %s" resourceName
    use r2 = makeResource "second resource"
    printfn "using second resource"
    printfn "done."

// ------- Object oriented code -------

// F# is also a fully fledged OO language.
// It supports classes, inheritance, virtual methods, etc.

// interface with generic type
type IEnumerator<'a> =
    abstract member Current : 'a
    abstract MoveNext : unit -> bool

// abstract base class with virtual methods
[<AbstractClass>]
type Shape() =
    // readonly properties
    abstract member Width : int with get
    abstract member Height : int with get
    // non-virtual method
    member this.BoundingArea = this.Height * this.Width
    // virtual method with base implementation
    abstract member Print : unit -> unit
    default this.Print () = printfn "I'm a shape"

// concrete class that inherits from base class and overrides
type Rectangle(x:int, y:int) =
    inherit Shape()
    override this.Width = x
    override this.Height = y
    override this.Print ()  = printfn "I'm a Rectangle"

// test
let r = Rectangle(2, 3)
printfn "The width is %i" r.Width
printfn "The area is %i" r.BoundingArea
r.Print()

// ------- extension methods  -------

// Just as in C#, F# can extend existing classes with extension methods.
type System.String with
   member this.StartsWithA = this.StartsWith "A"

// test
```

```
let s = "Alice"
printfn "'%s' starts with an 'A' = %A" s s.StartsWithA

// ------- events  -------

type MyButton() =
    let clickEvent = new Event<_>()

    [<CLIEvent>]
    member this.OnClick = clickEvent.Publish

    member this.TestEvent(arg) =
        clickEvent.Trigger(this, arg)

// test
let myButton = new MyButton()
myButton.OnClick.Add(fun (sender, arg) ->
        printfn "Click event with arg=%O" arg)

myButton.TestEvent("Hello World!")
```

More Information

For more demonstrations of F#, go to the Try F# site, or my why use F# series.

Read more about F# at fsharp.org.