

---

.Net for Quants

---

# F# for Quants



# Agenda : functional programming and F#

## Introduction

Functional programming concepts with F#

## Concurrency

Asynchronous and parallel programming in F#

## Units

F# units of measure

## Data

Data science with FsLab and Type Providers

# Agenda : using F# in a C# development environment

## Tests

Unit tests with FsUnit

## DSL

FAKE - F# Make - A DSL for build tasks

## Resources

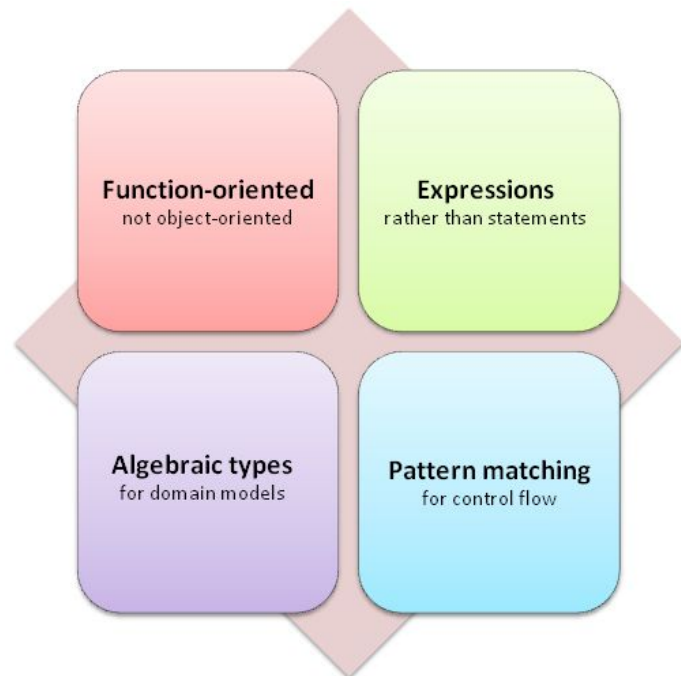
Resources to learn more about F#

# Functional programming concepts

Functional (declarative)  $\neq$  Imperative

4 key concepts :

- ★ **Function-oriented:** functions are first-class members
  - Programs are built with function composition.
- ★ **Expressions:** functional languages are sometimes called “expression-oriented” languages.
- ★ **Algebraic types:** new compound types are built by combining existing types in two different ways
  - Combination of values, called “product” types.
  - Disjoint union representing a choice between a set of types, called “sum” types.
- ★ **Pattern matching:** a powerful feature to bind values with expressions, and for branching (flow control).

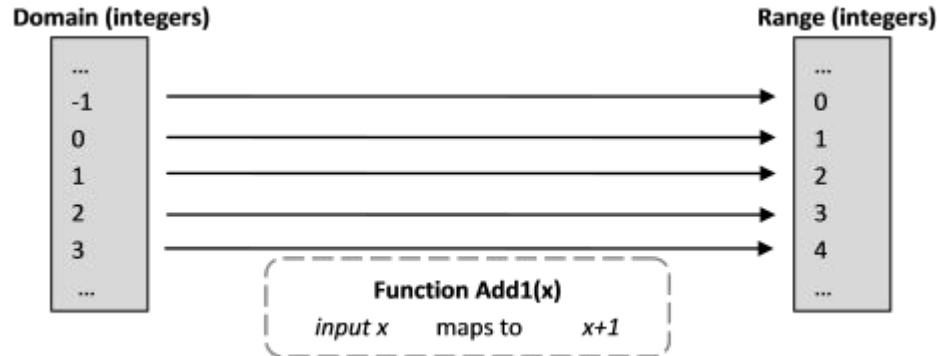


<http://fsharpforfunandprofit.com/assets/img/four-concepts2.png>

# Functional programming concepts

## Functions as first-class members

- ★ Functions are at the heart of functional programming.
- ★ Functions are values, and can be passed as argument to other functions (“higher-order” functions).
- ★ Functions only take one argument (currying).
- ★ Mathematical functions map an input domain to an output range (**no side effect**)!
  - In F#, the `let` keyword binds a name to a value.



# Functional programming concepts

## ★ Defining an anonymous function

- Use the `fun` keyword. They are equivalent to C# lambda expressions.

```
let add x y = x + y // Conventional function definition
```

```
let add = fun x y -> x + y // The equivalent using a lambda
```

```
let add = fun x -> (fun y -> x + y) // This definition makes clear an intermediate function is returned
```

- Lambdas are often used as short expressions:

```
[1..10] |> List.map (fun i -> i + 1)
```

```
[1..10] |> List.map ((+) 1)
```

## ★ Modules

- Functions can be grouped into modules.

```
module Arithmetic =  
  let inline add x y = x + y  
  let inline mult x y = x * y  
  
module MathCalculus =  
  let add1 = Arithmetic.add 1 // partial application
```

# Functional programming concepts

## ★ Combinators

- F# makes heavy use of combinators: they ease function composition and code readability.

```
let (|>) x f = f x           // forward pipe
let (<|) f x = f x           // reverse pipe
let (>>) f g x = g (f x)     // forward composition
let (<<) g f x = g (f x)     // reverse composition
```

## ★ Function composition, with the “forward composition” combinator :

```
let add5thenTimes3 = (+) 5 >> (*) 3 // Add 5, then multiply by 3
```

## ★ Operators

- You can define your own operators.

```
let (@@) path1 path2 = Path.Combine(path1, path2) // Combine 2 paths the functional way
let fullPath = @"D:\trainings\FSharp" @@ "FSharp4Quants.fsx" |> Path.GetFullPath
```

# Functional programming concepts

## ★ Recursive functions

```
open Checked           // Redefines common operators for arithmetic overflow checks
open System.Numerics
```

```
// Definition of a factorial
```

```
let rec factorial i =           // val factorial : byte -> BigInteger
    match i with
    | 0uy | 1uy -> 1I
    | _         -> (bigint (i |> int)) * factorial (i-1uy)
```

```
// Definition of a Fibonacci suite
```

```
let rec fib i =
    match i with
    | 0          -> 0
    | 1          -> 1
    | n when n > 1 -> fib(n-1) + fib(n-2) // use of 'when' guards
    | n when n < 0 -> fib(n+2) - fib(n+1)
```



# Functional programming concepts

## Functions as Interfaces: "program to an interface, not an implementation"

- ★ In functional programming, all functions are “interfaces” (in the sense that their signature is the interface)!

Consider for ex. the decorator design pattern:

### In C#

```
interface ICalculator
{
    int Calculate(int input);
}
```

```
class Adding1Calculator: ICalculator
{
    public int Calculate(int input) { return input + 1; }
}
```

```
class LoggingCalculator: ICalculator
{
    ICalculator innerCalculator;

    LoggingCalculator(ICalculator innerCalculator)
    {
        this.innerCalculator = innerCalculator;
    }

    public int Calculate(int input)
    {
        Console.WriteLine("input is {0}", input);
        var result = this.innerCalculator.Calculate(input);
        Console.WriteLine("result is {0}", result);
        return result;
    }
}
```

# Functional programming concepts

In F#:

- ★ You can do the same thing without having to define the interface first.
- ★ Any function can be transparently swapped for any other function as long as the signatures are the same.

```
let add1 input = input + 1 // val add1 : int -> int
let genericLogger anyFunc input = // val genericLogger : ('a -> 'b) -> 'a -> 'b
    printfn "input is %A" input
    let result = anyFunc input
    printfn "result is %A" result
    result
let genericTimer anyFunc input = // val genericTimer : ('a -> 'b) -> 'a -> 'b
    let stopwatch = System.Diagnostics.Stopwatch()
    stopwatch.Start()
    let result = anyFunc input
    printfn "elapsed ms is %A." stopwatch.ElapsedMilliseconds
    result
let add1WithLogging = genericLogger add1 // val add1WithLogging : (int -> int)
let add1WithTimer = genericTimer add1WithLogging // val add1WithTimer : (int -> int)
```

# Functional programming concepts

## Expressions rather than statements

- ★ In functional languages, there are no statements, only expressions
  - There are no variables, only values.
  - Every chunk of code always returns a value.
  - “No” `return` keyword to return a value.

`// add1 is a function value. The name “add1” itself is a binding to “the function that adds one to its input”.`

`let add1 x = x + 1 // “x” is not a variable! It is a onetime association of the name “x” with the value.`

The signature of a function value is: `val functionName : domain -> range`

# Functional programming concepts

## “Values” versus “Objects”

- ★ In a functional programming language like F#, most things are called "values":
  - A value, as we have seen above, is just a member of a domain (mathematical abstraction).
    - The domain of ints,
    - The domain of strings,
    - The domain of functions that map ints to strings, ...
  - In principle, values are immutable.
  - They do not have any behavior attached them.
  
- ★ In an object-oriented language like C# or Python, most things are called "objects":
  - An object is an encapsulation of a data structure with its associated behavior (methods).
  - In general, objects are expected to have state (that is, be mutable).
  - All operations that change the internal state must be provided by the object itself (via "dot" notation).

# Functional programming concepts

## Algebraic types and Domain-Driven Design

The type system in F# is based on the concept of **algebraic types** (composable types):

- ★ First, a combination of values, each picked from a set of types. These are called "product" types.
  - Tuples aka product types.
- ★ Of, alternately, as a disjoint union representing a choice between a set of types. These are called "sum" types.
  - Unions aka sum types.

```
// Example: representing Stock Options
/// The option kind
type OptionKind = Put | Call

/// An option is either European or a combination of options
type OptionAEC =
    | American of OptionKind * float
    | European of OptionKind * float
    | Combine  of OptionAEC * OptionAEC
```

# Functional programming concepts

## “Pattern matching” and “Active Patterns”

- ★ Pattern matching is a very powerful feature of functional languages. It is used:
  - for binding values to expressions with `let`, and in function parameters.
  - for branching using the `match..with` syntax.

```
let first, second, _ = (1,2,3) // matching tuples directly. Underscore means ignore
let e1::e2::rest = [1..10]    // matching lists directly

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

# Functional programming concepts

## “Pattern matching” and “Active Patterns”

- ★ F# has a special type of pattern matching called “active patterns”, where the pattern can be parsed or detected dynamically.

```
/// Active pattern which discriminates between files and directories.  
let (|File|Directory|) (fileSysInfo : FileSystemInfo) =  
    match fileSysInfo with  
    | :? FileInfo      as file -> File(file)  
    | :? DirectoryInfo as dir  -> Directory(dir, dir.EnumerateFileSystemInfos())  
    | _ -> failwith "No file or directory given."
```

# Functional programming concepts

## Why F# ?



### Completeness

- ★ F# is part of the .Net ecosystem
  - You can reference any .Net assembly in your F# programs.
- ★ F# is a hybrid functional / OO language
  - Its is not a pure functional language (eg. Haskell).
  - Everything C# / Python can do can be done in F#.
- ★ Both a compiled language (.fs files) and a scripting language (.fsx files)

```
// impure code when needed
let mutable counter = 0

// create C# compatible classes and interfaces
type IEnumerator<'a> =
    abstract member Current : 'a
    abstract MoveNext : unit -> bool

// extension methods
type System.Int32 with
    member this.IsEven = (this % 2 = 0)

let i = 20
match i.IsEven with
| true -> printfn "%i' is even." i
| false -> printfn "%i' is not even." i

// UI code
open System.Windows.Forms
let form = new Form(Width = 400, Height = 300
    , Visible = true, Text = "Hello World")
form.TopMost <- true
form.Click.Add (fun args -> printfn "Clicked!")
form.Show()
```



# Functional programming concepts



## Why F# ?

- ★ Easy prototyping with its REPL (Read - Eval - Print - Loop)
  - Execute the F# Interactive Console “fsi.exe”, and start prototyping!
- ★ No cyclic dependencies in F#
  - All values must be defined before they can be used: build order is important.
- 🔓 Open source, cross-platform
  - F# source code and tools available on GitHub (since version 3.0): <https://github.com/fsharp/fsharp>
  - Targets platforms Windows, Linux, OS X, GPU...

# Asynchronous and parallel programming in F#

## ★ F# is both

- **A parallel language:** F# programs can have multiple active evaluations (e.g. .NET threads actively computing results).
- **A reactive language:** F# programs can have multiple pending reactions (e.g. callbacks and agents waiting to react to events and messages).
  - Reactive programming is a programming paradigm oriented around data flows and the propagation of change.

## Why functional parallel programming?

### ★ What is the purpose of synchronisation?

- To avoid conflicting updates of shared data.

### ★ Functional programming

- No updates to shared data: immutable by default -> **no lock!**
- Instead: Copying, partial sharing, intermediate data structures, message passing, agents, ...

# Asynchronous and parallel programming in F#

## Asynchronous workflows in F# with the `async` computation expression

These workflows are objects that encapsulate a background task, and provide a number of useful operations to manage them.

### ★ CPU-bound parallel programming

- Computing 40 Fibonacci numbers

```
let fibs = [ for i in 0..39 do yield fib(i) ]  
// Real: 00:00:06.385, CPU: 00:00:06.386
```

- Doing it in parallel

```
let fibs =  
    let tasks = [ for i in 0..39 do yield async { return fib(i) } ]  
    tasks |> Async.Parallel |> Async.RunSynchronously  
// Real: 00:00:02.662, CPU: 00:00:07.352 (2.4 x faster on a 4-core machine)
```

### Tips:

`Async.Parallel<'T>` Method

```
// signature  
static member Parallel :  
    seq<Async<'T>> -> Async<'T []>
```

Creates an asynchronous computation that executes all the given asynchronous computations, initially queueing each as work items and using a fork/join pattern.

# Asynchronous and parallel programming in F#

- ★ More concurrency: I/O-bound parallel programming
  - Not optimal (because `lengthSync` blocks your calling thread):

```
let lens =  
    let tasks = [ for url in urls do yield async { return lengthSync url } ]  
    tasks |> Async.Parallel |> Async.RunSynchronously
```

- Better: let I/O system deal with responses

```
let lengthAsync (url : string) =  
    async {  
        let wc = new WebClient()  
        let! html = wc.AsyncDownloadString(Uri(url))  
        return html.Length }  
  
let lens =  
    [ for url in urls -> lengthAsync url ] |> Async.Parallel |> Async.RunSynchronously
```

## Tips:

`Async.RunSynchronously<'T>`

Method

```
// signature  
static member RunSynchronously :  
    Async<'T> * ?int * ?  
    CancellationToken -> 'T
```

Runs the provided asynchronous computation and awaits its result.

# F# units of measure

## Units of measure

Define and convert units of measure (with static checking and type inference):

```
[<Measure>] type EUR
[<Measure>] type USD
[<Measure>] type GBP

let inline convert rate = // statically resolved type parameters
    fun price -> price * rate

let price_eur = 10.0<EUR>
let eur_usd = 1.06734<USD/EUR>
let price_usd = convert eur_usd price_eur

let eur_gbp = 0.707480198<GBP/EUR>

let gbp_usd = eur_usd / eur_gbp
// val gbp_usd : float<USD/GBP> = 1.508649999
```



# Data science with FsLab and Type Providers

**FsLab** : Data science and machine learning with F#.



FsLab is a collection of libraries for data-science.

It provides a rapid development environment that lets you write advanced analysis with a few lines of production-quality code.

Access data using **type providers**, explore it using an efficient **data-frame** and **time-series** library and through the **R language** integration and use rich **visualization** and **reporting**.



# Data science with FsLab and Type Providers

```
open RProvider
open RProvider.graphics
```

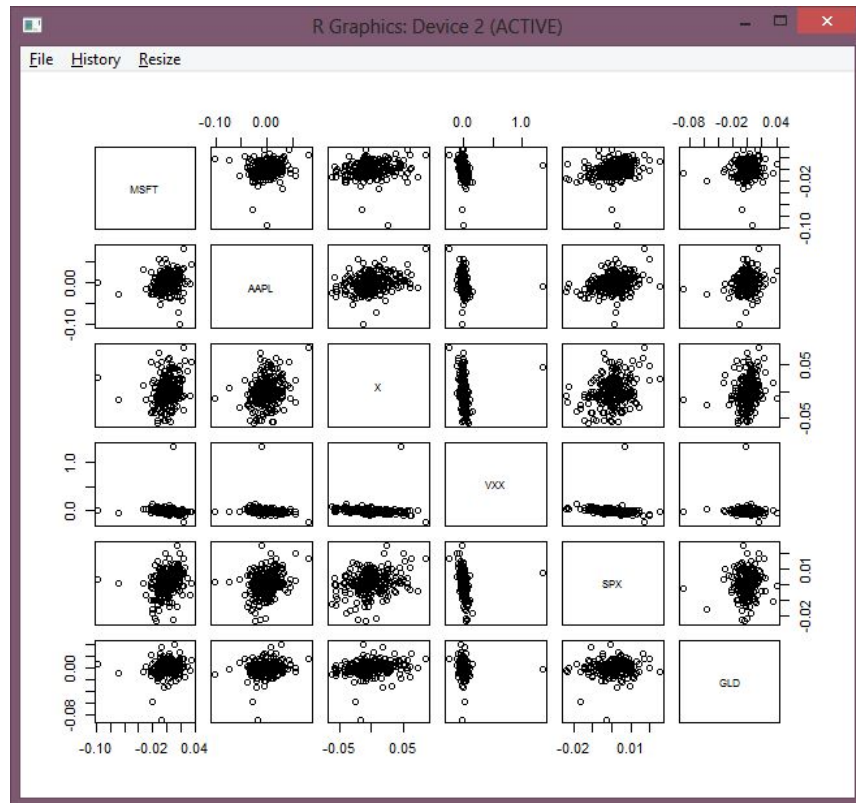
```
type Stocks = CsvProvider<"http://ichart.finance.yahoo.
com/table.csv?s=SPX">
```

```
/// Returns prices of a given stock for a specified number
/// of days (starting from the most recent)
```

```
let getStockPrices stock count =
    let url = "http://ichart.finance.yahoo.com/table.csv?s="
    [| for r in Stocks.Load(url + stock).Take(count).Rows ->
        float r.Open |] |> Array.rev
```

```
// Build a list of tickers and get diff of logs of prices
let tickers = [ "MSFT"; "AAPL"; "X"; "VXX"; "SPX"; "GLD" ]
let data =
    [ for t in tickers ->
        t, getStockPrices t 255 |> R.log |> R.diff ]
```

```
// Create an R data frame with the data and call 'R.pairs'
let df = R.data_frame(namedParams data)
R.pairs(df)
```



# Data science with FsLab and Type Providers

## ★ What are Type Providers

- Type providers are part of **F# 3.0** support for information-rich programming.
- An F# type provider is a component that provides types, properties, and methods for use in your program.

## ★ A focus on the F# R Type Provider

- It is a mechanism that enables smooth interoperability between F# and R.
- The Type Provider discovers R packages that are available in your R installation.
- It makes it possible to use all of R capabilities, from the F# interactive environment.
- It enables on-the-fly charting and data analysis using R packages, with the added benefit of
  - IntelliSense over R, and compile-time type-checking that the R functions you are using exist.

## ★ Examples of other Type Providers

- JSON, XML, CSV, EDMX
- SQL
- WSDL
- WMI...



# Unit testing with FsUnit

- ★ F# allows you to write sentences for test names: use ``double backticks``
- ★ There are powerful libraries to help you write good tests:
  - FsUnit (with NUnit or xUnit)
  - FsCheck and QuickCheck for property based testing.

```
open NUnit.Framework
```

```
open FsUnit
```

```
let inline add x y = x + y
```

```
[<Test>]
```

```
let ``When 2 is added to 2 expect 4``() =  
    add 2 2 |> should equal 4
```

```
[<Test>]
```

```
let ``When 2.0 is added to 2.0 expect 4.01``() =  
    add 2.0 2.0 |> should (equalWithin 0.1) 4.01
```

```
[<Test>]
```

```
let ``When ToLower(), expect lowercase letters``() =  
    "F#SHARP".ToLower() |> should startWith "fs"
```

# FAKE - F# Make - A DSL for build tasks



## ★ What is a DSL?

- It is a “Domain-Specific Language”.
- Sometimes called “mini-language” aimed at specific tasks.

FAKE is a cross-platform build automation tool written in F#, analogous to *make* and Ruby's *Rake*.

FAKE has built-in support for *git*, *NuGet*, unit tests and more, and makes it easy to develop complex scripts with dependencies.

FAKE to remove dependencies on a particular build server, e.g. you can run full builds without having *TeamCity* installed.

# FAKE - F# Make - A DSL for build tasks

```
#r "packages/FAKE/tools/FakeLib.dll" // include Fake lib
open Fake

let buildDir = "./build/" // Properties

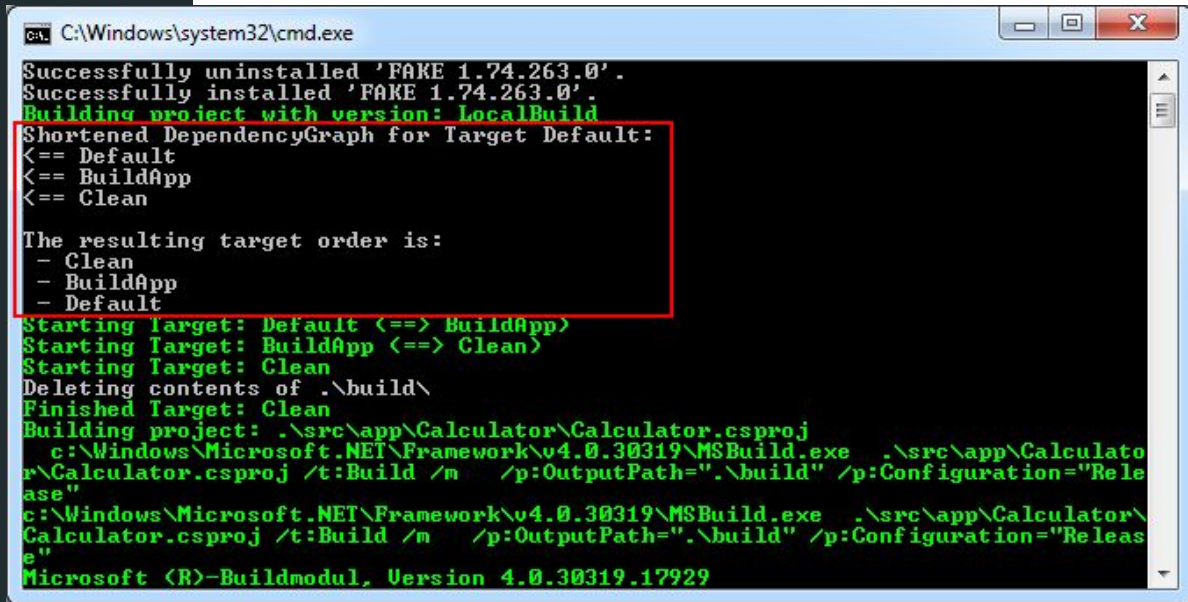
// Targets
Target "Clean" (fun _ ->
    CleanDir buildDir
)

Target "BuildApp" (fun _ ->
    !! "src/app/**/*.csproj"
    |> MSBuildRelease buildDir "Build"
    |> Log "AppBuild-Output: "
)

Target "Default" (fun _ ->
    trace "Hello World from FAKE"
)

// Dependencies
"Clean"
==> "BuildApp"
==> "Default"

// start build
RunTargetOrDefault "Default"
```



```
C:\Windows\system32\cmd.exe
Successfully uninstalled 'FAKE 1.74.263.0'.
Successfully installed 'FAKE 1.74.263.0'.
Building project with version: LocalBuild
Shortened DependencyGraph for Target Default:
<== Default
<== BuildApp
<== Clean

The resulting target order is:
- Clean
- BuildApp
- Default

Starting Target: Default <==> BuildApp>
Starting Target: BuildApp <==> Clean>
Starting Target: Clean
Deleting contents of .\build\
Finished Target: Clean
Building project: .\src\app\Calculator\Calculator.csproj
c:\Windows\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe .\src\app\Calculator\Calculator.csproj /t:Build /m /p:OutputPath=".build" /p:Configuration="Release"
c:\Windows\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe .\src\app\Calculator\Calculator.csproj /t:Build /m /p:OutputPath=".build" /p:Configuration="Release"
Microsoft (R)-Buildmodule, Version 4.0.30319.17929
```

# Resources

- ★ **The official F# site**  
<http://fsharp.org/>
- ★ **An excellent blog series to learn functional programming and F#**  
<http://fsharpforfunandprofit.com/>
- ★ **FsLab: Data science tools**  
<http://fslab.org/>
- ★ **FAKE: a cross platform build automation tool**  
<http://fsharp.github.io/FAKE/>
- ★ **Twenty six low-risk ways to use F# at work**  
<http://fsharpforfunandprofit.com/posts/low-risk-ways-to-use-fsharp-at-work/>
- ★ **Training examples on GitHub**  
<https://github.com/lventre/Trainings.git>