This cheatsheet aims to succinctly cover the most important aspects of F# 7.0.

The Microsoft F# Documentation is complete and authoritative and has received a lot of love in recent years; it's well worth the time investment to read. Only after you've got the lowdown here of course ;)

If you have any comments, corrections, or suggested additions, please open an issue or send a pull request to https://github.com/fsprojects/fsharp-cheatsheet. Questions are best addressed via the F# slack or the F# discord.

Contents

Comments

Block comments are placed between `(*` and `*)`. Line comments start from `//` and continue until the end of the line.

```
(* This is block comment *)

// And this is line comment
```

XML doc comments come after `///` allowing us to use XML tags to generate documentation.

```
/// The `let` keyword defines an (immutable) value
let result = 1 + 1 = 2
```

Strings

F# `string` type is an alias for `System.String` type.

```
// Create a string using string concatenation
let hello = "Hello" + " World"
```

Use *verbatim strings* preceded by `@` symbol to avoid escaping control characters (except escaping `"` by `""`).

```
let verbatimXml = @"<book title=""Paradise Lost"">"
```

We don't even have to escape `"` with *triple-quoted strings*.

```
let tripleXml = """<book title="Paradise Lost">"""
```

Backslash strings indent string contents by stripping leading spaces.

```
let poem =
    "The lesser world was daubed\n\
     By a colorist of modest skill\n\
     A master limned you in the finest inks\n\
     And with a fresh-cut quill."
```

String Slicing is supported by using `[start..end]` syntax.

```
let str = "Hello World"
let firstWord = str[0..4] // "Hello"
let lastWord = str[6..] // "World"
```

String Interpolation is supported by prefixing the string with `$` symbol. All of these will output `"Hello" \ World!`:

```
let expr = "Hello"
printfn " \"%s\" \\ World! " expr
printfn $" \"{expr}\" \\ World! "
printfn $" \"%s{expr}\" \\ World! " // using a format specifier
printfn $@" ""{expr}"" \ World! "
printfn $@" ""%s{expr}"" \ World! "
```

See Strings (MS Learn) for more on escape characters, byte arrays, and format specifiers.

Basic Types and Literals

Integer Prefixes for hexadecimal, octal, or binary

```
let numbers = (0x9F, 0o77, 0b1010)  // (159, 63, 10)
```

Literal Type Suffixes for integers, floats, decimals, and ascii arrays

```
let ( sbyte, byte   )  = ( 55y, 55uy )  // 8-bit integer

let ( short, ushort )  = ( 50s, 50us )  // 16-bit integer

let ( int,   uint   )  = ( 50,  50u  )  // 32-bit integer

let ( long,  ulong  )  = ( 50L, 50uL )  // 64-bit integer
```

```
let bigInt              = 9999999999999I // System.Numerics.BigInteger

let float               = 50.0f          // signed 32-bit float

let double              = 50.0           // signed 64-bit float

let scientific          = 2.3E+32        // signed 64-bit float

let decimal             = 50.0m          // signed 128-bit decimal

let byte                = 'a'B           // ascii character; 97uy

let byteArray           = "text"B        // ascii string; [|116uy; 101uy; 120uy; 116uy|]
```

Primes (or a tick `'` at the end of a label name) are idiomatic to functional languages and are included in F#. They are part of the identifier's name and simply indicate to the developer a variation of an existing value or function. For example:

```
let x = 5
let x' = x + 1
let x'' = x' + 1
```

See Literals (MS Learn) for complete reference.

Functions

The `let` keyword also defines named functions.

```
let negate x = x * -1
let square x = x * x
let print x = printfn "The number is: %d" x
```

Pipe and composition operators

Pipe operator `|>` is used to chain functions and arguments together. Double-backtick identifiers are handy to improve readability especially in unit testing:

```
let ``square, negate, then print`` x =
    x |> square |> negate |> print
```

This operator can assist the F# type checker by providing type information before use:

```
let sumOfLengths (xs : string []) =
    xs
    |> Array.map (fun s -> s.Length)
    |> Array.sum
```

Composition operator >> is used to compose functions:

```
let squareNegateThenPrint' =
    square >> negate >> print
```

Recursive Functions

The rec keyword is used together with the let keyword to define a recursive function:

```
let rec fact x =
    if x < 1 then 1
    else x * fact (x - 1)
```

Mutually recursive functions (those functions which call each other) are indicated by and keyword:

```
let rec even x =
    if x = 0 then true
    else odd (x - 1)

and odd x =
    if x = 0 then false
    else even (x - 1)
```

Pattern Matching

Pattern matching is often facilitated through match keyword.

```
let rec fib n =
    match n with
    | 0 -> 0
    | 1 -> 1
    | _ -> fib (n - 1) + fib (n - 2)
```

In order to match sophisticated inputs, one can use when to create filters or guards on patterns:

```
let sign x =
    match x with
    | 0 -> 0
    | x when x < 0 -> -1
    | x -> 1
```

Pattern matching can be done directly on arguments:

```
let fst' (x, _) = x
```

or implicitly via function keyword:

```
/// Similar to `fib`; using `function` for pattern matching
```

```
let rec fib' = function
    | 0 -> 0
    | 1 -> 1
    | n -> fib' (n - 1) + fib' (n - 2)
```

For more complete reference visit Pattern Matching (MS Learn).

Collections

Lists

A *list* is an immutable collection of elements of the same type.

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

// Recursion on list using (::) operator
let rec sum list =
    match list with
    | [] -> 0
    | x :: xs -> x + sum xs
```

Arrays

Arrays are fixed-size, zero-based, mutable collections of consecutive data elements.

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

Sequences

A *sequence* is a logical series of elements of the same type. Individual sequence elements are computed only as required, so a sequence can provide better performance than a list in situations in which not all the elements are used.

```
// Sequences can use yield and contain subsequences
let seq1 =
    seq {
        // "yield" adds one element
        yield 1
        yield 2

        // "yield!" adds a whole subsequence
```

```
        yield! [5..10]
    }
```

Higher-order functions on collections

The same list `[ 1; 3; 5; 7; 9 ]` or array `[| 1; 3; 5; 7; 9 |]` can be generated in various ways.

- Using range operator `..`

  ```
  let xs = [ 1..2..9 ]
  ```

- Using list or array comprehensions

  ```
  let ys = [| for i in 0..4 -> 2 * i + 1 |]
  ```

- Using `init` function

  ```
  let zs = List.init 5 (fun i -> 2 * i + 1)
  ```

Lists and arrays have comprehensive sets of higher-order functions for manipulation.

- `fold` starts from the left of the list (or array) and `foldBack` goes in the opposite direction

  ```
  let xs' =
      Array.fold (fun str n ->
          sprintf "%s,%i" str n) "" [| 0..9 |]
  ```

- `reduce` doesn't require an initial accumulator

  ```
  let last xs = List.reduce (fun acc x -> x) xs
  ```

- `map` transforms every element of the list (or array)

  ```
  let ys' = Array.map (fun x -> x * x) [| 0..9 |]
  ```

- `iter`ate through a list and produce side effects

  ```
  let _ = List.iter (printfn "%i") [ 0..9 ]
  ```

All these operations are also available for sequences. The added benefits of sequences are laziness and uniform treatment of all collections implementing `IEnumerable<'T>`.

```
let zs' =
    seq {
        for i in 0..9 do
            printfn "Adding %d" i
            yield i
    }
```

Tuples and Records

Tuple

A *tuple* is a grouping of unnamed but ordered values, possibly of different types:

```
// Tuple construction
let x = (1, "Hello")

// Triple
let y = ("one", "two", "three")

// Tuple deconstruction / pattern
let (a', b') = x
```

The first and second elements of a tuple can be obtained using `fst`, `snd`, or pattern matching:

```
let c' = fst (1, 2)
let d' = snd (1, 2)

let print' tuple =
    match tuple with
    | (a, b) -> printfn "Pair %A %A" a b
```

Record

Records represent simple aggregates of named values, optionally with members:

```
// Declare a record type
type Person = { Name : string; Age : int }

// Create a value via record expression
let paul = { Name = "Paul"; Age = 28 }

// 'Copy and update' record expression
let paulsTwin = { paul with Name = "Jim" }
```

Records can be augmented with properties and methods:

```
type Person with
    member x.Info = (x.Name, x.Age)
```

Records are essentially sealed classes with extra topping: default immutability, structural equality, and pattern matching support.

```
let isPaul person =
    match person with
    | { Name = "Paul" } -> true
    | _ -> false
```

Discriminated Unions

Discriminated unions (DU) provide support for values that can be one of a number of named cases, each possibly with different values and types.

```
type Tree<'T> =
    | Node of Tree<'T> * 'T * Tree<'T>
    | Leaf

let rec depth = function
    | Node(l, _, r) -> 1 + max (depth l) (depth r)
    | Leaf -> 0
```

F# Core has built-in discriminated unions for error handling, e.g., `option` and `Result`.

```
let optionPatternMatch input =
    match input with
    | Some i -> printfn "input is an int=%d" i
    | None -> printfn "input is missing"
```

Single-case discriminated unions are often used to create type-safe abstractions with pattern matching support:

```
type OrderId = Order of string

// Create a DU value
let orderId = Order "12"

// Use pattern matching to deconstruct single-case DU
let (Order id) = orderId
```

Statically Resolved Type Parameters

A *statically resolved type parameter* is a type parameter that is replaced with an actual type at compile time instead of at run time. They are primarily useful in conjunction with member constraints.

```
let inline add x y = x + y
let integerAdd = add 1 2
let floatAdd = add 1.0f 2.0f // without `inline` on `add` function, this would cause a type
```

```
type RequestA = { Id: string; StringValue: string }
type RequestB = { Id: string; IntValue: int }

let requestA: RequestA = { Id = "A"; StringValue = "Value" }
let requestB: RequestB = { Id = "B"; IntValue = 42 }
```

```
let inline getId<'t when 't : (member Id: string)> (x: 't) = x.Id

let idA = getId requestA // "A"
let idB = getId requestB // "B"
```

See Statically Resolved Type Parameters (MS Learn) and Constraints (MS Learn) for more examples.

Exceptions

Try..With

An illustrative example with: custom F# exception creation, all exception aliases, `raise()` usage, and an exhaustive demonstration of the exception handler patterns:

```
open System
exception MyException of int * string // (1)
let guard = true

try
    failwith   "Message"                 // throws a System.Exception (aka exn)
    nullArg    "ArgumentName"            // throws a System.ArgumentNullException
    invalidArg "ArgumentName" "Message" // throws a System.ArgumentException
    invalidOp  "Message"                 // throws a System.InvalidOperation

    raise(NotImplementedException("Message")) // throws a .NET exception (2)
    raise(MyException(0, "Message"))          // throws an F# exception (2)

    true // (3)
with
| :? ArgumentNullException                      -> printfn "NullException"; false // (3)
| :? ArgumentException as ex                    -> printfn $"{ex.Message}"; false // (4)
| :? InvalidOperationException as ex when guard -> printfn $"{ex.Message}"; reraise() // (5
| MyException(num, str) when guard              -> printfn $"{num}, {str}"; false // (5)
| MyException(num, str)                         -> printfn $"{num}, {str}"; reraise() // (6
| ex when guard                                 -> printfn $"{ex.Message}"; false
| ex                                            -> printfn $"{ex.Message}"; false
```

(1) define your own F# exception types with `exception`, a new type that will inherit from `System.Exception`;
(2) use `raise()` to throw an F# or .NET exception;
(3) the entire `try..with` expression must evaluate to the same type, in this example: bool; (4)`ArgumentNullException` inherits from `ArgumentException`, so `ArgumentException` must follow after;
(4) support for `when` guards;

(5) use `reraise()` to re-throw an exception; works with both .NET and F# exceptions

The difference between F# and .NET exceptions is how they are created and how they can be handled.

Try..Finally

The `try..finally` expression enables you to execute clean-up code even if a block of code throws an exception. Here's an example that also defines custom exceptions.

```
exception InnerError of string
exception OuterError of string

let handleErrors x y =
    try
        try
            if x = y then raise (InnerError("inner"))
            else raise (OuterError("outer"))
        with
        | InnerError str -> printfn "Error1 %s" str
    finally
        printfn "Always print this."
```

Note that `finally` does not follow `with`. `try..with` and `try..finally` are separate expressions.

Classes and Inheritance

This example is a basic class with (1) local let bindings, (2) properties, (3) methods, and (4) static members.

```
type Vector(x : float, y : float) =
    let mag = sqrt(x * x + y * y) // (1)
    member _.X = x // (2)
    member _.Y = y
    member _.Mag = mag
    member _.Scale(s) = // (3)
        Vector(x * s, y * s)
    static member (+) (a : Vector, b : Vector) = // (4)
        Vector(a.X + b.X, a.Y + b.Y)
```

Call a base class from a derived one.

```
type Animal() =
    member _.Rest() = ()

type Dog() =
```

```
    inherit Animal()
    member _.Run() =
        base.Rest()
```

Upcasting is denoted by `:>` operator.

```
let dog = Dog()
let animal = dog :> Animal
```

Dynamic downcasting (`:?>`) might throw an `InvalidCastException` if the cast doesn't succeed at runtime.

```
let shouldBeADog = animal :?> Dog
```

Interfaces and Object Expressions

Declare `IVector` interface and implement it in `Vector'`.

```
type IVector =
    abstract Scale : float -> IVector

type Vector'(x, y) =
    interface IVector with
        member __.Scale(s) =
            Vector'(x * s, y * s) :> IVector
    member _.X = x
    member _.Y = y
```

Another way of implementing interfaces is to use *object expressions*.

```
type ICustomer =
    abstract Name : string
    abstract Age : int

let createCustomer name age =
    { new ICustomer with
      member __.Name = name
      member __.Age = age }
```

Active Patterns

Single-case active patterns

```
// Basic
let (|EmailDomain|) email =
    let match' = Regex.Match(email, "@(.*)$")
    if match'.Success
    then match'.Groups[1].ToString()
    else ""
let (EmailDomain emailDomain) = "yennefer@aretuza.org"  // emailDomain = 'aretuza.org'
```

```
// As Parameters
open System.Numerics
let (|Real|) (x: Complex) =
    (x.Real, x.Imaginary)
let addReal (Real (real1, _)) (Real (real2, _)) =  // conversion done in the parameters
    real1 + real2
let addRealOut = addReal Complex.ImaginaryOne Complex.ImaginaryOne

// Parameterized
let (|Default|) onNone value =
    match value with
    | None -> onNone
    | Some e -> e
let (Default "random citizen" name) = None // name = "random citizen"
let (Default "random citizen" name) = Some "Steve" // name = "Steve"
```

Single-case active patterns can be thought of as a simple way to convert data to a new form.

Complete active patterns

```
let (|Even|Odd|) i =
    if i % 2 = 0 then Even else Odd

let testNumber i =
    match i with
    | Even -> printfn "%d is even" i
    | Odd -> printfn "%d is odd" i

let (|Phone|Email|) (s:string) =
    if s.Contains '@' then Email $"Email: {s}" else Phone $"Phone: {s}"

match "yennefer@aretuza.org" with // output: "Email: yennefer@aretuza.org"
| Email email -> printfn $"{email}"
| Phone phone -> printfn $"{phone}"
```

Partial active patterns

```
let (|DivisibleBy|_|) by n =
    if n % by = 0 then Some DivisibleBy else None

let fizzBuzz = function
    | DivisibleBy 3 & DivisibleBy 5 -> "FizzBuzz"
    | DivisibleBy 3 -> "Fizz"
    | DivisibleBy 5 -> "Buzz"
    | i -> string i
```

Partial active patterns share the syntax of parameterized patterns but their active recognizers accept only one argument.

Code Organization

Modules

Modules are key building blocks for grouping related code; they can contain `types`, `let` bindings, or (nested) sub `modules`. Identifiers within modules can be referenced using dot notation, or you can bring them into scope via the `open` keyword. Illustrative-only example:

```
module Money =
    type CardInfo =
        { number: string
          expiration: int * int }

    type Payment =
        | Card of CardInfo
        | Cash of int

    module Functions =
        let validCard (cardNumber: string) =
            cardNumber.Length = 16 && (cardNumber[0], ['3';'4';'5';'6']) ||> List.contains
```

If there is only one module in a file, the `module` name can be declared at the top, and all code constructs within the file will be included in the `module`s definition (no indentation required).

```
module Functions  // notice there is no '=' when at the top of a file

let sumOfSquares n = seq {1..n} |> Seq.sumBy (fun x -> x * x)  // Functions.sumOfSquares
```

Namespaces

Namespaces are simply dotted names that prefix `type` and `module` declarations to allow for hierarchical scoping. The first `namespace` directives must be placed at the top of the file. Subsequent `namespace` directives either: (a) create a sub-namespace; or (b) create a new namespace.

```
namespace MyNamespace

module MyModule =  // MyNamspace.MyModule
    let myLet = ...  // MyNamspace.MyModule.myLet

namespace MyNamespace.SubNamespace

namespace MyNewNamespace  // a new namespace
```

A top-level `module`'s namespace can be specified via a dotted prefix:

```
module MyNamespace.SubNamespace.Functions
```

Open and AutoOpen

The `open` keyword can be used on `module`, `namespace`, and `type`.

```
module Groceries =
    type Fruit =
        | Apple
        | Banana

let fruit1 = Groceries.Banana
open Groceries  // module
let fruit2 = Apple
```

```
open System.Diagnostics  // namespace
let stopwatch = Stopwatch.StartNew()  // Stopwatch is accessible
```

```
open type System.Text.RegularExpressions.Regex  // type
let isHttp url = IsMatch("^https?:", url)  // Regex.IsMatch directly accessible
```

Available to `module` declarations only, is the `AutoOpen` attribute, which alleviates the need for an `open`.

```
[<AutoOpen>]
module Groceries =
    type Fruit =
        | Apple
        | Banana

let fruit = Banana
```

However, `AutoOpen` should be used cautiously. When an `open` or `AutoOpen` is used, all declarations in the containing element will be brought into scope. This can lead to shadowing; where the last named declaration replaces all prior identically-named declarations. There is *no* error - or even a warning - in F#, when shadowing occurs. A coding convention (MS Learn) exists for `open` statements to avoid pitfalls; `AutoOpen` would sidestep this.

Accessibility Modifiers

F# supports `public`, `private` (limiting access to its containing `type` or `module`) and `internal` (limiting access to its containing assembly). They can be applied to `module`, `let`, `member`, `type`, `new` (MS Learn), and `val` (MS Learn).

With the exception of `let` bindings in a class `type`, everything defaults to `public`.

Element	Example with Modifier
Module	`module internal MyModule =`
Module .. `let`	`let private value =`
Record	`type internal MyRecord = { id: int }`
Record ctor	`type MyRecord = private { id: int }`
Discriminated Union	`type internal MyDiscUni = A | B`
Discriminated Union ctor	`type MyDiscUni = private A | B`
Class	`type internal MyClass() =`
Class ctor	`type MyClass private () =`
Class Additional ctor	`internal new() = MyClass("defaultValue")`
Class .. `let`	*Always private. Cannot be overridden*
`type` .. `member`	`member private _.TypeMember =`
`type` .. `val`	`val internal explicitInt : int`

Smart Constructors

Making a primary constructor (ctor) `private` or `internal` is a common convention for ensuring value integrity; otherwise known as "making illegal states unrepresentable" (YouTube:Effective ML).

Example of Single-case Discriminated Union with a `private` constructor that constrains a quantity between 0 and 100:

```
type UnitQuantity =
    private UnitQuantity of int

module UnitQuantity =  // common idiom: type companion module
    let tryCreate qty =
        if qty < 1 || qty > 100
        then None
        else Some (UnitQuantity qty)
    let value (UnitQuantity uQty) = uQty
    let zero = UnitQuantity 0
...
let unitQtyOpt = UnitQuantity.tryCreate 5

let validQty =
    unitQtyOpt
    |> Option.defaultValue UnitQuantity.zero
```

Recursive Reference

F#'s type inference and name resolution runs in file and line order. By default, any forward references are considered errors. This default provides a single benefit, which can be hard to appreciate initially: you never need to look beyond the current file for a dependency. In general this also nudges toward more careful design and organisation of codebases, which results in cleaner, maintainable code. However, in rare cases forward referencing might be needed. To do this we have `rec` for `module` and `namespace`; and `and` for `type` and `let` (Recursive Functions) functions.

```
module rec CarModule

exception OutOfGasException of Car  // Car not defined yet; would be an error

type Car =
    { make: string; model: string; hasGas: bool }
    member self.Drive destination =
        if not self.hasGas
        then raise (OutOfGasException self)
        else ...



type Person =
    { Name: string; Address: Address }
and Address =
    { Line1: string; Line2: string; Occupant: Person }
```

See Namespaces (MS Learn) and Modules (MS Learn) to learn more.

Compiler Directives

Load another F# source file into FSI.

```
#load "../lib/StringParsing.fs"
```

Reference a .NET assembly (`/` symbol is recommended for Mono compatibility).
Reference a .NET assembly:

```
#r "../lib/FSharp.Markdown.dll"
```

Reference a nuget package

```
#r "nuget:Serilog.Sinks.Console" // latest production release
#r "nuget:FSharp.Data, 6.3.0"    // specific version
#r "nuget:Equinox, *-*"          // latest version, including `-alpha`, `-rc` version etc
```

Include a directory in assembly search paths.

```
#I "../lib"
```

```
#r "FSharp.Markdown.dll"
```

Other important directives are conditional execution in FSI (`INTERACTIVE`) and querying current directory (`__SOURCE_DIRECTORY__`).

```
#if INTERACTIVE
let path = __SOURCE_DIRECTORY__ + "../lib"
#else
let path = "../../../lib"
#endif
```