

# 1 Syntax, variables, functions

## Key concepts:

1. Having a good text editor helps you code much easier.
2. (a) Once defined, a variable in F# cannot change value (unless "mutable" is used)  
(b) If you need an updated value, create a new one.
3. Different datatypes (e.g. integer and decimal-numbers) do not combine easily.
4. Defining and using functions in F# is slightly different from math notation/ other languages.
  - (a) F# automatically detects the type of the variables (e.g. integer, double, etc.) for a function.
  - (b) The variable types for a function will be enforced.

## 1.1 Setting Up

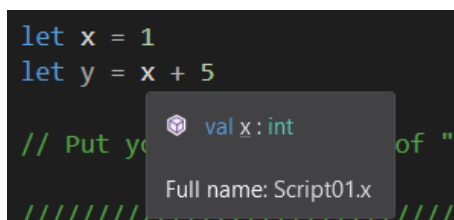
### 1.1.1 Comments

You can use double-slash `//`, triple-slash `///`, or star-bracket `(* ..... *)` to make comments.

```
1 // These words are ignored.
2 /// These words are ignored.
3 (* These words are ignored. *)
4 let x = 1
5 let y = x + 5
```

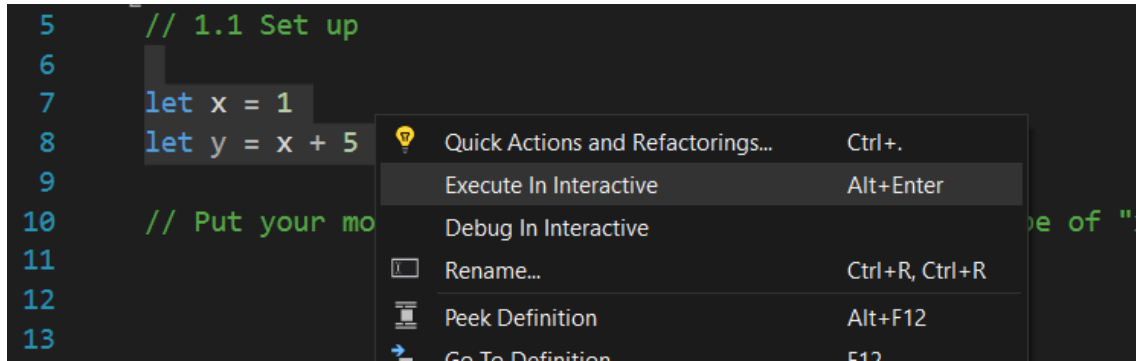
### 1.1.2 Intellisense

If you are using Visual Studio or Visual Studio Code, you can put your mouse on top of the variable name `x` or `y`, and see that it is an `int` or integer. This feature will help you identify what is each variable/function, and make coding easier for you.



### 1.1.3 F# Interactive

If you are using Visual Studio, you can run the code above by highlighting/selecting the code using your mouse, and press ALT + ENTER, or right-click and select **Execute** in Interactive.



## 1.2 Data Type

### 1.2.1 Common data types and printing

Some of the common types in F# are:

Keyword	Description
int	Integer
double or float	Decimal numbers
string	Words/Sentences
bool	True/False

### 1.2.2 Equality and simple if-else

The `let ... = ...` combination is used to assigned a value to a variable. Other than this situation, the equal sign `=` is used for equality testing. `=`, `<>` are used for equality/inequality testing.

In Java/C++/Python, `==`, `!=` are used for comparison, and in Javascript, `===`, `!==` are used.

```
1 let valueToTest = 20
2 let isValueEqualToTwenty = (valueToTest = 20)
3
4 if isValueEqualToTwenty then
5     printfn "Yes, the value is Twenty"
6 else
7     printfn "No, the value is not Twenty"
8 // Output: "Yes, the value is Twenty"
```

In the example above, `(valueToTest = 20)` is a boolean, so the equal sign in this expression is used for equality testing.

```

1 let inputUserName = "Jack"
2
3 if inputUserName = "John" then
4     printfn "Welcome back, John"
5 else
6     printfn "Access denied."
7 // Output: "Access denied."

```

### 1.2.3 Immutability

In F#, variables are by default immutable/unchangeable. Once defined, the value of a variable cannot be changed. You can make a variable changable/mutable using the keyword `mutable` and symbol `<-`, but this is highly discouraged. (If you use VisualStudio, then the color of the variable name will change color, warning you of potential mutable values)

```

// Warning: Do not use mutable value if possible!
//
// Using mutable value is a bad idea!
let mutable changableValue = 100
printfn "Original value is: %i" changableValue
// Output: "Original value is: 100"

changableValue <- 200
printfn "Updated value is: %i" changableValue
// Output: "Updated value is: 200"

```

If you try to update an immutable/unchangeable value using `<-`, you will get an error.

```

////////////////////
// #####
// This contains ERROR! //#
let immutableValue = 100 //#
immutableValue <- 300    //#

```

This value is not mutable. Consider using the mutable keyword, e.g. 'let mutable immutableValue = expression'.

## Benefit of immutable/unchangeable values

Imagine the code below, with a mutable value `x`, and after thousands of lines of code later, you used `x`'s value again:

```
1 let mutable x = 100
2 // Thousands of lines of code later.....
3 // You have many lines of code in between.....
4 // It is hard to keep track.....
5 // Have you changed/updated x's value?
6 // Did you accidentally call any function that modify x?
7 // Can you guarantee x's value stay unchanged?
8 let y = x + 1
9 // What is the value of y?
10 //
11 // That depends on what happens between y's definition
12 // and x's definition.
```

---

On the other hand, if `x` is immutable/unchangeable:

```
1 let x = 100
2 // Thousands of lines of code later.....
3 // You have many lines of code in between.....
4 // But because x is immutable/unchangeable.....
5 // We can be sure that x stays constant.....
6 // And we can safely conclude that.....
7 let y = x + 1
8 // y = 101
```

Conclusion: Use immutable/unchangeable value whenever possible. AVOID mutable/changable value whenever possible.

Remark: Immutability/unchangeable value really shines when doing concurrent programming, but we will not go into details here.

### 1.2.4 (+) Operator on the same type of variable

Integers, double, and string support the (+) operation:

```
1 let addTwoInteger = 40 + 55
2
3 // Remark: "float" and "double" mean the same thing in F#.
4 let addTwoDecimals = 1.414 + 3.1415926
5
6 let sentenceStart = "My school is "
7 let schoolName = "National University of Singapore"
8 let combinedSentence = sentenceStart + schoolName
```

However, you cannot add an integer with a decimal in F# directly using (+), and you cannot add/concatenate a string with a number directly using (+). If you use VisualStudio, then you may see an error similar to the one below.

```

////#####
// Cannot combine two different types using the "+" functions    //
// The following codes contain ERROR!                            //
//                                                                //
let addIntegerWithDecimal = 15 + 4.11                          //
let combineStringWithInteger = "My age is: " + 21              //
////#####

```

The type 'int' does not match the type 'string'

Similarly, we can only use the multiply symbol (\*) to multiply two integers or two decimals, NOT an integer with a decimal number.

Furthermore, some functions, like the square root `sqrt` and math exponent (\*\*) only accepts decimal numbers:

```

1 let sqrtRootOfNine = sqrt 9.0
2 let twoToPowerOfFive = 2.0 ** 5.0

```

And it will cause error if you use them with integer input instead.

```

//#####
// ERROR: sqrt and (power **) only accepts double/decimals/float //
let twoToPowerOfFiveError = 2 ** 5                                //
let sqrtRootOfNineError = sqrt 9                                //
// ERROR!                                                         //
//#####

```

The type 'int' does not support the operator 'Sqrt'

## 1.3 Functions

### 1.3.1 One variable functions

You can define functions using `let` followed by the inputs of your function.

```

1 let f x = x + 5
2
3 let result1 = f 10
4 let result2 = f 20

```

Output:

```

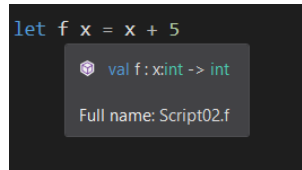
1 // val result1 : int = 15
2 // val result2 : int = 25

```

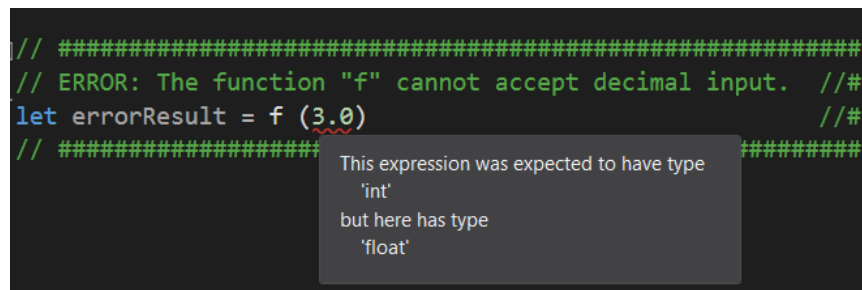
Notice the following:

1. To apply the function `f`, you do not need to use the math notation  $f(x)$ . You can apply the arguments by separating with a space.

2. If you hover your mouse on top of the function `f`, you will see that `f` is a function that accepts only integer `x` as the argument.



- (a) This is because in the function, `x` will be added (+) to the integer 5. We have seen before that we cannot use the symbol (+) to combine an integer with a decimal number directly. Hence, `x` has to be of type `int`.
- (b) As a consequence, if you try to input a decimal number to the function `f`, then it will fail:



3. As mentioned, F# automatically inferred that `x` is an integer. This is different from other languages (e.g. Java, C++) that needs you to specify the type of the variable (is it an integer? double? etc.)

So, you can spend less time on the tiny details (e.g. what is the variable type), and focus more on the correctness of your program.

Similarly, the following function accepts decimals/double only.

```
1 let DiscountFunc originalPrice = originalPrice * 0.8
2
3 let discountedPrice = DiscountFunc 399.99
4 printfn "New price: %.2f" discountedPrice
5 // Output: "New price: 319.99"
6
7 let anotherDiscount = DiscountFunc discountedPrice
8 printfn "New price: %.2f" anotherDiscount
9 // Output: "New price: 255.99"
```

Remark: The `%.2f` for printing 2 decimals.

This function does not accept integer values:

```
// #####
// ERROR: The function "DiscountFunc" cannot accept integer input.  //#
let errorResult = DiscountFunc 100                                     //#
// #####
```

This expression was expected to have type  
'float'  
but here has type  
'int'

We need to convert integer to decimal (using double or float) before using the function.

```
1 let convertedPrice = double 100
2 let decimalResult = DiscountFunc convertedPrice
3 printfn "New price: %.2f" decimalResult
4 // Output: "New price: 80.00"
```

Similarly, the following function accepts strings only.

```
1 // Define a function for string.
2 let AddGreeting name =
3     "Hello " + name
4
5 let greeting1 = AddGreeting "John"
6 let greeting2 = AddGreeting "Mary"
```

Output:

```
1 // val greeting1 : string = "Hello John"
2 // val greeting2 : string = "Hello Mary"
```

And it will cause error if you try to input an integer value to this function:

```
// #####
// ERROR: AddGreeting function does not accept integer/double/etc.  //#
let greetingError = AddGreeting 123                                     //#
// #####
```

This expression was expected to have type  
'string'  
but here has type  
'int'

Exercise: Write a function that calculates the area of a circle of radius  $r$ .

```
1 let CircleArea r =
2     //
3     //
4     // Hint: Use "System.Math.PI"
```

### 1.3.2 Two variable functions

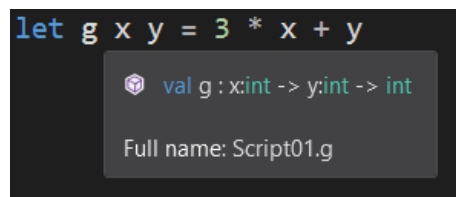
You can define a function that takes in two variables:

```
1 let g x y = 3 * x + y
2
3 let result3 = g 3 1
4 let result4 = g 10 2
```

```
1 // val result3 : int = 10
2 // val result4 : int = 32
```

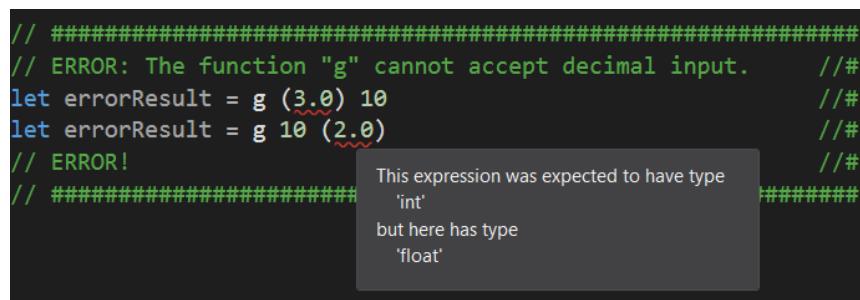
Notice the following:

1. To apply the function `g`, you do not need to use the math notation  $g(x, y)$  with brackets and commas. This is different from other programming languages (e.g. Java, C++). You can apply the arguments by separating with a space.
2. If you hover your mouse on top of `g`, as seen in this picture:



You will see that the variables `x`, `y` need to be integers.

- (a) This is because in the function, `x` will be multiplied with 3, and then later added with `y`. As seen before, the addition and multiplication symbol (+), (\*) only combined numbers of the same type (integers with integers, double with double)
- (b) As a consequence, if you input decimals into the function, it will fail:



3. Again, you can spend less time typing out the details (i.e. what are the types of `x`, `y`? Integer? Double?) and focus more on making your program/algorithm works, and make yourself more productive (compared to other programming languages)

Similarly, the following function accepts two decimal numbers:

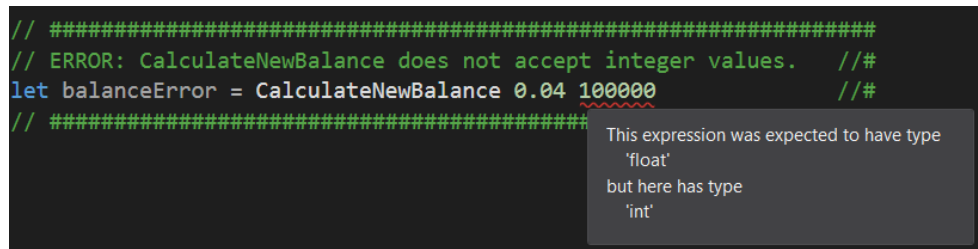


```

1 let CalculateNewBalance interestRate principal =
2     principal * (1.0 + interestRate)
3
4 let balance1 = CalculateNewBalance 0.05 100000.00
5 printfn "New Balance: %f" balance1
6 // Output: "New Balance: 105000.00"
7
8 let balance2 = CalculateNewBalance 0.03 5000.00
9 printfn "New Balance: %f" balance2
10 // Output: "New Balance: 5150.00"

```

And it will cause error if you try to change one of the input into integer.



```

// #####
// ERROR: CalculateNewBalance does not accept integer values.  // #
let balanceError = CalculateNewBalance 0.04 100000 // #
// #####

```

This expression was expected to have type 'float' but here has type 'int'

### 1.3.3 Multivariable functions

```

1 let h x y z = 3 * x + 4 * y + 5 * z
2
3 // 3*3 + 4*4 + 5*5 = 50
4 let result5 = h 3 4 5
5
6 // 3*1 + 4*1 + 5*1 = 12
7 let result6 = h 1 1 1

```

### 1.3.4 Limitation: Default integers for +, \*

If you use (+), (\*) with no other information available (e.g. an appearance of a decimal, string, etc.), then F# will assume the function variables as integers.

```

1 let AddThree x y z = x + y + z
2 let addThreeResult = AddThree 5 6 7

```

If you hover your mouse on top of AddThree, then you see that all the inputs are inferred to be integers.

If you want this function to work for decimals, then you will need to annotate/manually add in the type for one of the variables:

```

1 let AddThreeCustom (x:double) y z = x + y + z

```

Here, we are explicitly saying that x is a double. And since y, z interacts with x using (+), we can infer that y, z are also doubles (and we do not need to explicitly label them as decimal/doubles)

## 1.4 Scoping

### 1.4.1 Indenting

You can use a `let` inside a `let`, i.e. you can define a variable inside a variable. For example:

```
1 let AddFriend person1 =  
2     let endOfSentence = " and Mary are friends"  
3     person1 + endOfSentence  
4  
5 let combinedSentence1 = AddFriend "Jack"
```

Output:

```
1 // combinedSentence1 : string = "Jack and Mary are friends"
```

Notice that the two lines immediately after the `AddFriend` function has some spaces in front of each line. This means that those two lines are accessible only inside the `AddFriend` function.

So, you cannot access the `endOfSentence` variable outside of the function. The following code will not work:

```
1 let AddFriend person1 =  
2     let endOfSentence = " and Mary are friends"  
3     person1 + endOfSentence  
4  
5 // ERROR: "endOfSentence" is not accessible outside of "  
6   AddFriend"  
7 let x = endOfSentence  
8 "ERROR: endOfSentence is not accessible outside of  
9   AddFriend"
```

### 1.4.2 Reuse variable name

By carefully using indenting/spacing, you can repeatedly use the same variable name, as long as the spacing/indenting is such that the variables do not cause conflict with each other.

```
1 let DrinkFunction person =  
2     let endOfSentence = " likes to drink coffee."  
3     person + endOfSentence  
4  
5 let EatFunction person =  
6     let endOfSentence = " prefers eating chocolate."  
7     person + endOfSentence  
8  
9 printfn "%s" (DrinkFunction "Jack")  
10 // Output:
```

```

11 // "Jack likes to drink coffee."
12
13 printfn "%s" (EatFunction "Jill")
14 // Output:
15 // "Jill prefers eating chocolate."

```

The `endOfSentence` inside these two functions will not cause conflict with each other.

### 1.4.3 From top to bottom

F# code are read from top to bottom. For example, look at the following code:

```

1 let a = 5
2
3 let f1 b =
4     a + b
5
6 let f2 b =
7     a + a + b
8
9 printfn "%i" (f1 10)
10 printfn "%i" (f2 10)

```

Notice that there are no spacing/indenting before `let a = 5` and the definition of `f1`, `f2`. These variables and functions are equally indented, and so the value of `a` is accessible from `f1`, `f2`

However, the following code below will not be accepted, because `a` is defined later/down lower in the code, but it is incorrectly used before it is defined (i.e. above it).

```

1 // ERROR: "a" is not yet defined.
2 let f1 b =
3     a + b
4 "ERROR!"
5 // ERROR: "a" is not yet defined.
6 let f2 b =
7     a + a + b
8 "ERROR!"
9 // ERROR: "a" is defined too late! It is used above.
10 let a = 5

```


### 1.4.4 Warning: No TAB

In Python, you use `TAB` to indent the file. The `TAB` button will insert a special character.

However, in F#, you use blank spaces to do indenting. You should configure/adjust your IDE (e.g. VisualStudio, VisualStudioCode, etc.) so that it insert multiple blank spaces instead of a special character.

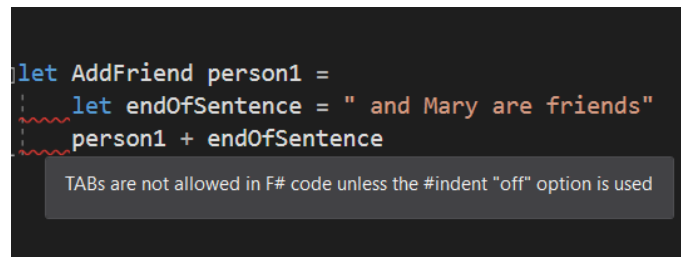
For example, the code below is indented using 4 spaces for the second and third line.

These are 4 blank spaces! Not the special character "TAB"



```
1 let AddFriend person1 =  
2     let endOfSentence = " and Mary are friends"  
3     person1 + endOfSentence
```

If you did not configure your IDE correctly, or if you copy-and-paste the special TAB character from another source (e.g. Notepad), then you may see the following error:

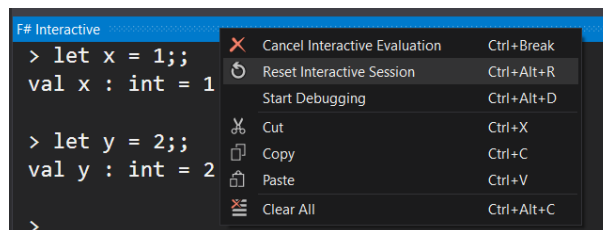


```
let AddFriend person1 =  
    let endOfSentence = " and Mary are friends"  
    person1 + endOfSentence
```

TABs are not allowed in F# code unless the #indent "off" option is used

## 1.5 Reset F# Interactive

Remember to reset your F# Interactive once in a while, so that you don't have too many previous variables (especially if you re-use the same variable names)



In Visual Studio, you can Right-click the interactive window, and select "Reset Interactive Session", or use the shortcut key CTRL + ALT + R