The anatomy of Functions in Go

Like JavaScript, functions in Go are first class citizens. They can be assigned to variables, passed as an argument, immediately invoked or deferred for last execution.

Uday Hiwarale Follow
Aug 18, 2018 · 9 min read

☛ What is a function

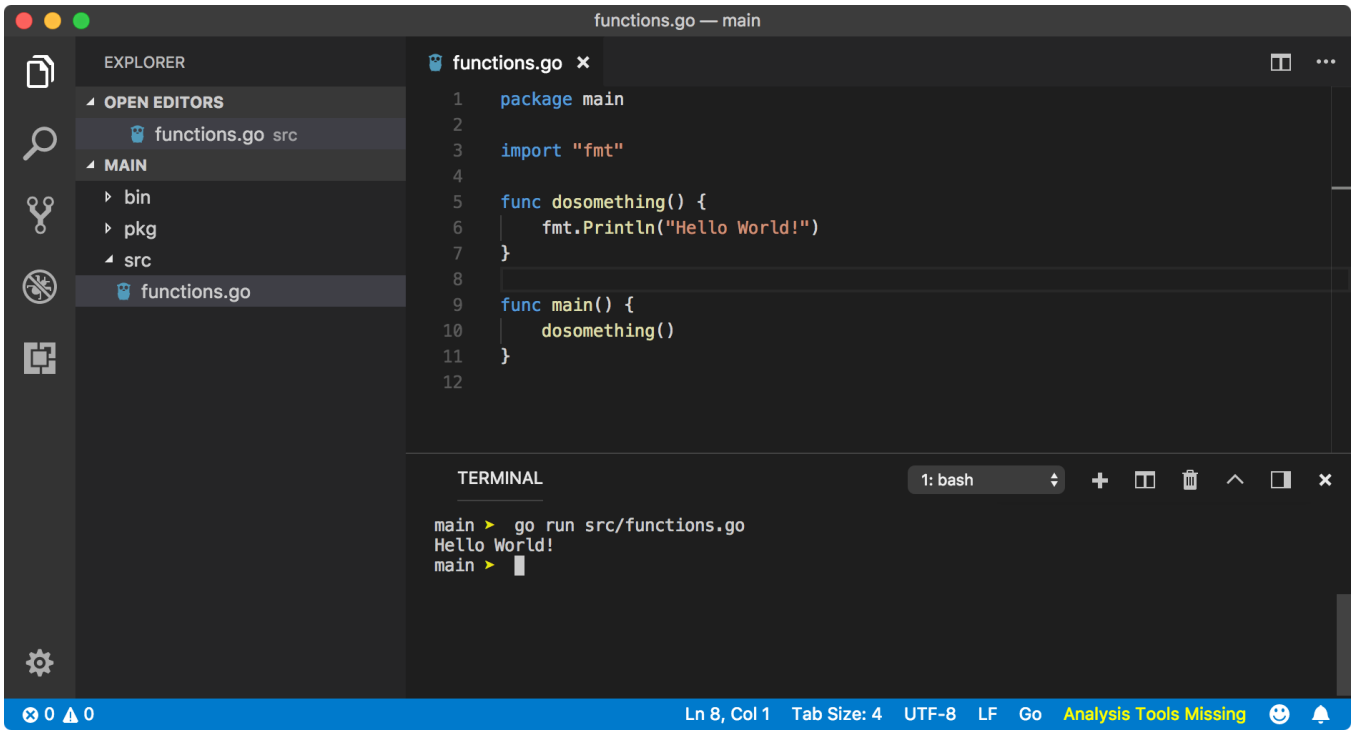
A function, in general, is a small piece of code that is dedicated to a perform particular task based on some input values. We create a function so that we can perform this operation whenever we want from wherever we want by providing some input values. These input values provide extra information to a function and they are totally optional. A function's execution may or may not return any result.

In Go, a function is defined using `func` keyword.

```
func dosomething() {
    fmt.Println("Hello World!")
```

```
}
```

This function can be called from anywhere inside a function body in the program like `dosomething()`. For example,



https://play.golang.org/p/THBF9b1nOr-

☛ function name convention

Go recommends writing function names in `simple` word or `camelCase`.
Even `under_score` function names are valid, they are not idiomatic in Go.

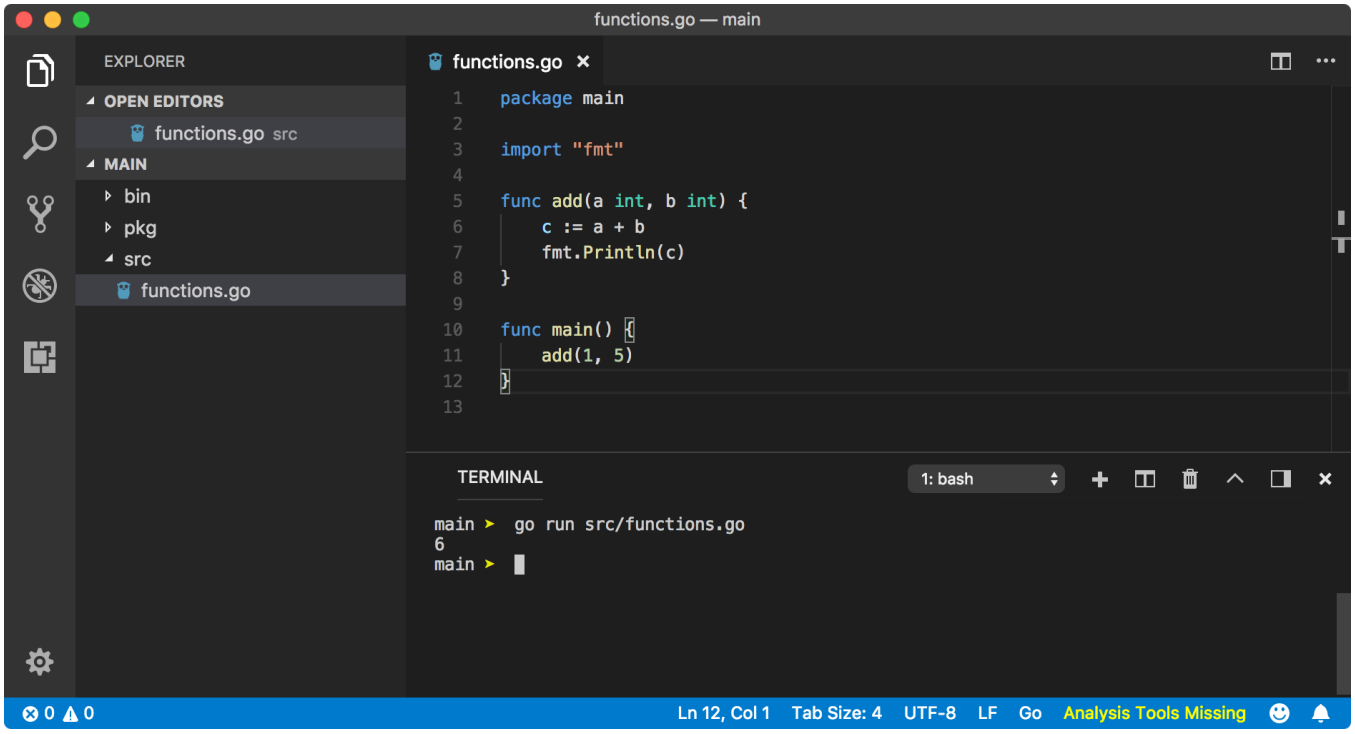
☛ Function parameters

As discussed earlier, a function may take input values for its execution. These input values provided in a function call are called **arguments**. One or multiple arguments can also be passed to a function.

Example 1. Return greeting message

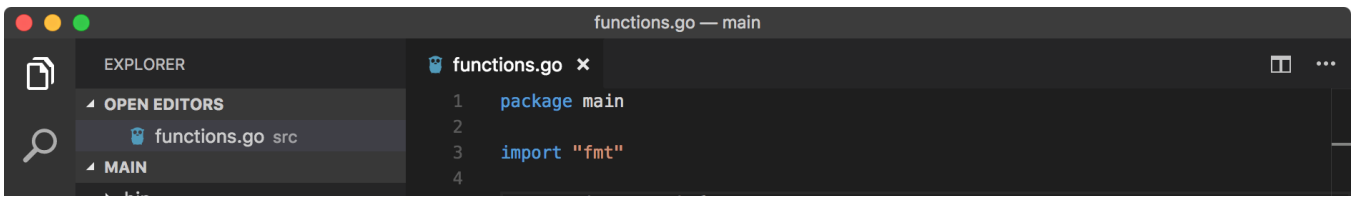
https://play.golang.org/p/E_7DdS_Hw4f

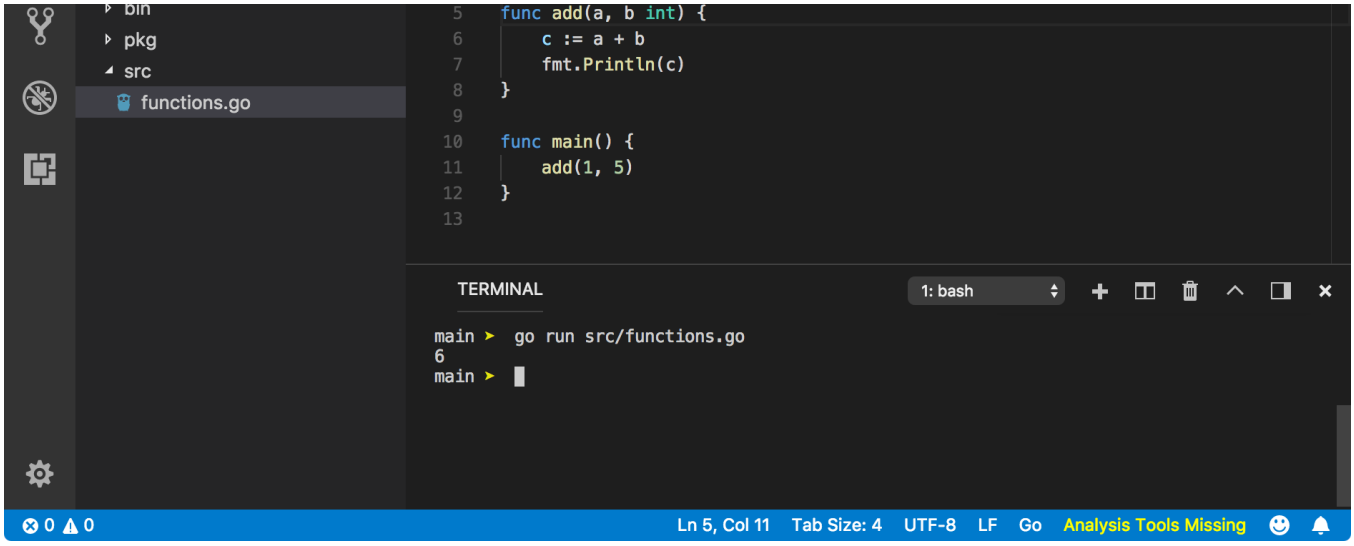
Example 2. Add two integers



https://play.golang.org/p/QUNdCtGO4sA

You can use shorthand parameters notation in case all parameters are of the same data type.

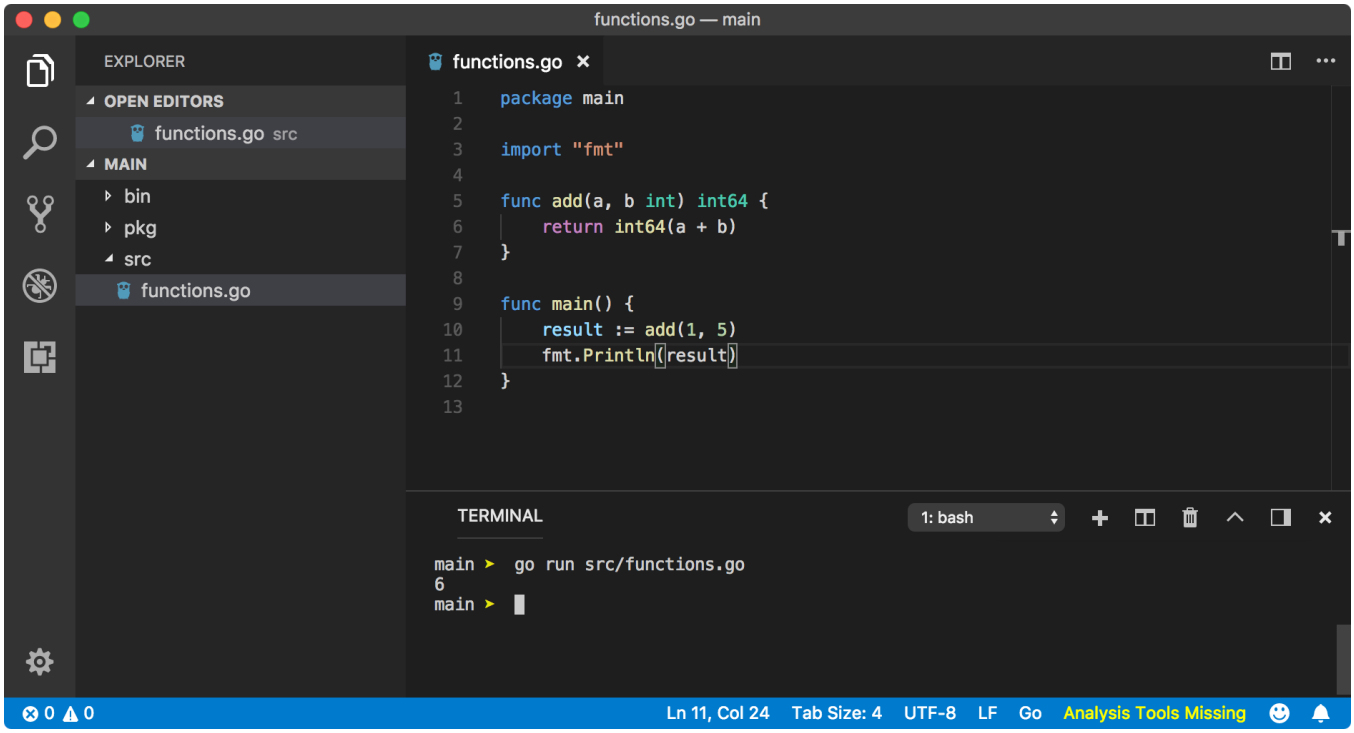




https://play.golang.org/p/j8yvuTv10KK

☛ Return value

A function can also return a value which can be printed or assigned to another variable.



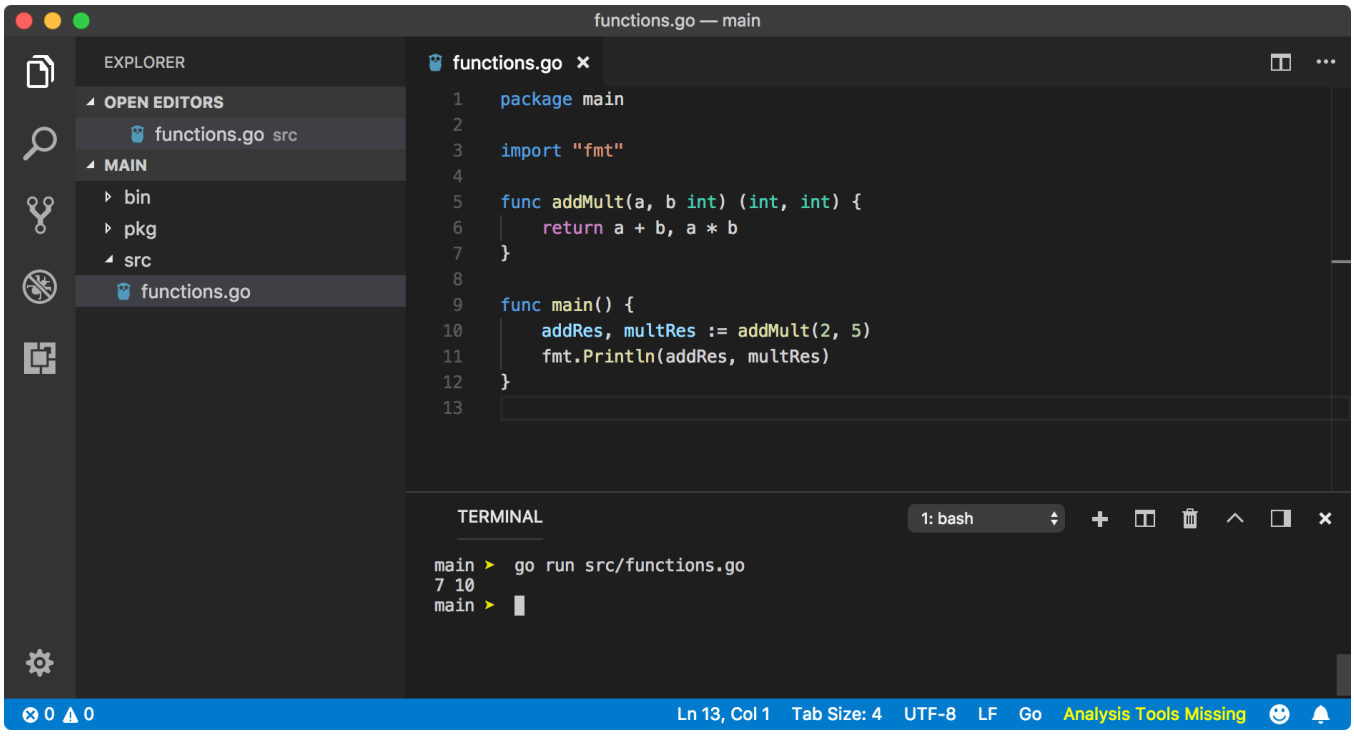
https://play.golang.org/p/b0V3589pbVj

In case a function returns a value, you must specify the data type of a return value just after the function parameter parentheses. In the above program, we made sure that

return value matches the return type of a function by converting the type of result (*originally* `int`) into `int64`.

☛ Multiple return values

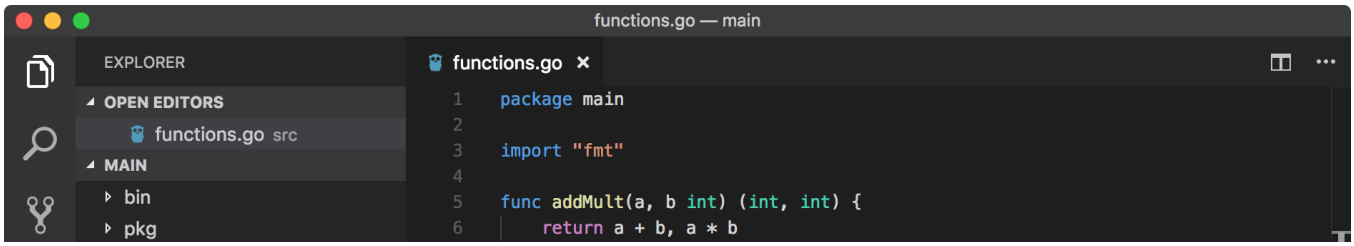
Unlike other programming languages, Go can return multiple values from the function. In this case, we must specify return types of the values (*just like above*) inside parentheses just after the function parameter parentheses.



https://play.golang.org/p/TSSbha-8g9f

To catch multiple values from a function that returns multiple values, we must specify the comma-separated variable declaration.

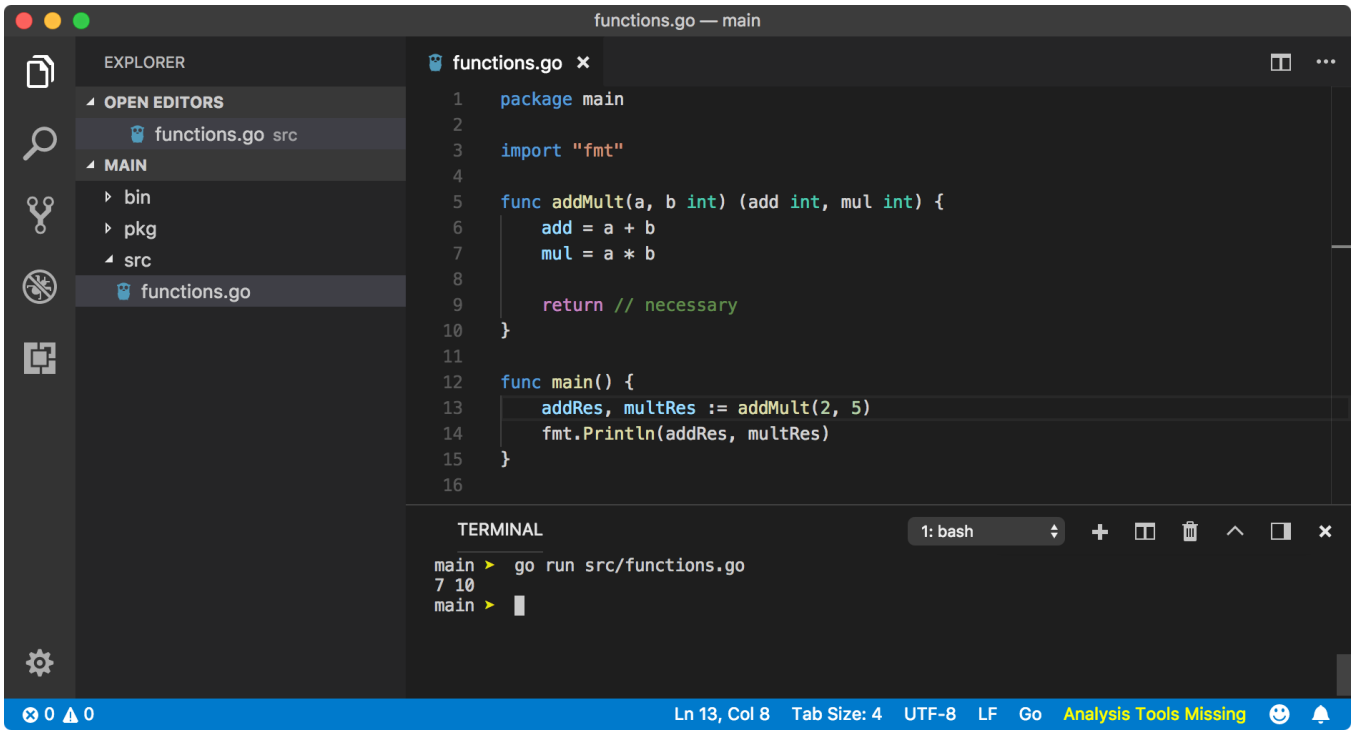
In case of multiple returns values but you are only interested in one single value returned by the function, you can assign other value(s) to `_` (*blank identifier*) which stores the value to an empty variable. This is needed because if a variable is defined but not used in Go, the compiler complains about it.



https://play.golang.org/p/eqLywlCdAAW

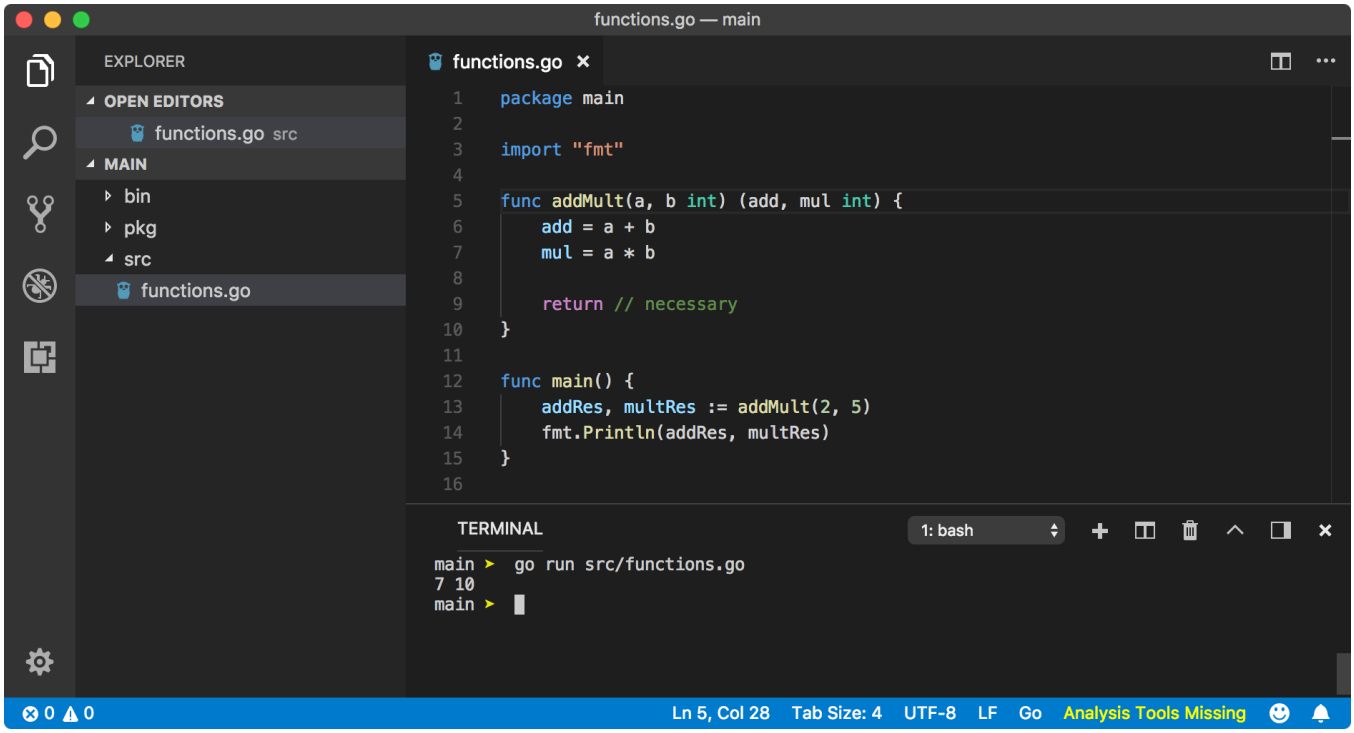
☛ Named return values

Named return values are a great way to **explicitly mention return variables in the function definition itself**. These variables will be created automatically and made available inside the function body. You can change the values of these variables inside a function. A `return` statement at the end of the function is necessary to return named values. Go will automatically return these variables when function hits the return statement.



https://play.golang.org/p/7mjltVetYS8

You can also combine named return values when they hold same data type like below



https://play.golang.org/p/UBtP7nnBD8_c

☛ Recursive function

A function is called recursive when it calls itself from inside the body. A simple syntax for the recursive function is

```
func r() {
    r()
}
```

If we run above function r, it will loop infinitely. Hence, in a recursive function, we generally use a conditional statement such as `if-else` to come out of the infinite loop.

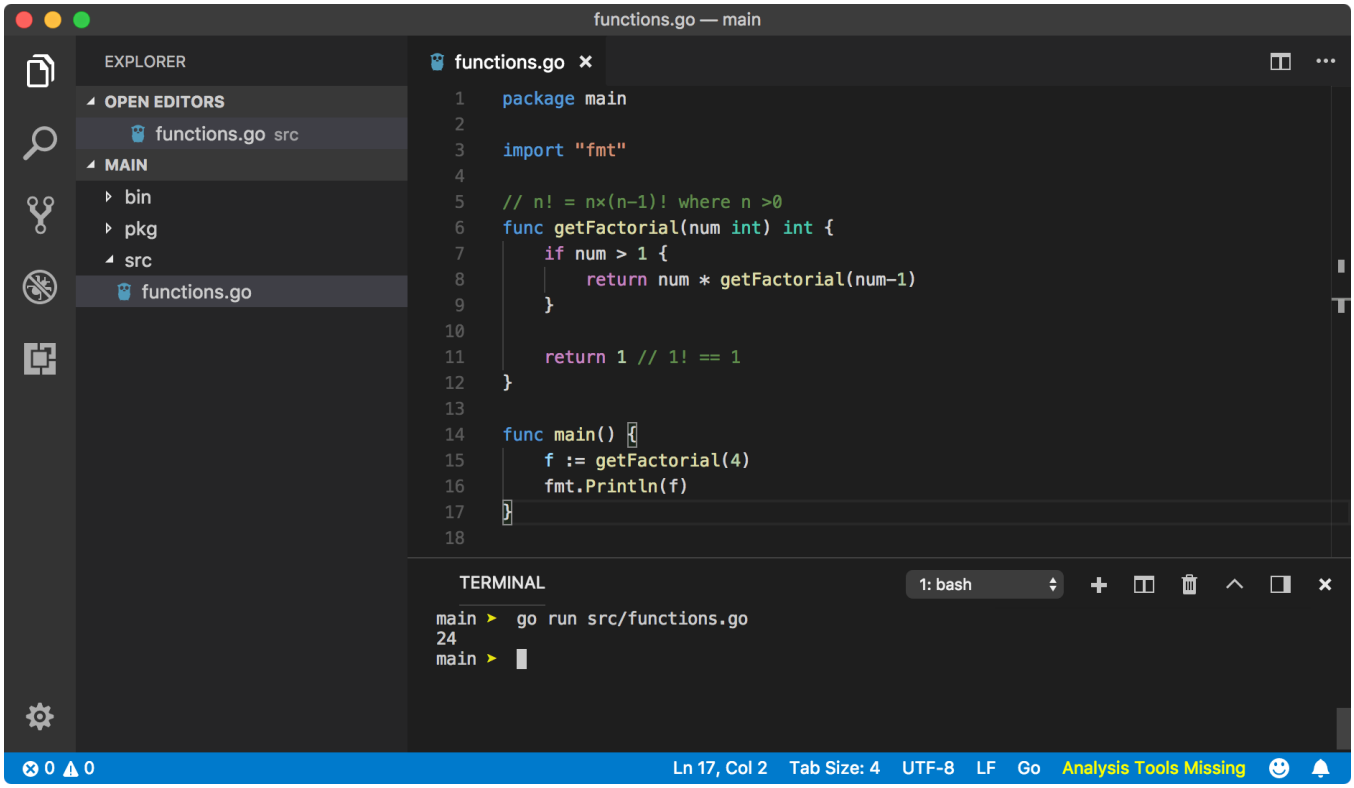
A simple example of a recursive function is `factorial of n`. A simple recursive formula for factorial of `n` is `n * (n-1)` provided `n > 0`.

```
// n! = nx(n-1)! where n >0
func getFactorial(num int) int {
        if num > 1 {
                return num * getFactorial(num-1)
```

```
        } else {
                return 1 // 1! == 1
        }
}
```

Above `getFactorial` function is recursive, as we are calling `getFactorial` from inside `getFactorial` function. The steps to understand are very simple.

When `getFactorial` get called with a `int` parameter `num`, if `num` is equal to `1`, function returns `1`, else it goes inside `if` block and executes `num * getFactorial(num-1)`. Since, there is a function call, function `getFactorial` called again and return value will be kept on hold until `getFactorial` returns something. This stack will be kept on building until `getFactorial` returns something, which is `1` eventually. As soon as that happens, all `call stack` will be resolved one by one eventually resolving first `getFactorial` call.

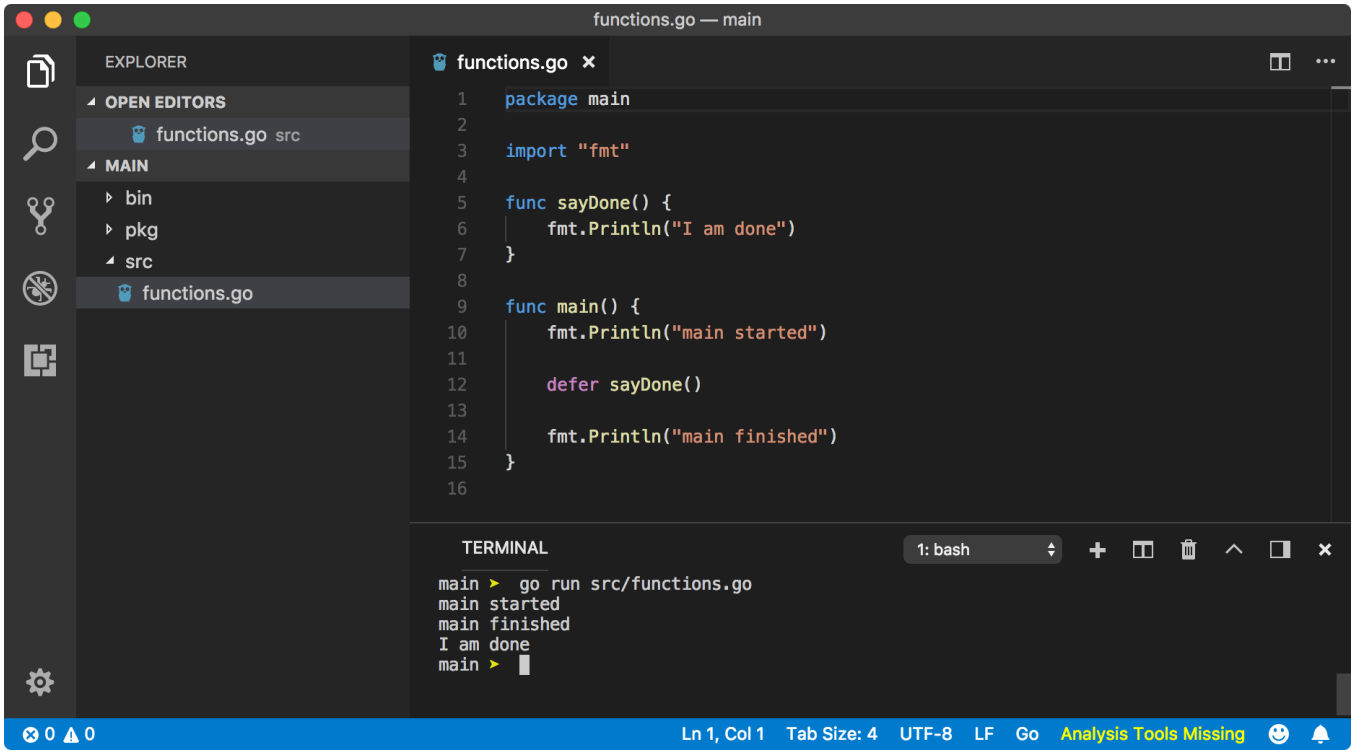


https://play.golang.org/p/xg1_zTFUsd5

☛ `defer` keyword

`defer` is a keyword in Go that makes a function executes at the end of the execution of parent function or when parent function hits `return` statement.

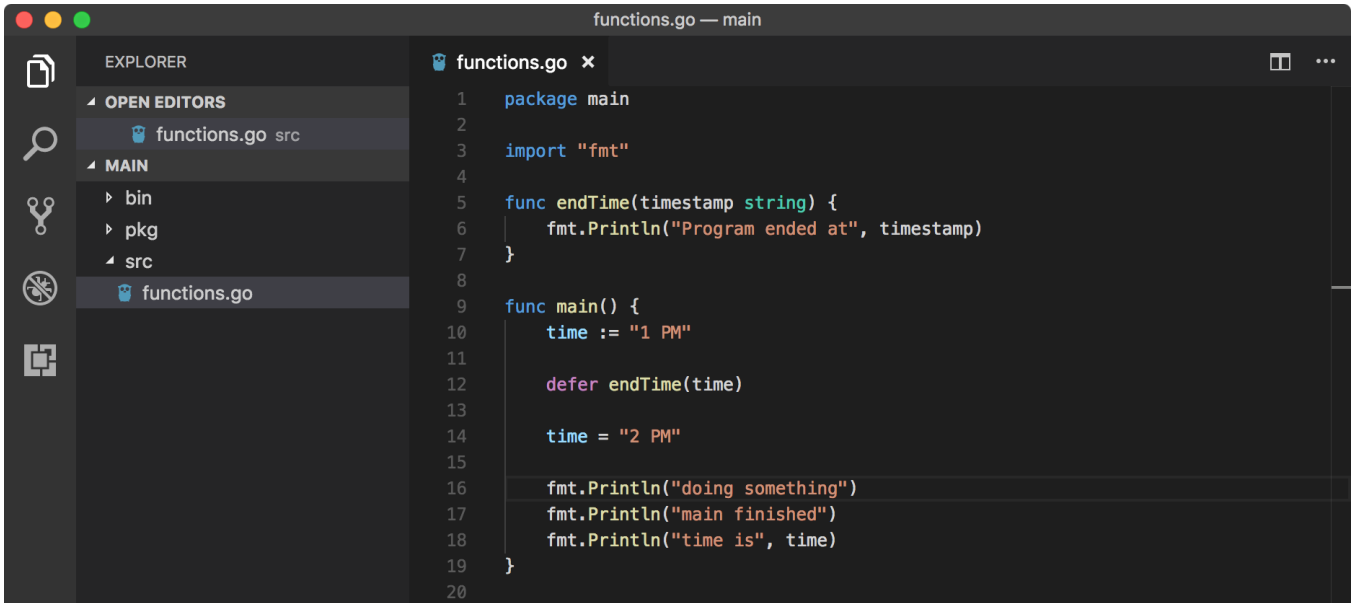
Let's dive into the example to understand better.

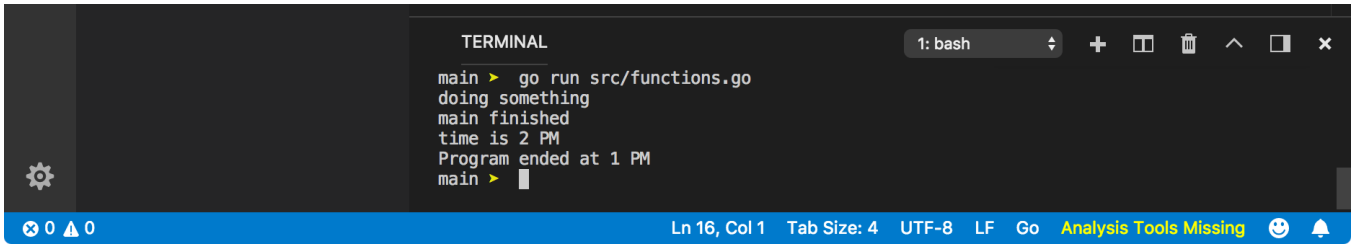


https://play.golang.org/p/x4JfXO3DEng

As `main` function executes, it will print `main started` and then hits `sayDone` but keeps in the waiting list because of `defer` function. Then it prints `main finished` and as `main` function stops executing, `sayDone()` gets executed.

We can pass parameters to defer function if it supports but there is a hidden gotcha. Let's create a simple function with arguments.



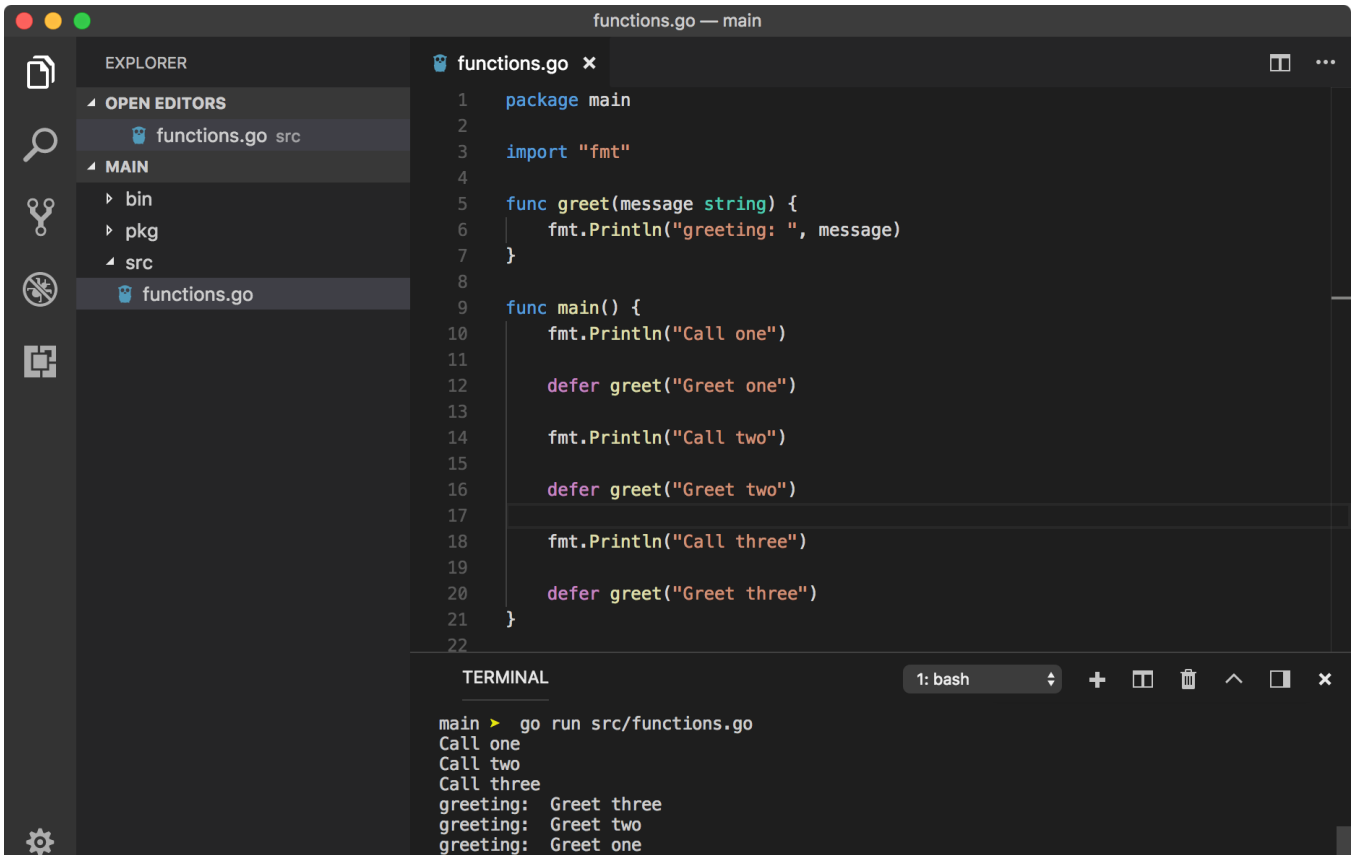


https://play.golang.org/p/bxUskp0MCvo

In the above program, we `deferred` execution of `endTime` function which means it will get executed in the end of `main` function but since at the end `main` function, `time === "2 PM"`, we were expecting `Program ended at 2 PM` message. Even though because of deferred execution, `endTime` function is executing at the end of `main` function, it was pushed into the `stack` with all available argument values earlier when `time` variable was still `1 PM`.

> *You may ask, what is this `stack` I am referring to. A stack is like a notebook where Go compiler writes down `deferred functions` to execute at the end of current function execution. This stack follows `Last In First Out (LIFO)` order of execution. Which means, any task pushed first, will execute in the end.*

Let's write multiple deferred tasks and see what I mean



https://play.golang.org/p/NmeNjRPEmTK

Practical use of `defer` can be seen when a function has too many conditions whether `if-else` or `case` statements, and at the end of each condition, you need to do something like `close a file` or `send http response`. Instead of writing multiple calls, we can use defer to save the day.

Below is an example of a **bad** program.

```
if cond1 {
    ...
    fs.Close(file)
} else if cond2 {
    ...
    fs.Close(file)
} else if cond3 {
    ...
    fs.Close(file)

} else {
    ...
    fs.Close(file)
}
```

Below is an example of a **good** program.

```
defer fs.Close(file)

if cond1 {
    ...
} else if cond2 {
    ...
} else if cond3 {
    ...
} else {
    ...
}
```

☛ Function as type

A function in Go is also a type. If two function accepts the same parameters and returns the same values, then these two functions are of the same type.

For example, `add` and `substract` which individually takes two integers of type `int` and return an integer of type `int` are of the same type.

We have seen some function definitions before, for example, built-in `append` function has a definition like

```
func append(slice []Type, elems ...Type) []Type
```

Hence any function, for example, `prepend` which adds elements at the beginning of the slice (*expandable array*) if has a definition like

```
func prepend(slice []Type, elems ...Type) []Type
```

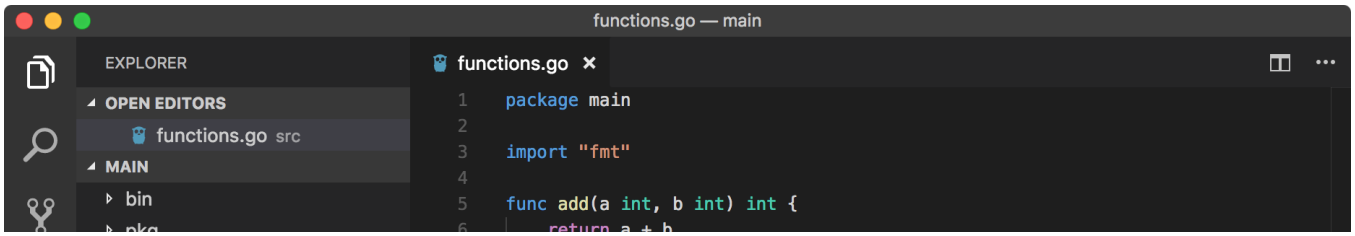
Then `append` and `prepend` will have the same type of

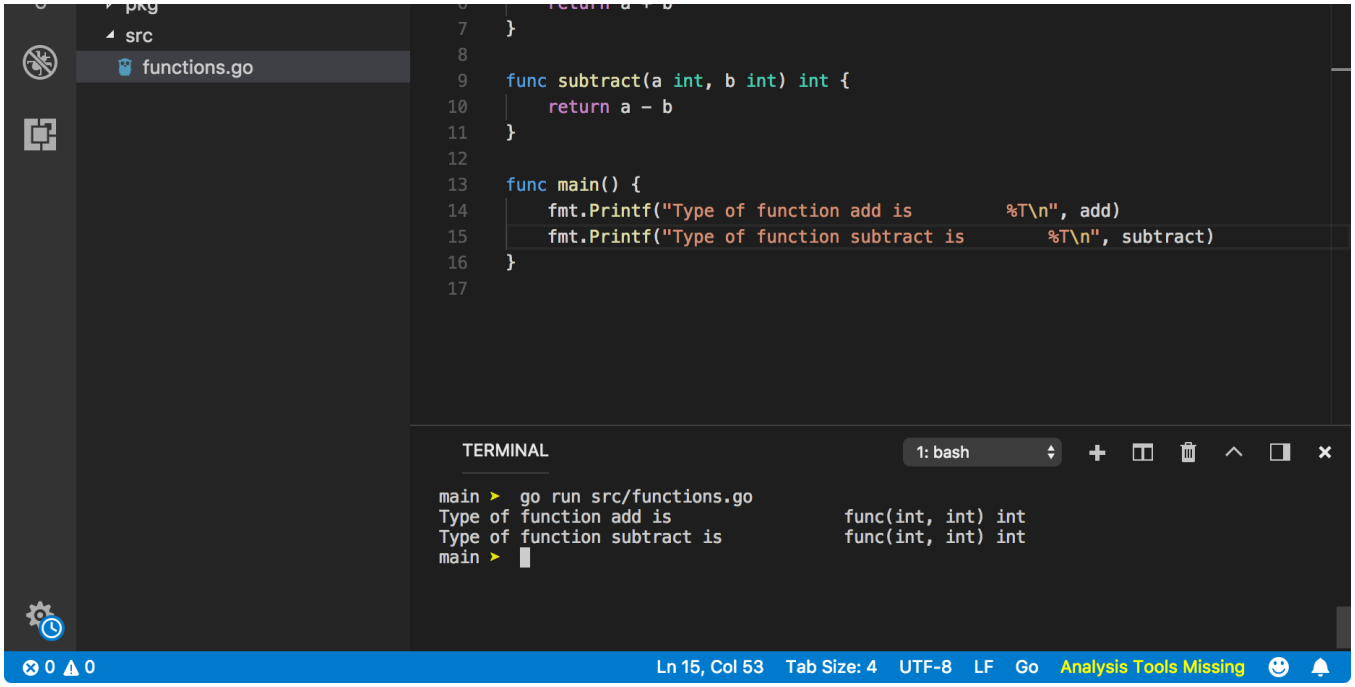
```
func (slice []Type, elems ...Type) []Type
```

So in Go, function body does not have anything to do with the function type. But what is the use of this type?

This can be useful if you are passing a function as an argument to another function or when a function returns another function and you need to give a return type in a function definition.

Let's create a function `add` and `subtract`, and see how they compare.

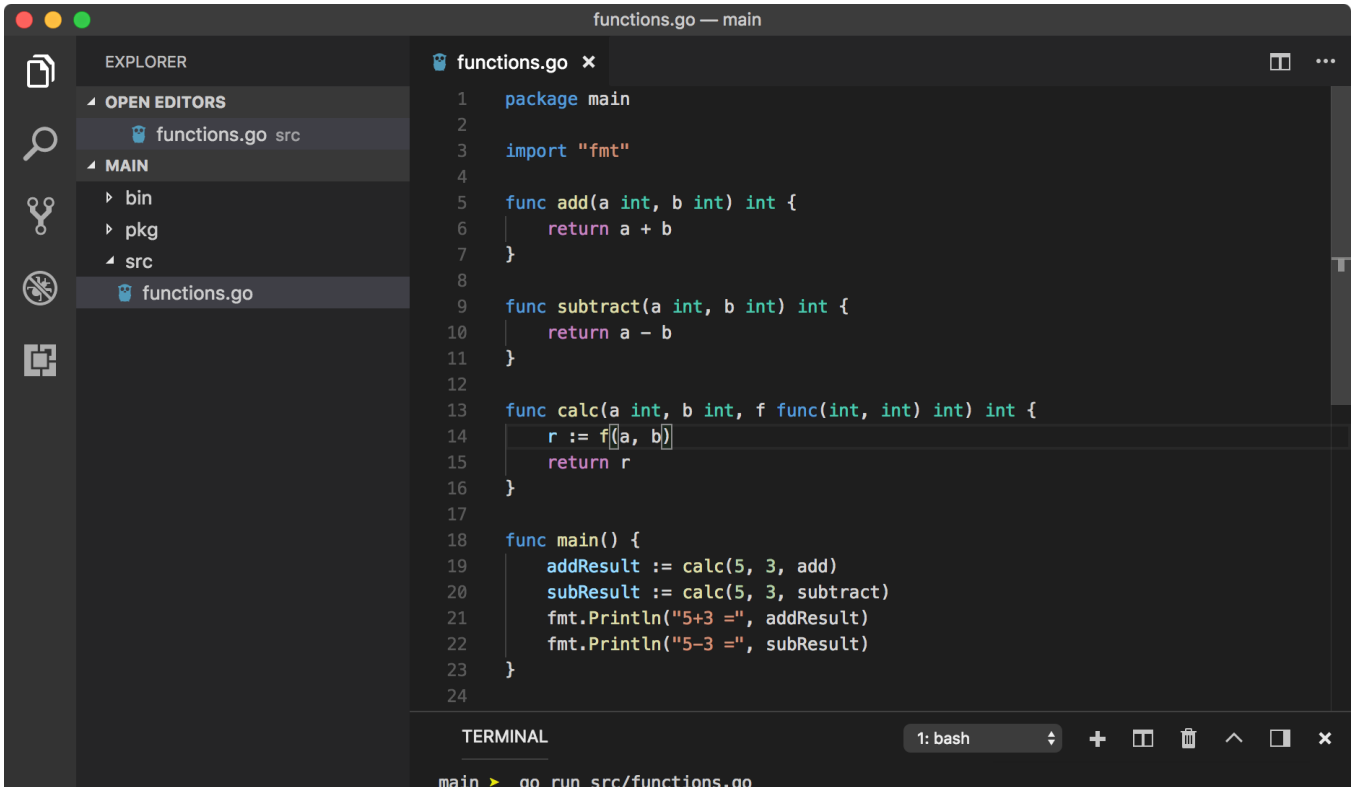


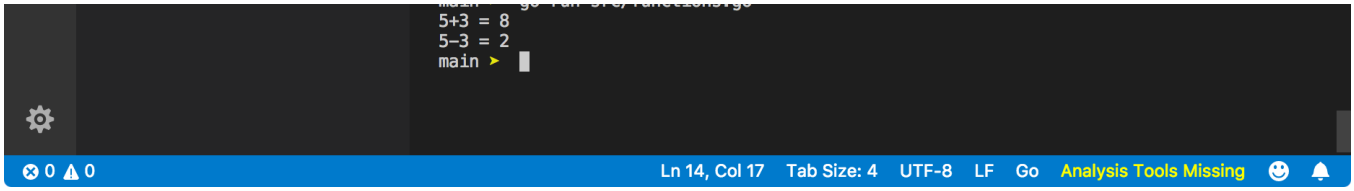


https://play.golang.org/p/LxOPPRvq4Ta

So you can see that both `add` and `subtract` function has the same type `func(int, int) int`.

Let's create a function which takes two integers and third argument a function which does mathematical operation on those two numbers. We will use `add` and `subtract` function as the third parameter to this function.

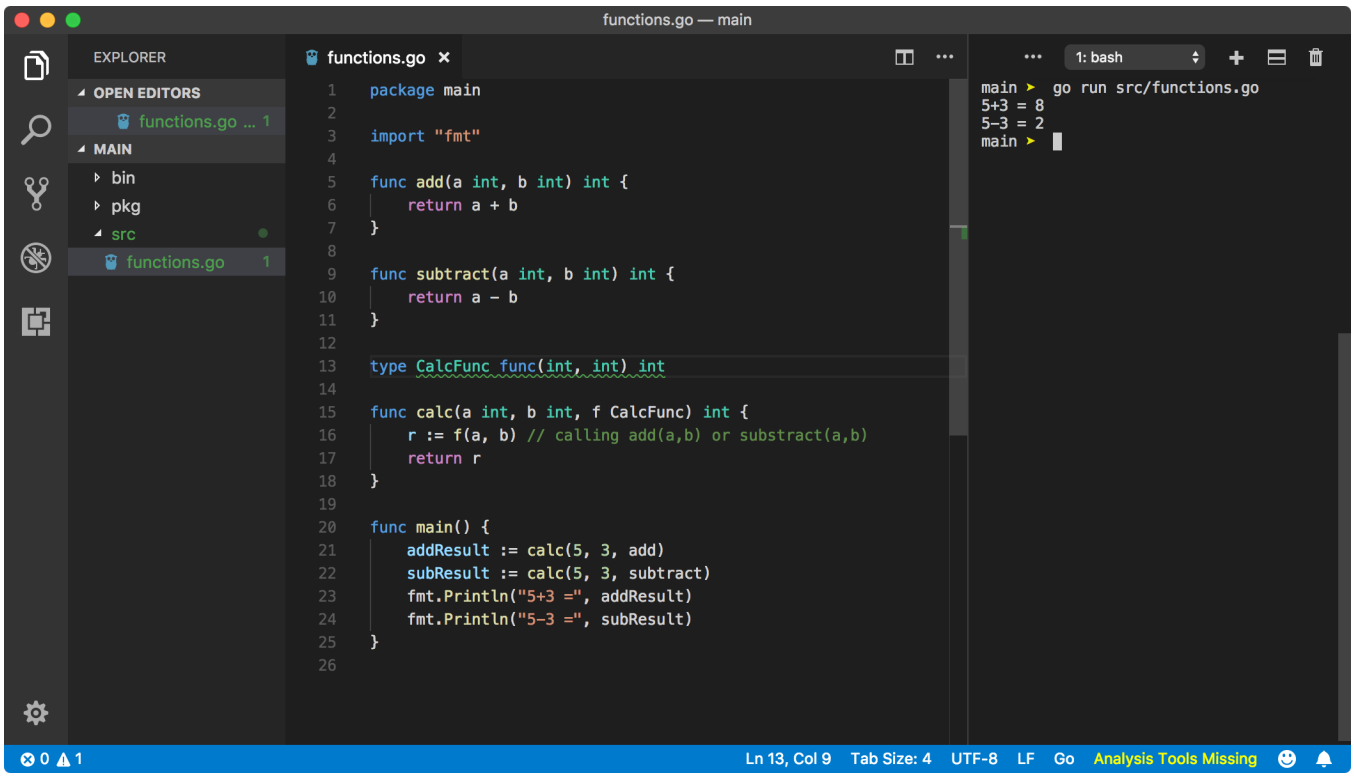




https://play.golang.org/p/z3lwQPAhNLJ

In the above program, we have defined a function `calc` which takes to `int` arguments `a` & `b` and third function argument `f` of type `func(int, int) int`. Then we are calling `f` function with `a` and `b` as arguments.

We can create a **type definition** which will make things simpler. We can rewrite the above program as

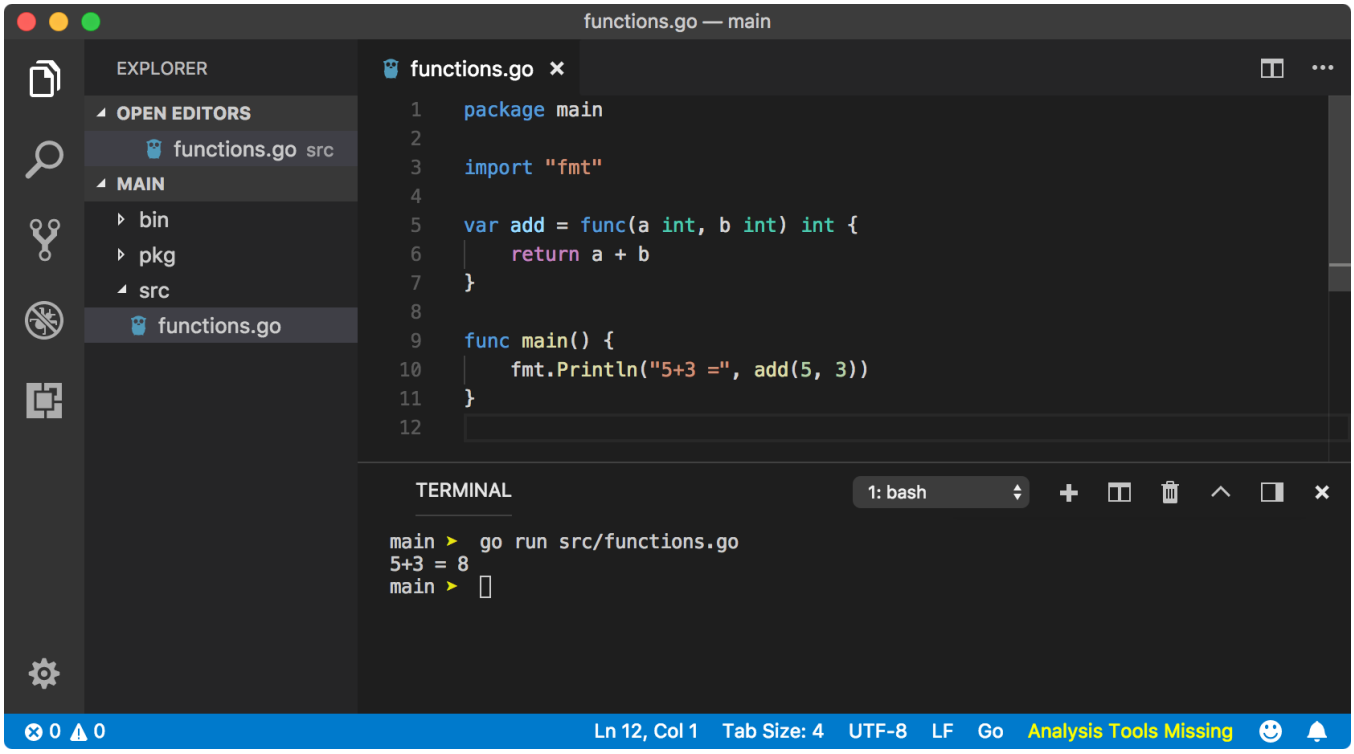


https://play.golang.org/p/RvayW75C8yy

We could also write the above program in a different way where `calc` function instead of taking the third argument as function take `string` command like `add` or `substract` and returns an anonymous function based on the command which we then can execute.

☛ Function as value (anonymous function)

A function in go can also be a value. This means you can assign a function to a variable.



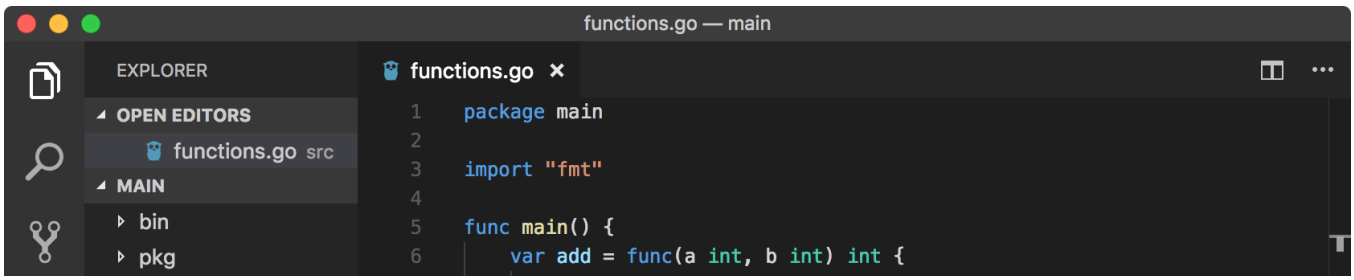
https://play.golang.org/p/UoorB5zCGb2

In the above program, we have created a global variable `add` and assigned a newly created function to it. We have used Go's type inference to get the type of anonymous function (*as we haven't mentioned the type of* `add`). In this case, `add` is an anonymous function as it was created from a function that lacked a name.

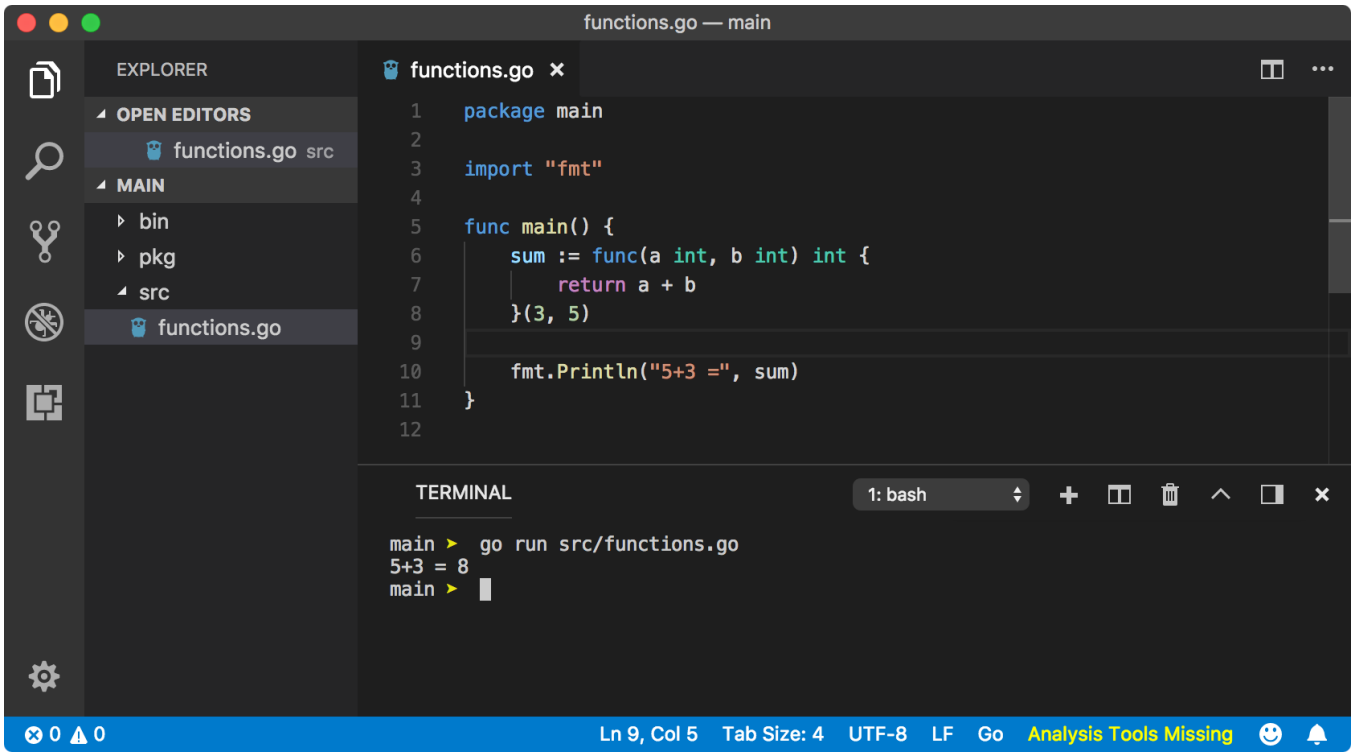
☛ Immediately-invoked function

If you came from `JavaScript` world, you know what **immediately invoked function** is but no worries. In Go, we can create an anonymous function can be **defined and executed at the same time**.

As we have seen an earlier example, an anonymous function defined as



Where, `add` is an anonymous function. Some can argue that it's not truly anonymous because we can still refer to the `add` function from anywhere in `main` function (*in other cases, from anywhere in the program*). But not in the case when a function is immediately invoked or executed. Let's modify the previous example.



https://play.golang.org/p/yxRJr51OxzY

In the above program, look at the function definition. The first part from `func` to `}` defines the function while later `(3, 5)` executes it. Hence `sum` is the value returned by function execution. Hence above program yields the following result

```
5+3 = 8
```

> *Immediately invoked function can also be used outside main function in global context. This can be useful when you need to create a global variable using return value of* ***a function*** *execution which you want to reveal to the world.*