Everything you need to know about Packages in Go

medium.com/rungo/everything-you-need-to-know-about-packages-in-go-b8bac62b74cc

If you are familiar to languages like **Java** or **NodeJS**, then you might be quite familiar with **packages**. A package is nothing but a directory with some code files, which exposes different variables (*features*) from a single point of reference. Let me explain, what that means.

Imagine you have more than a thousand functions which you need constantly while working on any project. Some of these functions have common behavior. For example, `toUpperCase` and `toLowerCase` function transforms **case** of a `string`, so you write them in a single file (*probably **case.go***). There are other functions which does some other operations on `string` data type, so you write them in separate file as well.

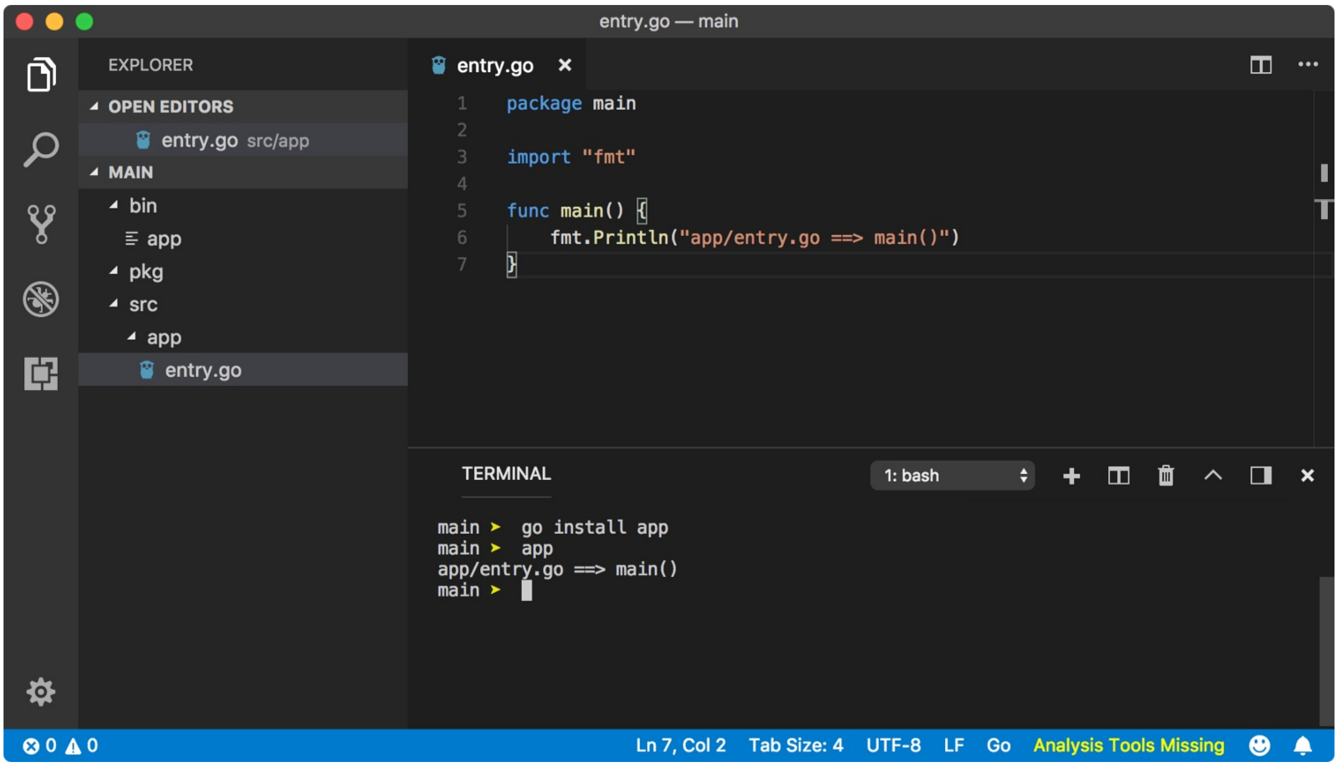
Since you have many files which do something with `string` data type, so you created a directory named `string` and put all `string` related files into it. Finally, you put all of these directories in one parent directory which will be your package. The whole package structure looks like below.

```
package-name
├── string
|  ├── case.go
|  ├── trim.go
|  └── misc.go
└── number
   ├── arithmetics.go
   └── primes.go
```

I will explain thoroughly, how we can import functions and variables from a package and how everything blends together to form a package, but for now, imagine your package as a directory containing `.go` files.

Every Go program **must be** a part of some package. As discussed in **Getting started with Go** lesson, a standalone executable Go program must have `package main` declaration. If a program is part of main package, then `go install` will create a binary file; which upon execution calls `main` function of the program. If a program is part of package other than `main`, then a **package archive** file is created with `go install` command. **Don't worry, I will explain all this in upcoming topics.**

Let's create an executable package. As we know, to create a binary executable file, we need our program to be a part of `main` package and it must have `main` function which is entry point of execution.



```go
package main

import "fmt"

func main() {
	fmt.Println("app/entry.go ==> main()")
}
```

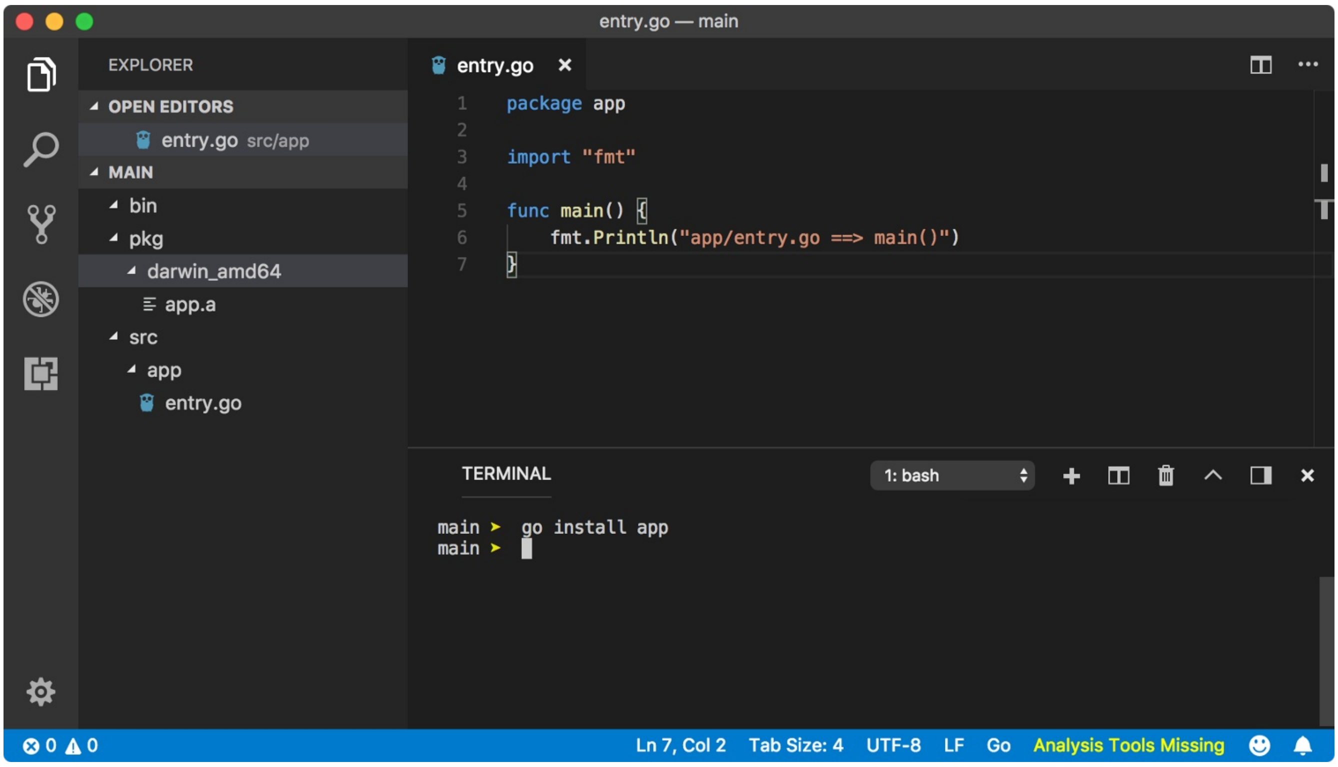
```
main ➤  go install app
main ➤  app
app/entry.go ==> main()
main ➤
```

A package name is name of the directory contained in `src` directory. In above case, `app` is the package since `app` is the child directory of `src` directory. Hence, `go install app` command looked for `app` sub-directory inside `src` directory of `GOPATH`. Then it compiled the package and created `app` binary executable file inside `bin` directory which should be executable from terminal since `bin` directory in the `PATH`.

> **Package declaration** which should be first line of code like `package main` in above example, can be different than package name. Hence, you might find some packages where package name (**name of the directory**) is different than package declaration. When you import a package, package declaration is used to create package reference variable, explained later in the article.

`go install <package>` command looks for any file with `main` **package declaration** inside given `package` directory. If it finds a file, then Go knows this is an executable program and it needs to create a binary file. A package can have many files but only one file with `main` function, since that file will be entry point of the execution.

If a package does not contain file with `main` package declaration, then Go creates a **package archive** (`.a`)file inside `pkg` directory.



```go
package app

import "fmt"

func main() {
    fmt.Println("app/entry.go ==> main()")
}
```

```
main ➤  go install app
main ➤  
```

Since, `app` is not an executable package, it created `app.a` file inside `pkg` directory. We can not execute this file as it's not a binary file.

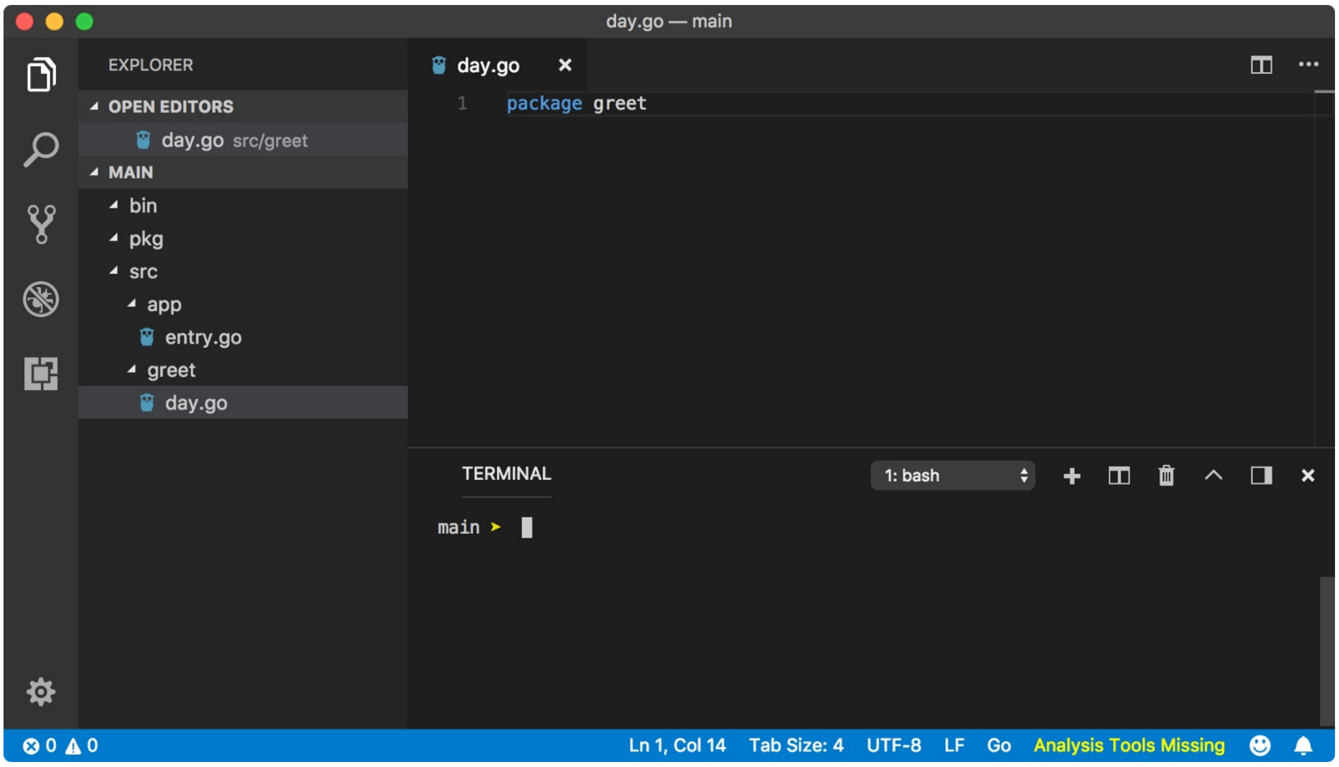
Package naming convention

Go community recommends to use plain and simple names for packages. For example, `strutils` for **string utility** functions or `http` for HTTP requests related functions. A package names with `under_scores`, `hy-phens` or `mixedCaps` should be avoided.

Creating a package

As we discussed, there are two types of packages. An **executable package** and an **utility package**. A executable package is your main application since you will be running it. An utility package is not self executable, instead it enhances functionality of an executable package by providing utility functions and other important assets.

As we know, a package nothing but a directory, let's create `greet` directory inside `src` and crate few files in it. This time, we will write `package greet` declaration on the top of each file to state that this is an utility package.



```
package greet
```

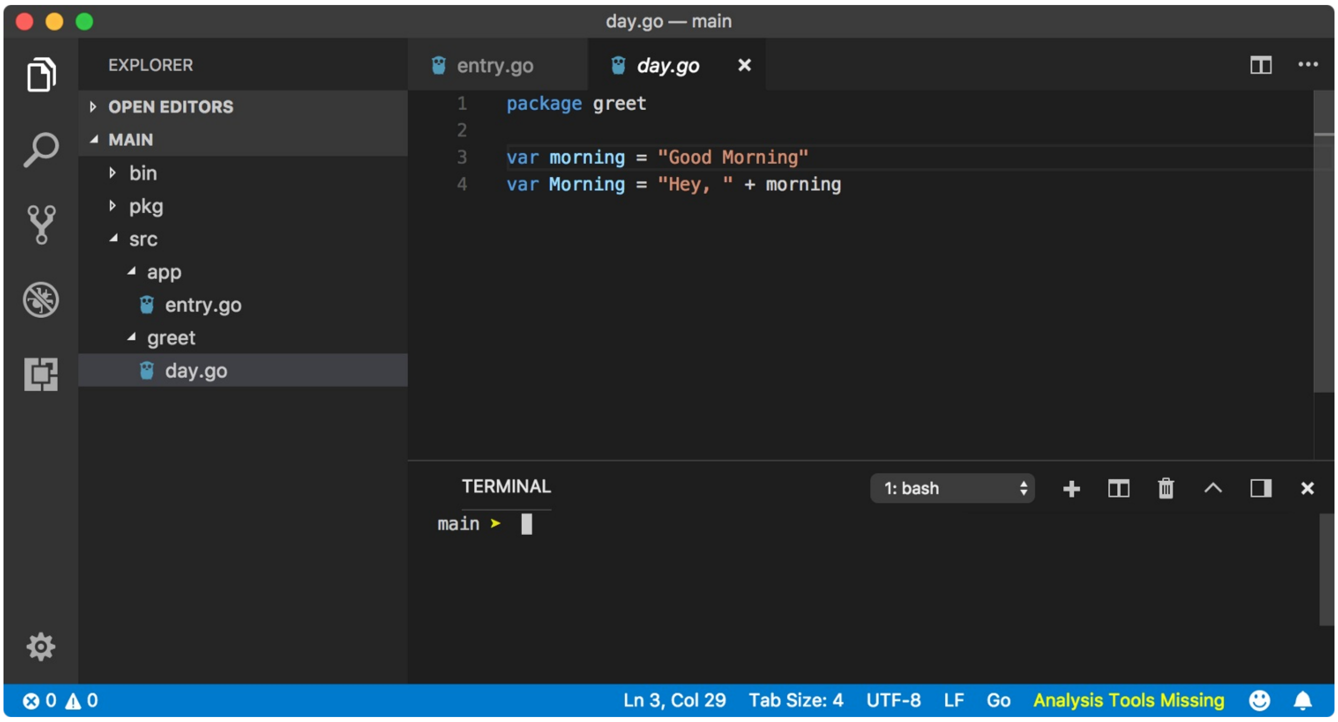
```
main ➤ █
```

Export members

An utility package is supposed to provide some **variables** to a package who imports it. Like `export` syntax in `JavaScript`, Go exports a variable if a variable name starts with **Uppercase**. All other variables not starting with an uppercase letter is private to the package.

> I am going to use **variable** word from now on in this article, to describe an export member but export members can be of any type like *`constant`*, *`map`*, *`function`*, *`struct`*, `array`, `slice` etc.

Let's export a greeting variable from `day.go` file.



```go
package greet

var morning = "Good Morning"
var Morning = "Hey, " + morning
```

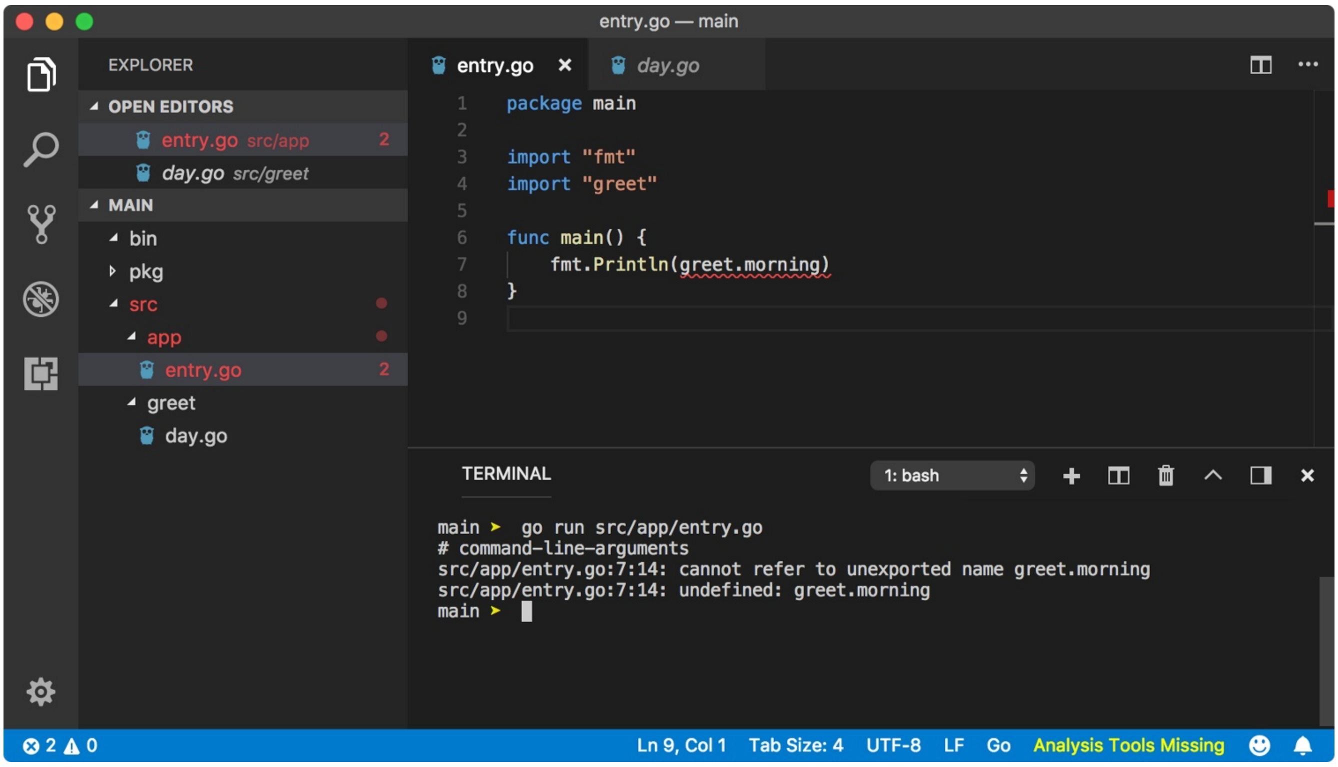
```
main > 
```

In above program, `Morning` variable will be exported from the package but `morning` variable won't be since it starts with lowercase letter.

Importing a package

Now, we need an **executable package** which will consume our `greet` package. Let's create an `app` directory inside `src` and create `entry.go` file with `main` package declaration and `main` function. Note here, Go packages do not have any **entry file naming system** like `index.js` in Node. For an executable package, a file with `main` function is entry file for execution.

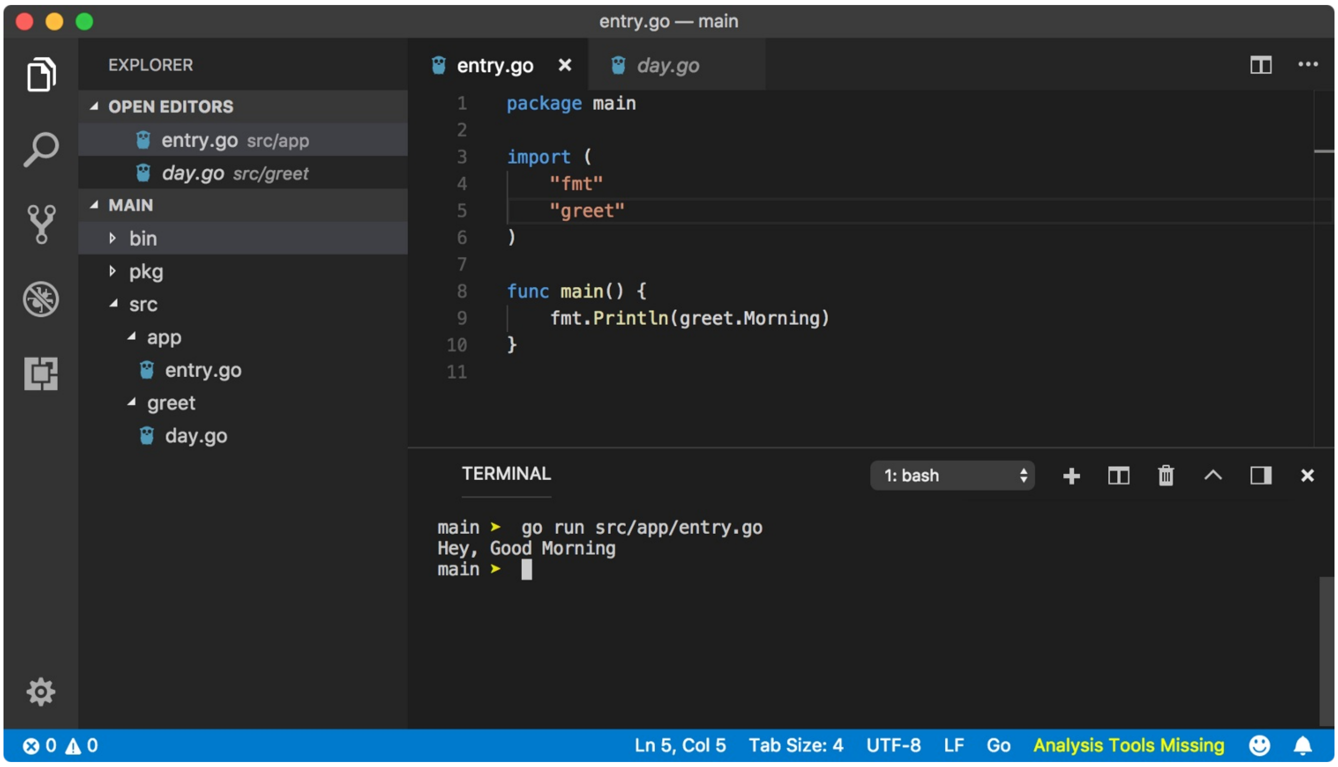
To import a package, we use `import` syntax followed by package name. Go first searches for package directory inside **GOROOT**/src directory and if it doesn't find the package, then it looks for **GOPATH**/src. Since, `fmt` package is part of Go's standard library which is located in `GOROOT/src`, it is imported from there. Since Go can not find `greet` package inside `GOROOT`, it will lookup inside `GOPATH/src` and we have it there.



```go
package main

import "fmt"
import "greet"

func main() {
    fmt.Println(greet.morning)
}
```

```
main ➤  go run src/app/entry.go
# command-line-arguments
src/app/entry.go:7:14: cannot refer to unexported name greet.morning
src/app/entry.go:7:14: undefined: greet.morning
main ➤
```

Above program throws compilation error, as `morning` variable is not visible from package `greet`. As you can see, we use `.` (dot) notation to access exported members from a package. When you import a package, Go creates a global variable using **package declaration** of the package. In above case, `greet` is the global variable created by Go because we used `package greet` declaration in programs contained in `greet` package.



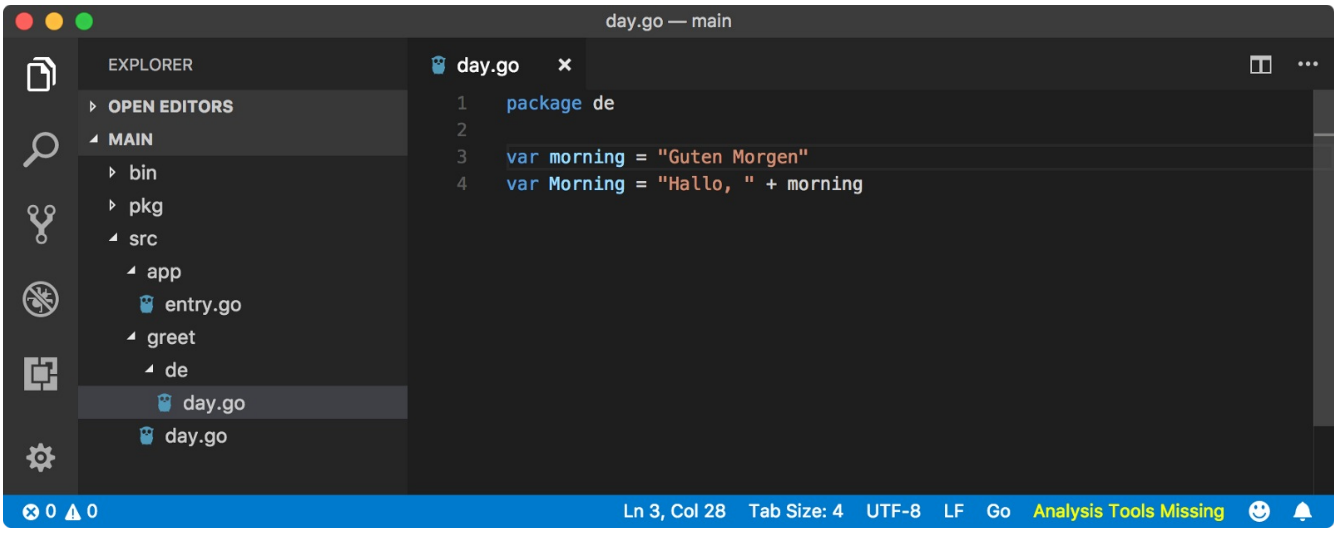
```go
package main

import (
    "fmt"
    "greet"
)

func main() {
    fmt.Println(greet.Morning)
}
```

```
main ➤  go run src/app/entry.go
Hey, Good Morning
main ➤
```

We can group `fmt` and `greet` package imports together using grouping syntax (*parentheses*). This time, our program will compile just fine, because `Morning` variable is available from outside the package.

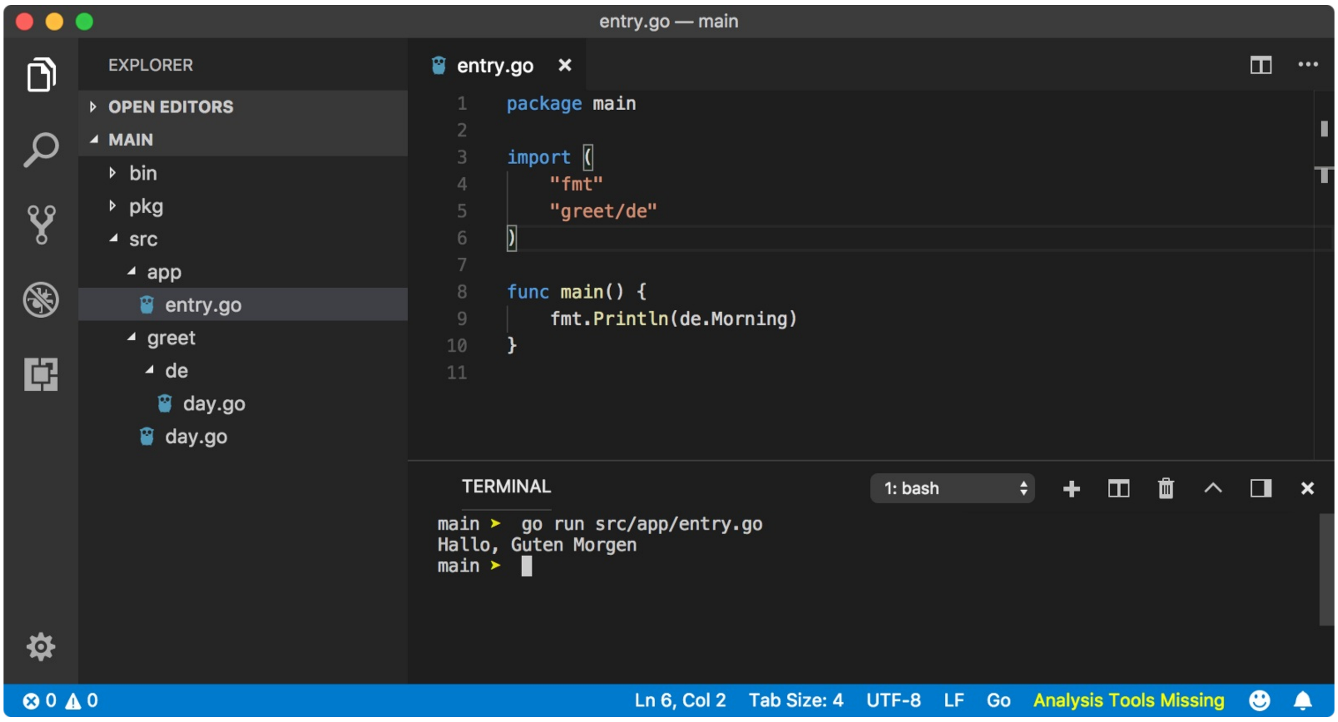
Nested package

We can nest a package inside a package. Since for Go, a package is just a directory, it's like creating a sub directory inside an already existing package. All we have to do is provide relative path of nested package.



day.go — main

```go
package de

var morning = "Guten Morgen"
var Morning = "Hallo, " + morning
```



entry.go — main

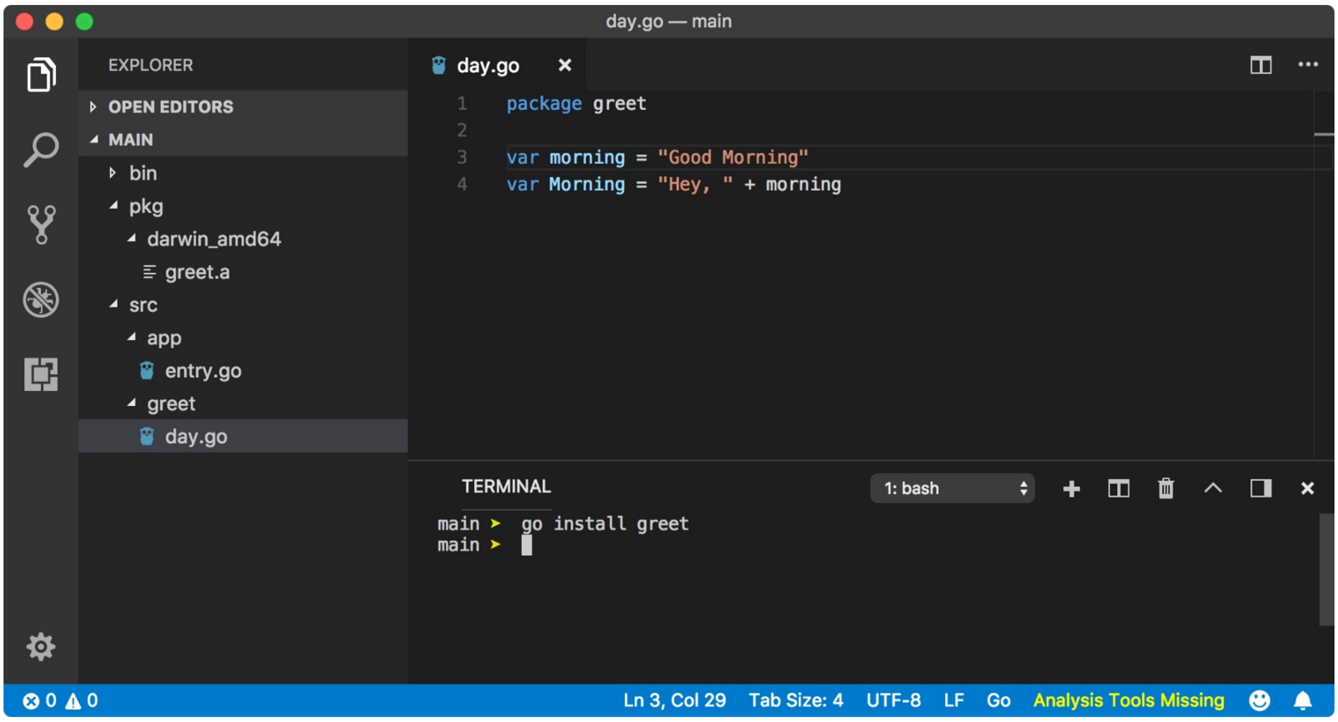
```go
package main

import (
    "fmt"
    "greet/de"
)

func main() {
    fmt.Println(de.Morning)
}
```

```
main ➤ go run src/app/entry.go
Hallo, Guten Morgen
main ➤
```

Package compilation

As discussed in previous lesson, `go run` command compiles and executes a program. We know, `go install` command compiles packages and creates binary executable files or package archive files. This is to avoid compilation of package(s) every single time a (*program where these packages are imported*) is compiled . `go install` pre-compiles a package and Go refers to `.a` files.

> Generally, when you install a 3rd party package, Go compiles the package and create package archive file. If you have written package locally, then your **IDE** might create package archive as soon as you save the file in the package or when package gets modified. **VSCode compiles the package when you save it if you have Go plugin installed.**



```go
package greet

var morning = "Good Morning"
var Morning = "Hey, " + morning
```

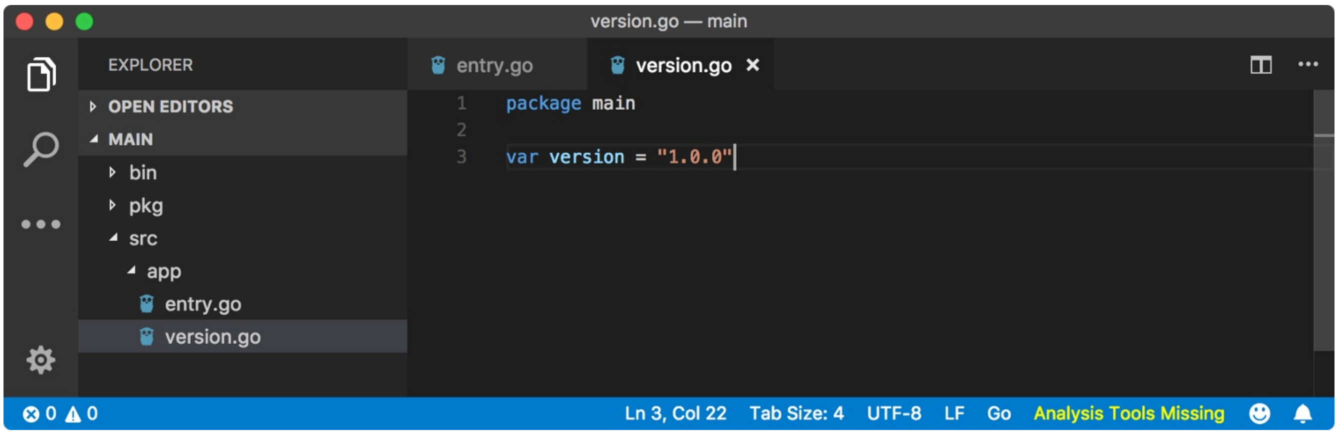
```
main ➤  go install greet
main ➤
```

Package initialization

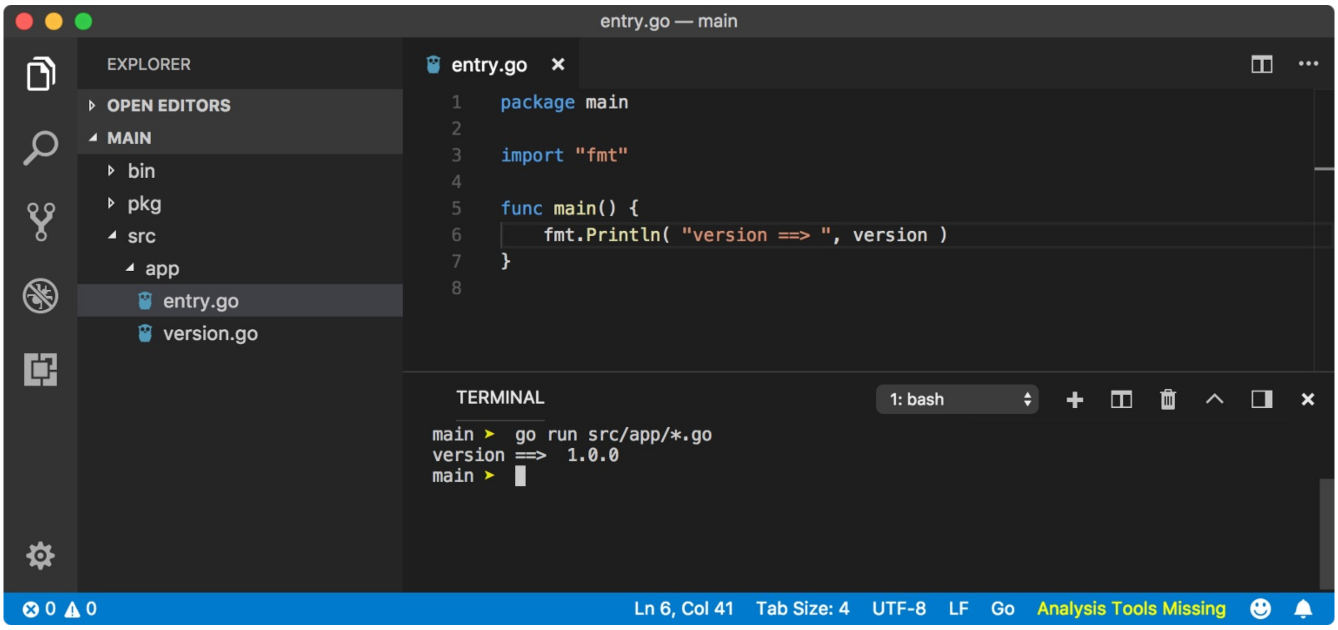
When we run a Go program, Go compiler follows a certain execution order for packages, files in a package and variable declaration in the package.

Package scope

A scope is a region in code block where a defined variable is accessible. A package scope is a region within a package where a declared variable is accessible from within a package (*across all files in the package*). This region is the top-most block of any file in the package.



```go
package main

var version = "1.0.0"
```



```go
package main

import "fmt"

func main() {
    fmt.Println( "version ==> ", version )
}

```

```
main ➤  go run src/app/*.go
version ==>  1.0.0
main ➤
```

Take a look at `go run` command. This time, instead of executing one file, we gave a glob pattern to include all files inside `app` package for execution. Go is smart enough to figure out an entry point of the application which is `entry.go` because it has `main` function. We can also use command like below (*file name order doesn't matter*).

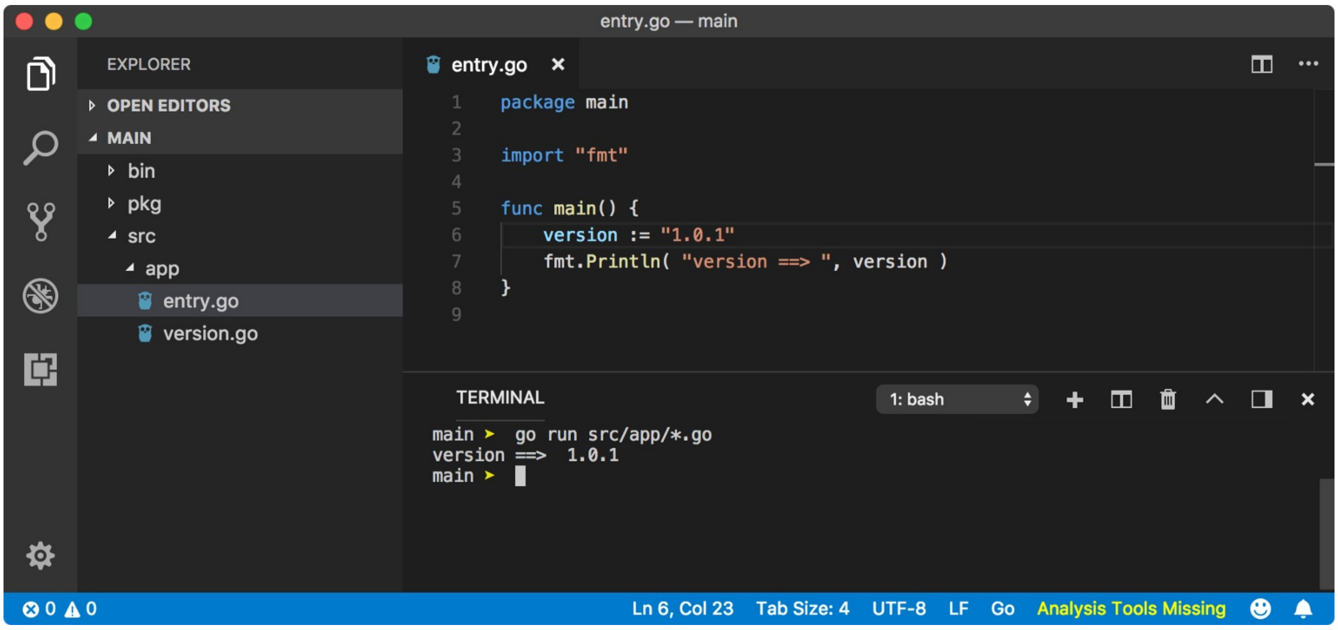
```
go run src/app/version.go src/app/entry.go
```

> `go install` or `go build` command requires a package name, which includes all the files inside a package, so we don't have to specify them like above.

Coming back to our main issue, we can use variable `version` declared in `version.go` file from anywhere in the package even through it is not exported (*like* `Version`), because it is declared in package scope. If `version` variable would have been declared in a function, it wouldn't have been in package scope and above program would have failed to compile.

You are not allowed to re-declare global variable with same name in

the same package. Hence, once `version` variable is declared, it can not be re-declared in the package scope. But you are free to re-declare elsewhere.

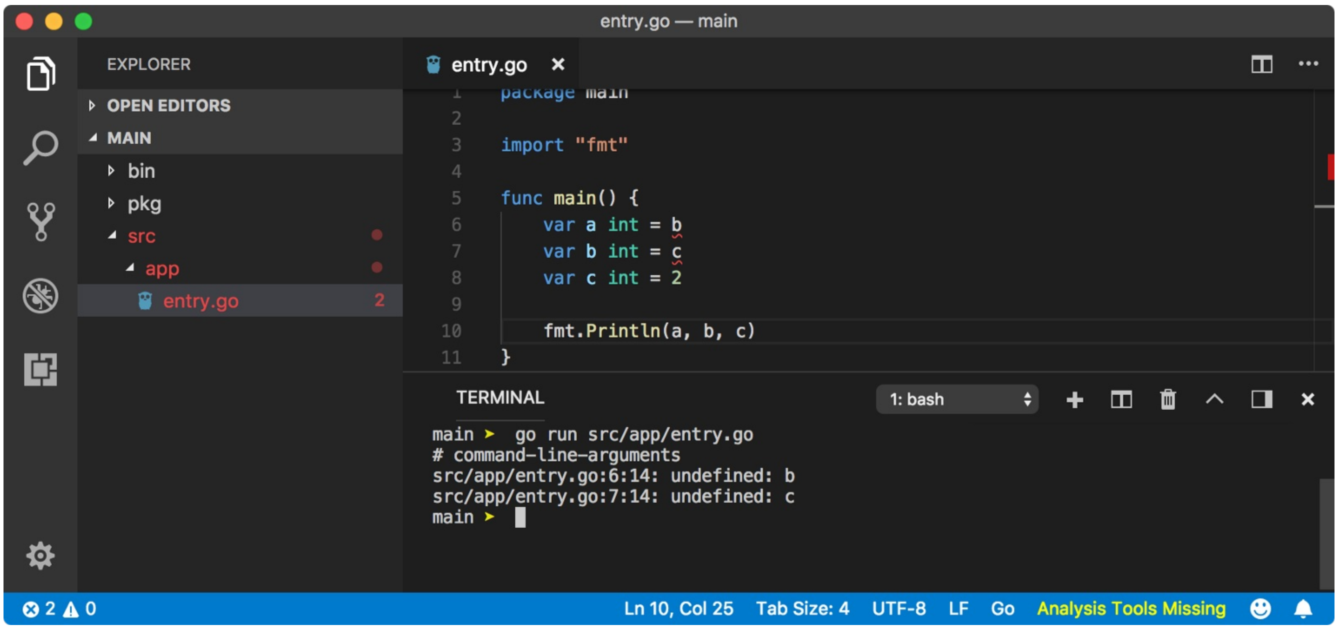


```
package main

import "fmt"

func main() {
    version := "1.0.1"
    fmt.Println( "version ==> ", version )
}
```

```
main ➤ go run src/app/*.go
version ==> 1.0.1
main ➤
```

Variable initialization

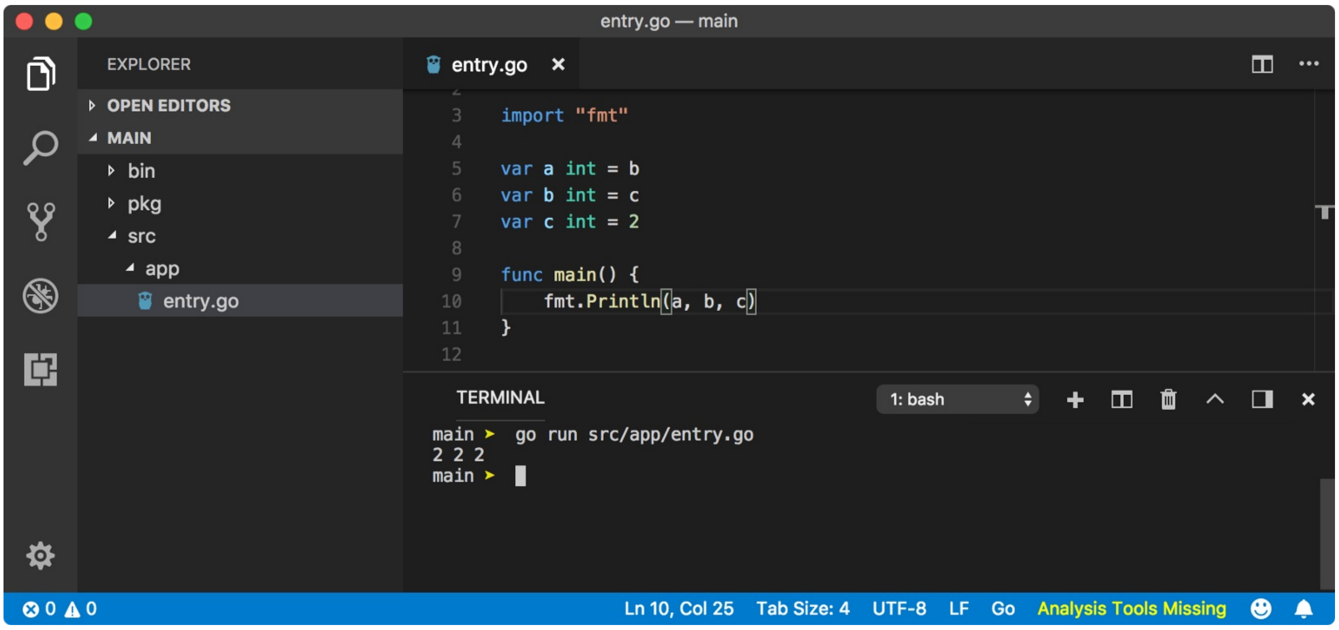
When a variable `a` depends on another variable `b`, `b` should be defined beforehand, else program won't compile. Go follows this rule inside functions.



```
package main

import "fmt"

func main() {
    var a int = b
    var b int = c
    var c int = 2

    fmt.Println(a, b, c)
}
```

```
main ➤ go run src/app/entry.go
# command-line-arguments
src/app/entry.go:6:14: undefined: b
src/app/entry.go:7:14: undefined: c
main ➤
```

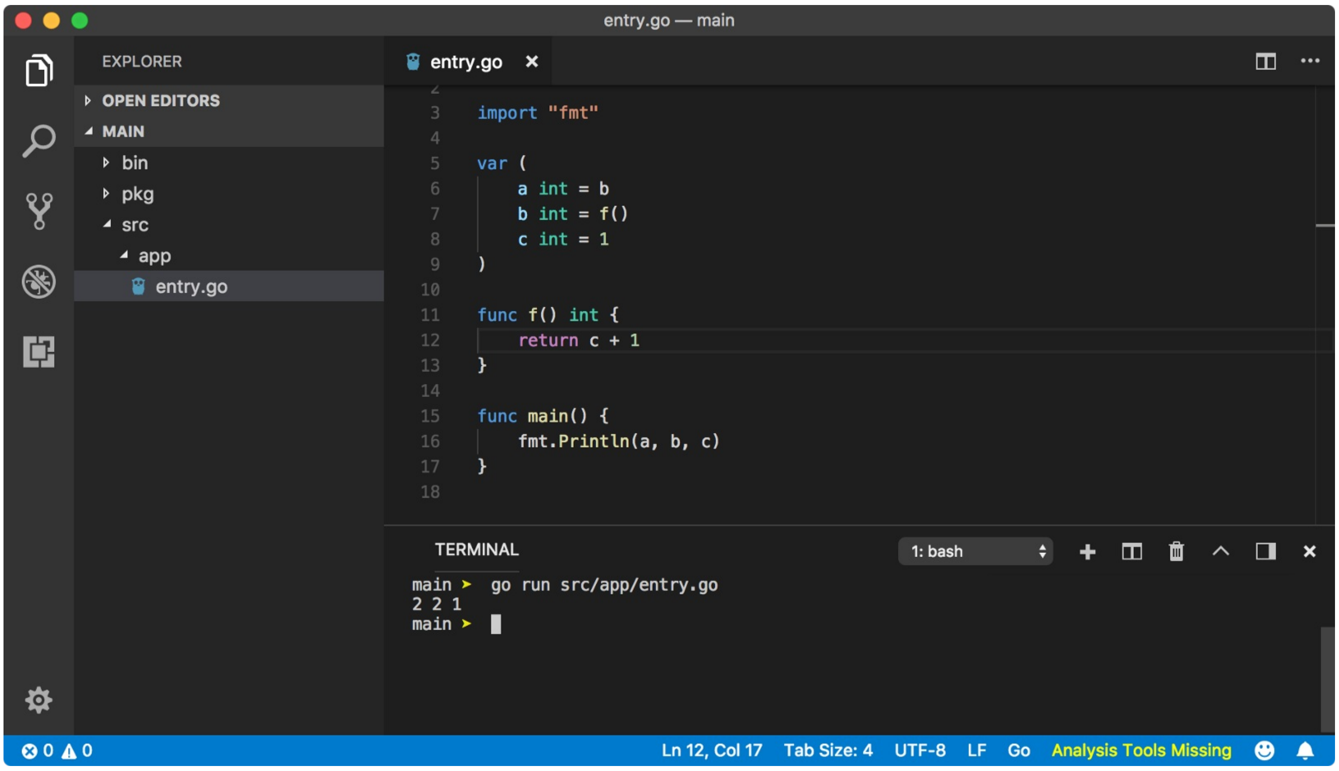
But when these variables are defined in package scope, they are declared in initialization cycles. Let's have a look at simple example below.



```go
import "fmt"

var a int = b
var b int = c
var c int = 2

func main() {
	fmt.Println(a, b, c)
}
```

```
main ➤  go run src/app/entry.go
2 2 2
main ➤
```

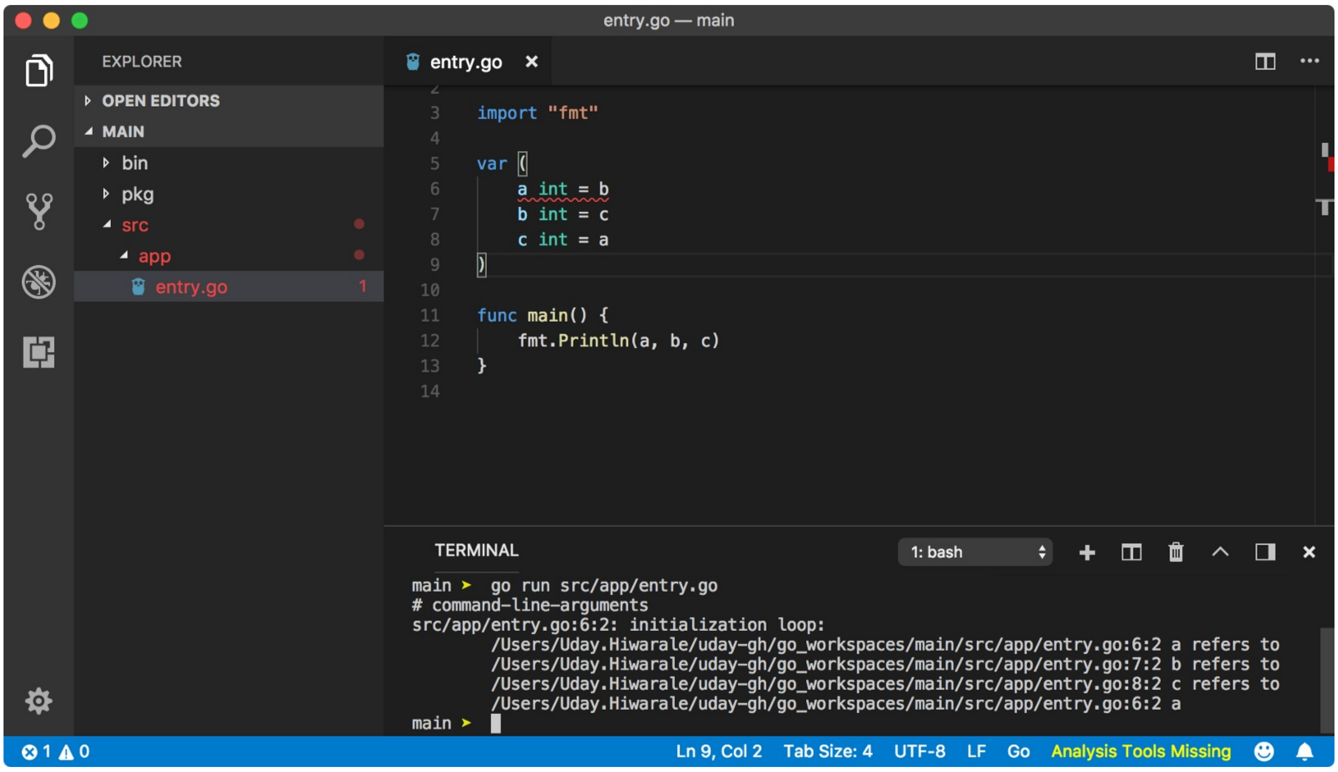
In above example, first `c` is declared because its value is already declared. In later initialization cycle, `b` is declared, because it depends on `c` and value of `c` is already declared. In final initialization cycle, `a` is declared and assigned to the value of `b`. Go can handle complex initialization cycles like below.



```go
import "fmt"

var (
	a int = b
	b int = f()
	c int = 1
)

func f() int {
	return c + 1
}

func main() {
	fmt.Println(a, b, c)
}
```

```
main ➤  go run src/app/entry.go
2 2 1
main ➤
```

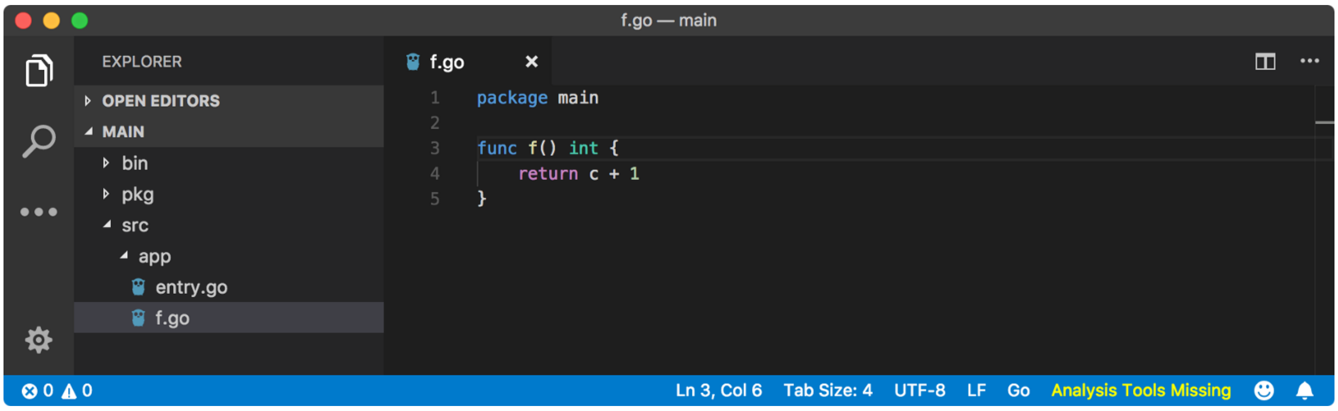
In above example, first `c` will be declared and then `b` as it's value depends on `c` and finally `a` as its value depends on `b`. You should avoid any initialization loop like below where initialization get's into a recursive loop.



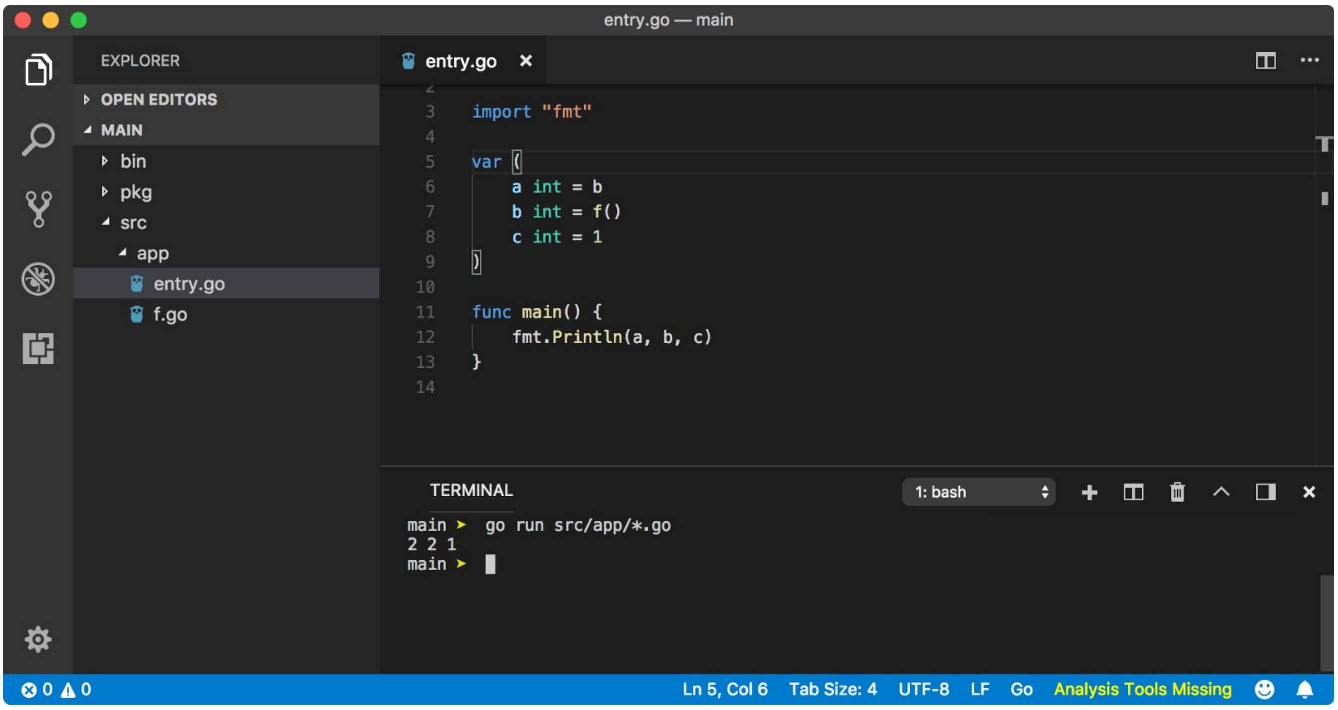
```go
import "fmt"

var (
    a int = b
    b int = c
    c int = a
)

func main() {
    fmt.Println(a, b, c)
}
```

```
main ➤  go run src/app/entry.go
# command-line-arguments
src/app/entry.go:6:2: initialization loop:
        /Users/Uday.Hiwarale/uday-gh/go_workspaces/main/src/app/entry.go:6:2 a refers to
        /Users/Uday.Hiwarale/uday-gh/go_workspaces/main/src/app/entry.go:7:2 b refers to
        /Users/Uday.Hiwarale/uday-gh/go_workspaces/main/src/app/entry.go:8:2 c refers to
        /Users/Uday.Hiwarale/uday-gh/go_workspaces/main/src/app/entry.go:6:2 a
main ➤
```

Another example of package scope would be, having function `f` in separate file which references variable `c` from main file.



```go
package main

func f() int {
    return c + 1
}
```

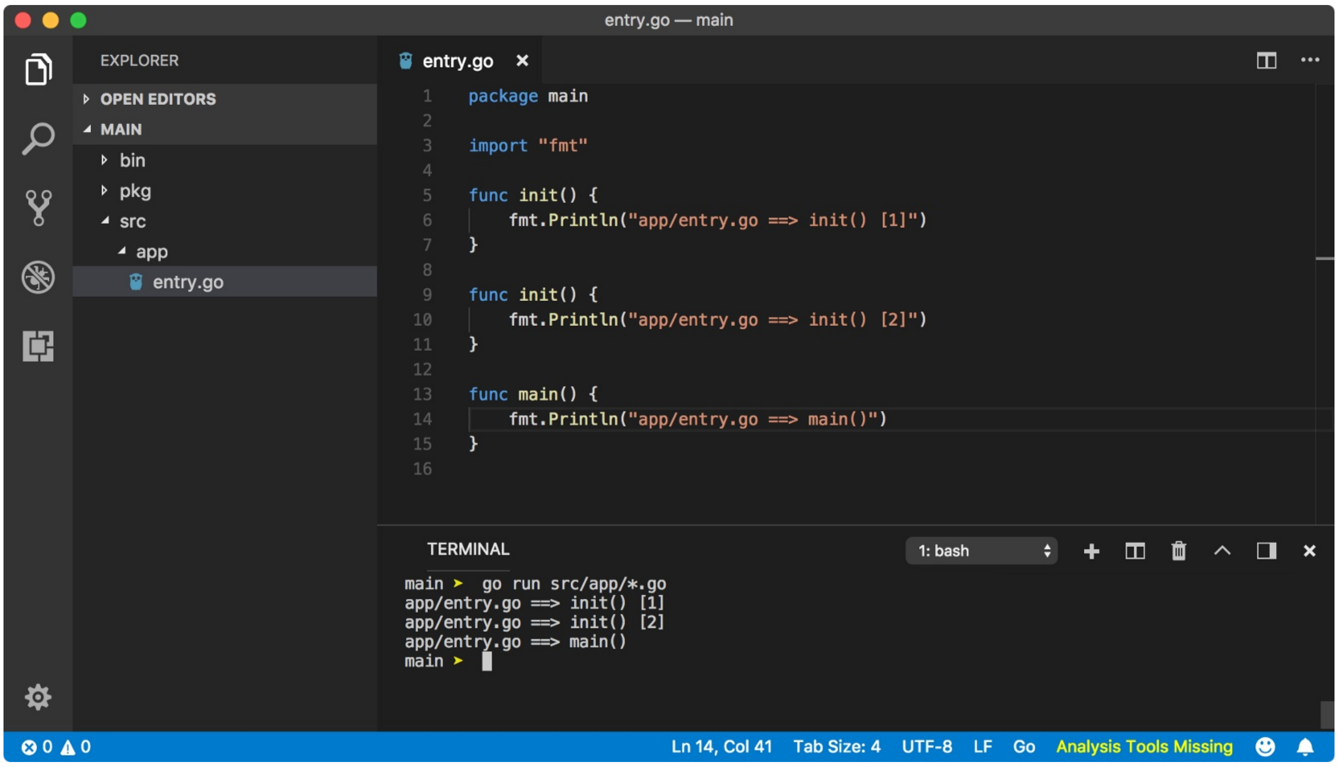


```go
import "fmt"

var (
    a int = b
    b int = f()
    c int = 1
)

func main() {
    fmt.Println(a, b, c)
}
```

```
main ➤  go run src/app/*.go
2 2 1
main ➤
```

Init function

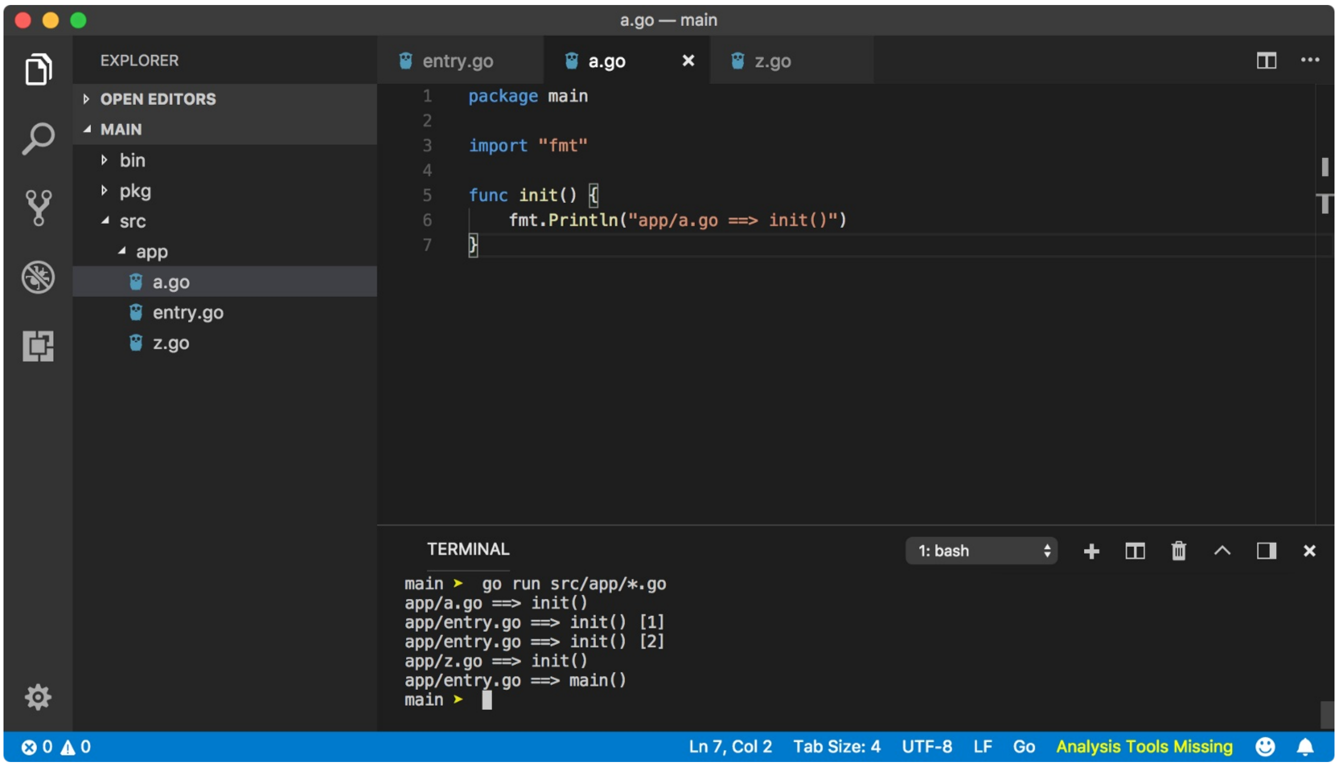
Like `main` function, `init` function is called by Go when a package is initialized. It does not take any arguments and doesn't return any value. `init` function is implicitly declared by Go, hence you can not reference it from anywhere (*or call it like* *`init()`*). You can have multiple `init` functions in a file or a package. Order of the execution of `init` function in a file will be according to the order of their appearances.



```go
package main

import "fmt"

func init() {
    fmt.Println("app/entry.go ==> init() [1]")
}

func init() {
    fmt.Println("app/entry.go ==> init() [2]")
}

func main() {
    fmt.Println("app/entry.go ==> main()")
}
```

```
main ➤  go run src/app/*.go
app/entry.go ==> init() [1]
app/entry.go ==> init() [2]
app/entry.go ==> main()
main ➤
```

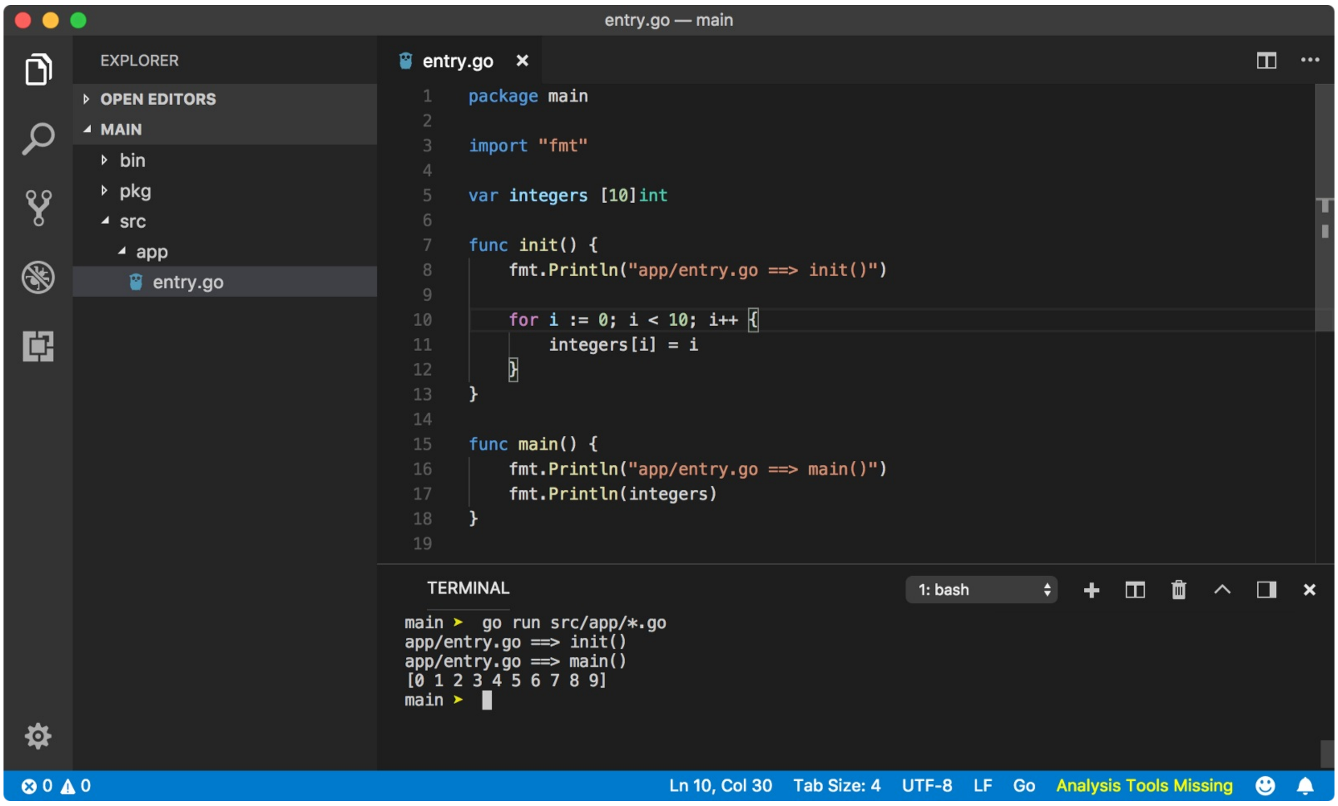
You can have `init` function anywhere in the package. These `init` functions are called in lexical file name order (***alphabetical order***).



```go
package main

import "fmt"

func init() {
    fmt.Println("app/a.go ==> init()")
}
```

```
main ➤  go run src/app/*.go
app/a.go ==> init()
app/entry.go ==> init() [1]
app/entry.go ==> init() [2]
app/z.go ==> init()
app/entry.go ==> main()
main ➤
```

After all `init` functions are executed, `main` function is called. Hence, **main job of `init` function is to initialize global variables** that can not be initialized in global context. For example, initialization of an array.



```go
package main

import "fmt"

var integers [10]int

func init() {
    fmt.Println("app/entry.go ==> init()")

    for i := 0; i < 10; i++ {
        integers[i] = i
    }
}

func main() {
    fmt.Println("app/entry.go ==> main()")
    fmt.Println(integers)
}

```

```
main ➤  go run src/app/*.go
app/entry.go ==> init()
app/entry.go ==> main()
[0 1 2 3 4 5 6 7 8 9]
main ➤
```

Since, `for` syntax is not valid in package scope, we can initialize array `integers` of size `10` using `for` loop inside `init` function.

Package alias

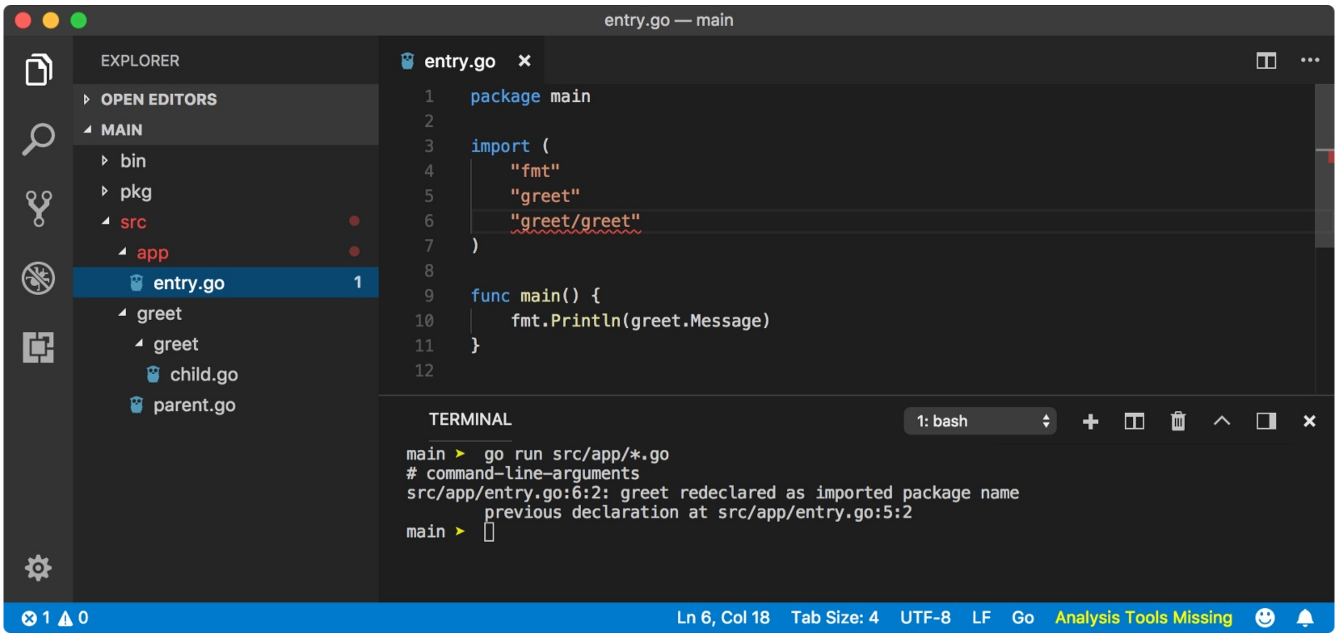
When you import a package, Go create a variable using package declaration of the package. If you are importing multiple packages with same name, this will lead into a conflict.

// parent.go
package greet

var Message = "Hey there. I am parent."

// child.go
package greet

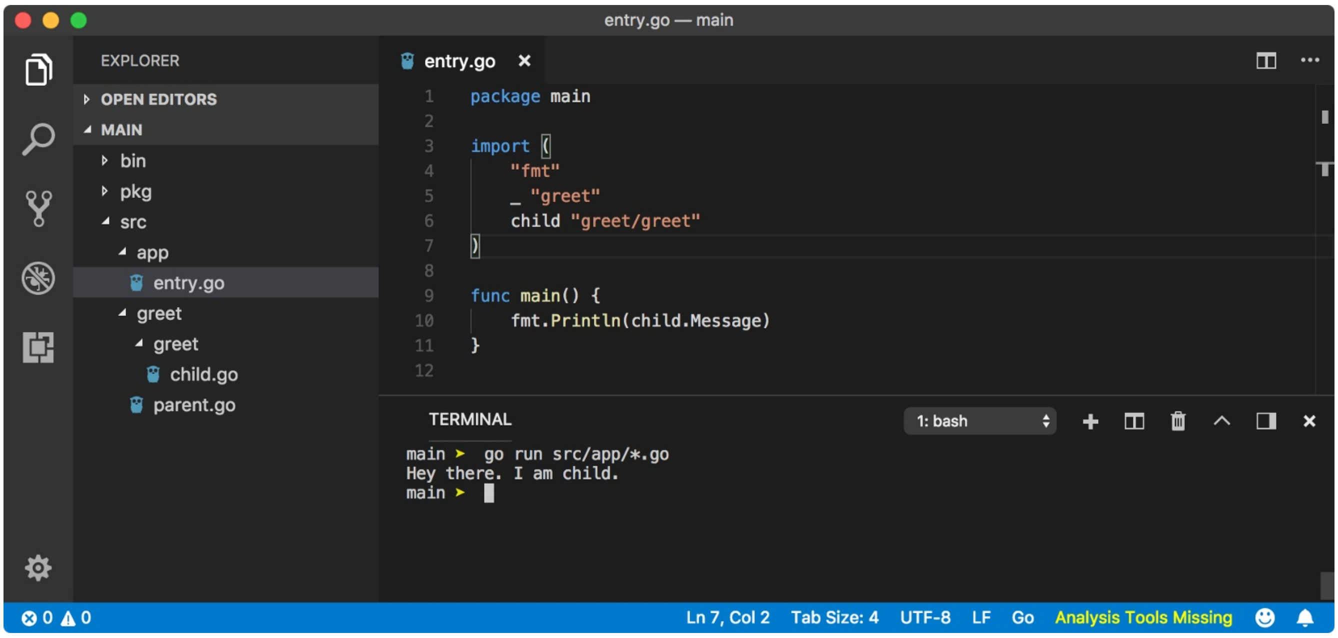
var Message = "Hey there. I am child."



```go
package main

import (
    "fmt"
    "greet"
    "greet/greet"
)

func main() {
    fmt.Println(greet.Message)
}
```

```
main ➤  go run src/app/*.go
# command-line-arguments
src/app/entry.go:6:2: greet redeclared as imported package name
        previous declaration at src/app/entry.go:5:2
main ➤  
```

Hence, we use **package alias**. We state a variable name in between `import` keyword and package name which is new variable to references the package.



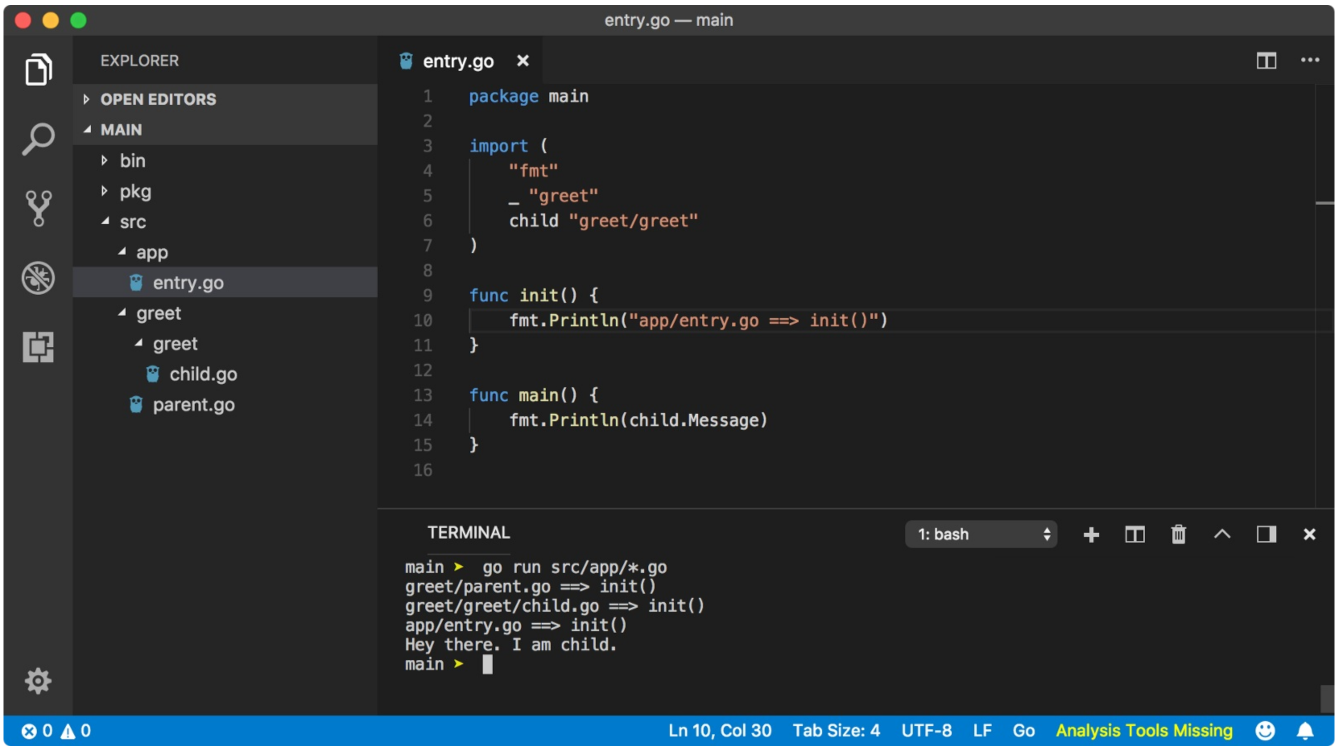
```go
package main

import (
    "fmt"
    _ "greet"
    child "greet/greet"
)

func main() {
    fmt.Println(child.Message)
}
```

```
main ➤  go run src/app/*.go
Hey there. I am child.
main ➤
```

In above example, `greet/greet` package now is referenced by `child` variable. If you notice, we aliased `greet` package with underscore. Underscore is a special character in Go which act as `null` container. Since, we are importing `greet` package but not using it, Go compiler will complain about it. To avoid that, we are storing reference of that package into `_` and Go compiler will simply ignore it.

Aliasing a package with **underscore** which seems to do nothing is quite useful sometimes when you want to initialize a package but not use it.

```go
// parent.go
package greet

import "fmt"

var Message = "Hey there. I am parent."

func init() {
  fmt.Println("greet/parent.go ==> init()")
}

// child.go
package greet

import "fmt"

var Message = "Hey there. I am child."

func init() {
  fmt.Println("greet/greet/child.go ==> init()")
}
```



```go
package main

import (
    "fmt"
    _ "greet"
    child "greet/greet"
)

func init() {
    fmt.Println("app/entry.go ==> init()")
}

func main() {
    fmt.Println(child.Message)
}
```

```
main ➤  go run src/app/*.go
greet/parent.go ==> init()
greet/greet/child.go ==> init()
app/entry.go ==> init()
Hey there. I am child.
main ➤
```

A main thing to remember is, **an imported package is initialized only once per package**. Hence if you have many import statements in a package, an imported package is going to be initialized only once in the lifetime of main package execution.

Program execution order

So far, we understood everything there is about packages. Now, let’s combine our understanding about how a program initializes in Go.

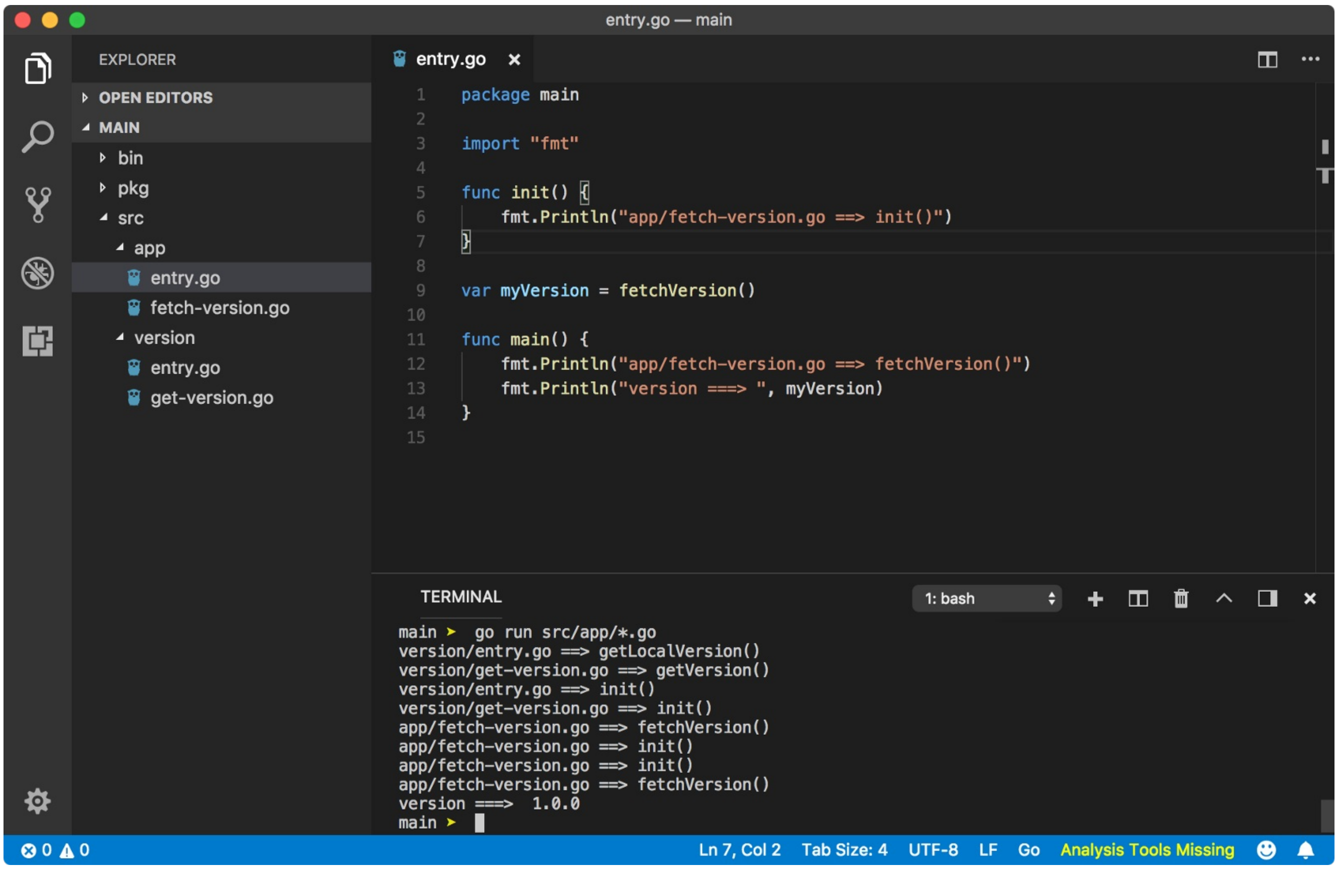
```
go run *.go
├── Main package is executed
├── All imported packages are initialized
|  ├── All imported packages are initialized (recursive definition)
|  ├── All global variables are initialized
|  └── init functions are called in lexical file name order
└── Main package is initialized
   ├── All global variables are initialized
   └── init functions are called in lexical file name order
```

Here is a small example to prove it.

```
// version/get-version.go
package version

import "fmt"
```

```
func init() {
 fmt.Println("version/get-version.go ==> init()")
}

func getVersion() string {
 fmt.Println("version/get-version.go ==> getVersion()")
 return "1.0.0"
}

/***************************/

// version/entry.go
package version

import "fmt"

func init() {
 fmt.Println("version/entry.go ==> init()")
}

var Version = getLocalVersion()

func getLocalVersion() string {
 fmt.Println("version/entry.go ==> getLocalVersion()")
 return getVersion()
}

/***************************/

// app/fetch-version.go
package main

import (
 "fmt"
 "version"
)

func init() {
 fmt.Println("app/fetch-version.go ==> init()")
}

func fetchVersion() string {
 fmt.Println("app/fetch-version.go ==> fetchVersion()")
 return version.Version
}

/***************************/
```

```
// app/entry.go
package main

import "fmt"

func init() {
 fmt.Println("app/entry.go ==> init()")
}

var myVersion = fetchVersion()

func main() {
 fmt.Println("app/fetch-version.go ==> fetchVersion()")
 fmt.Println("version ===> ", myVersion)
}
```



```
package main

import "fmt"

func init() {
    fmt.Println("app/fetch-version.go ==> init()")
}

var myVersion = fetchVersion()

func main() {
    fmt.Println("app/fetch-version.go ==> fetchVersion()")
    fmt.Println("version ===> ", myVersion)
}
```

```
main ➤  go run src/app/*.go
version/entry.go ==> getLocalVersion()
version/get-version.go ==> getVersion()
version/entry.go ==> init()
version/get-version.go ==> init()
app/fetch-version.go ==> fetchVersion()
app/fetch-version.go ==> init()
app/fetch-version.go ==> init()
app/fetch-version.go ==> fetchVersion()
version ===>  1.0.0
main ➤
```

Installing 3rd party package

Installing a 3rd party package is nothing but cloning the remote code into local `src/<package>` directory. Unfortunately, Go does not support package version or provide package manager but a proposal is waiting ***here***.

Since, Go does not have a centralize official package registry, it asks you to provide hostname and path to the package.

```
$ go get -u github.com/jinzhu/gorm
```

Above command imports files from `http://github.com/jinzhu/gorm` URL and saves it inside `src/github.com/jinzhu/gorm` directory. As discussed in nested packages, you can import `gorm` package like below.

```
package main

import "github.com/jinzhu/gorm"

// use ==> gorm.SomeExportedMember
```

So, if you made a package and want people to use it, just publish it on GitHub and you are good to go. If your package is executable, people can use it as command line tool else they can import it in a program and use it as utility module. Only thing they need to do is use below command.

```
$ go get github.com/your-username/repo-name
```