Anatomy of Modules in Go

Modules are a new way to manage dependencies of your project. Modules enable us to incorporate different versions of the same dependency without breaking the application.

Uday Hiwarale Follow
May 4 · 21 min read

(source: pexels.com)

> *Before we begin, it's worth mentioning that Modules are supported from Go version **1.11** but it will be finalized in Go version **1.13**. So, if you are using Go version below 1.13, then implementation of Go Modules might be subjected to change in the future.*

Let's talk about **pre-Go Modules** era. As we discussed in our earlier tutorials, in order to work with Go, we need our source code to be present inside `$GOPATH` directory (*this is your Go **workspace***). When you install any dependency packages using `go get` command, it is saved in the workspace directory. A Go program cannot import a dependency unless it is present inside `$GOPATH`. `go build` command creates **binary builds** and **package archives** inside `$GOPATH`. So `$GOPATH` is a big deal in Go.

always hated this as it forces us to relocate all the projects inside one directory. Also, I you can't install multiple versions of the same dependency package since `go get` puts the dependency package at the same location despite having different versions (*as directory name is same as the package name*). If you really need to work outside `$GOPATH` directory, then you will be switching ***GOPATH*** environment variable back and forth. And when you do that, you need to reinstall or copy all the packages in this new directory.

Here is an interesting fact, **Go does not have a central repository for downloading packages**. If you are a **node.js** developer, then you are familiar with **npm**. **npm** is the central registry to download and publish node.js modules (*packages*). Then how Go does it?

Typically, a **third party Go package** import statement looks like this

```
import "github.com/username/packagename"
```

If you look at package import statement, **package name** looks like an **URL**, because it is. Go can download and install packages located anywhere on the internet. To install the above package, you have to use `go get` command

```
go get github.com/username/packagename
```

Go visits the URL `https://github.com/username/packagename` and downloads the package if URL is successfully resolved. Once the download is complete, it saves the package content inside `github.com/username/packagename` directory under `$GOPATH/src`. As we have learned in Go installation tutorial, standard Go packages like located inside **$GOROOT** directory (*where Go is installed*). Other project level dependencies are stored inside **$GOPATH**

> *If you compare that with **npm**, npm stores packages inside the project folder (*inside `node_modules` directory*) and not at a central location like **$GOPATH (**but still not cenral as $GOPATH can be changed**)**. With npm, we also*

> *have* `package.json` *(along with* `package-lock.json`*) which is a text containing packages installed for the given project and their precise versions.*

⇨ Read my tutorial on Go packages from here.

Let's understand how `go get` works and how to host your own custom Go repository. Like we installed a package above, when you execute `go get` command like below

```
go get domain.com/path/to/sub/directory
```

Go visits website `domain.com/path/to/sub/directory` securely using the **HTTPS** protocol. If the given URL does not support **HTTPS** or results in SSL error, and if `GIT_ALLOW_PROTOCOL` environment variable contains **HTTP** protocol, then Go attempts to resolve the package URL with **HTTP** protocol.

A Go package located on the internet is a **version control system repository (VCS)** like **GIT** or **SVN**. Supported version control systems are mentioned below

```
Bazaar          .bzr
Fossil          .fossil
Git             .git
Mercurial       .hg
Subversion      .svn
```

If the `domain.com` is one of the code hosting sites **recognized** by Go, Go first tries to resolve `domain.com/path/to/sub/directory.{type}` where `type` can be `.git`, `.svn` etc. supported by the recognized website mentioned below

```
Bitbucket               (bitbucket.org)     .git/.hg
GitHub                  (github.com)        .git
Launchpad               (launchpad.net)     .bzr
IBM DevOps Services     (hub.jazz.net)      .git
```

When a **VCS** supports multiple protocols, Go tries to resolve the URL using all the supported protocols one at a time. For example, as Git supports `https://` and `git+ssh://` protocols, they both are tried in turn.

If Go successfully resolves the above URL, the **VCS** repository is cloned (*using **VCS** tools like Git*) and saved to the relevant path in `$GOPATH/src` as discussed earlier.

But If the package URL is not one of the recognized code hosting sites, Go have no way to determine the **VCS**. In that case, Go tries to resolve the URL as it is using supported protocols discussed earlier. **If the URL resolves successfully with an HTML document**, go specifically look for the specific **META tag** mentioned below

```
<meta name="go-import" content="import-prefix type repo-root">
```

> *It is recommended that this meta tag should appear as early as possible in the HTML source code, to adoide any possible parsing errors.*

Let's talk about this meta tag a little.

- **import-prefix:** This the import path of your module. In our case it is `domain.com/path/to/sub/directory`
- **type:** This is the type of your **VCS** repository. It can be one of the supported types mentioned earlier. In our case, it could be a Git repository, hence `git` type.
- **repo-root:** This the URL where the **VCS** repository is located. This should be a fully qualified version repository. For example, in our case, it could be `https://domain.com/repos/name.git` . This `git` the extension type is **optional** as we already mentioned the **type** as `git` .

Using this meta information, Go can clone the repository located at `https://domain.com/repos/name.git` and save inside `$GOPATH/src` under directory name `domain.com/path/to/sub/directory` .

If your **import-prefix** is different than the URL used in the `go get` command, then Go will clone the repository under that directory mentioned by **import-prefix** value. For example, if our `go get` command is like below

```
go get domain.com/some/sub/directory
```

Go will visit `https://domain.com/some/sub/directory` and if the server returns an HTML document with below meta tag

```
<meta name="go-import" content="domain.com/someother/sub/directory
git https://domain.com/repos/name.git">
```

Since the **import-prefix is different than the URL** used in `go get` command, Go will verify if `domain.com/someother/sub/directory` has the same meta tag as `https://domain.com/some/sub/directory/` and install the package under the directory name `domain.com/someother/sub/directory` and our package import statement will be like below

```
import "domain.com/someother/sub/directory"
```

I guess, so far we have gained a very good amount of knowledge of how **traditional package management** works in Go. Let's see how this can be harmful when a package becomes **backward incompatible**.

Let's say that a package located at `github.com/thatisuday/stringmanip` is currently supports string manipulation utilities like making string uppercase and all. When people download this package using `go get`, **they are cloning the master repository at the most recent commit**.

Suddenly new commits come in which either changes the implementation of utility functions, adds breaking changes or create bugs. Now, when people

upgrade/reinstalls this package, their application starts to break.

There is no way we can force `go get` to point to a **specific commit** or **release tag** in the Git history (***hope is not a tactic — DWH***). That means, **there is no way to download a package pointed to a specific version**. Also, since Go saves the package under the directory name specified by the package itself, **we can't save multiple versions of the same package** at a time. This is just brutal and **Go Modules** are here to rescue.

Let's first understand the mechanism of the Go modules, **in theory**. As we discussed some problems with Go's traditional dependency management, there are few modifications we can easily propose.

- First of all, we should be able to work from any directory, not just `$GOPATH` which gives us the flexibility to relocate our source code anywhere.
- We should be able to install the precise version of a dependency package to avoid breaking changes.
- We should be able to import multiple versions of the same dependency package. This is useful when our old application code is running with the old version of the dependency but we would like to use the new version of the dependency package for any new developments.
- Like `package.json` in npm, we need a file that can list the dependencies of our project. This is helpful because when you distribute your project, you don't need to distribute your dependencies with it, Go can just look at this file and download the dependencies for you.

Well, the good news is, Go modules can do all these for you and **mucho more**. Go modules gives us a built-in dependency management system. But, let's understand **what a module is** first. A module is a directory containing Go packages. These can be distribution packages or executable package.

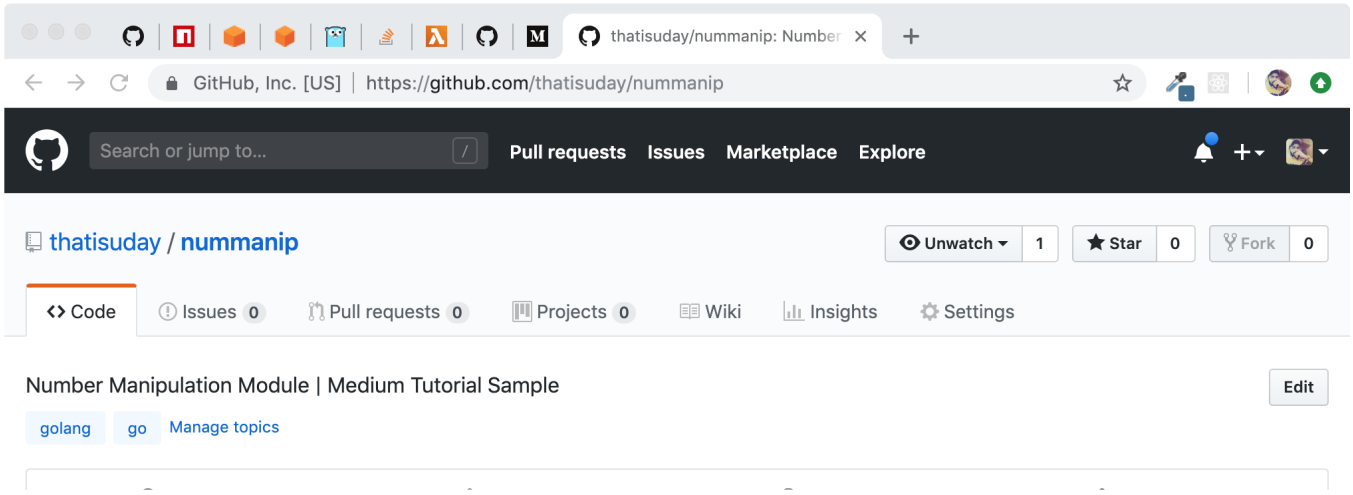
A module is also like a package that you can share with other people. Hence, it has to be a Git or any other **VCS** repository which we can be hosted on a code-sharing

platform like Github. Hence, Go recommends

- A Go module must be a **VCS** repository or a **VCS** repository should contain a single Go module.
- A Go module should contain one or more packages
- A package should contain one or more `.go` files in a single directory.

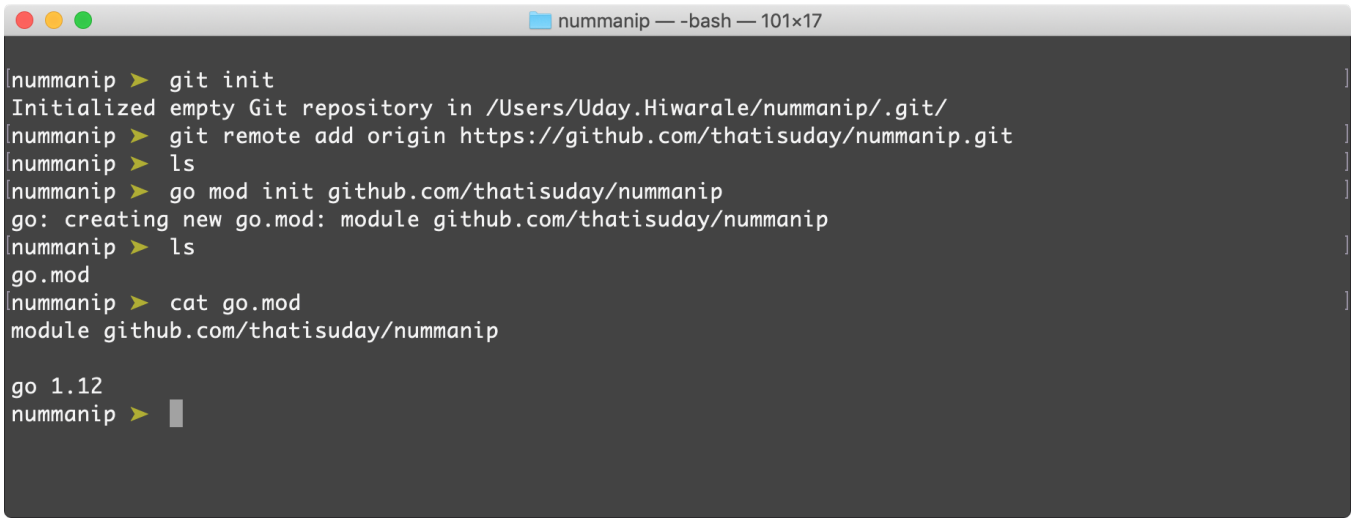
Enough theory, let's code now. Let's create an empty directory anywhere but inside `$GOPATH`. I am using `nummanip` directory to host my module and it will contain some packages to that to manipulate `number` data type.

As discussed before, since a Go module should be a **VCS** repository, I have created a Git repository on Github with below URL.



https://github.com/thatisuday/nummanip

Now, the first step is to initialize the Go Module inside this directory. For that, we have `go mod init` command. This command creates `go.mod` file (*which is like* `package.json` *file in npm*) that contains module import path and dependencies used by packages inside it. We should also initialize the Git repository with the above GitHub remote URL.

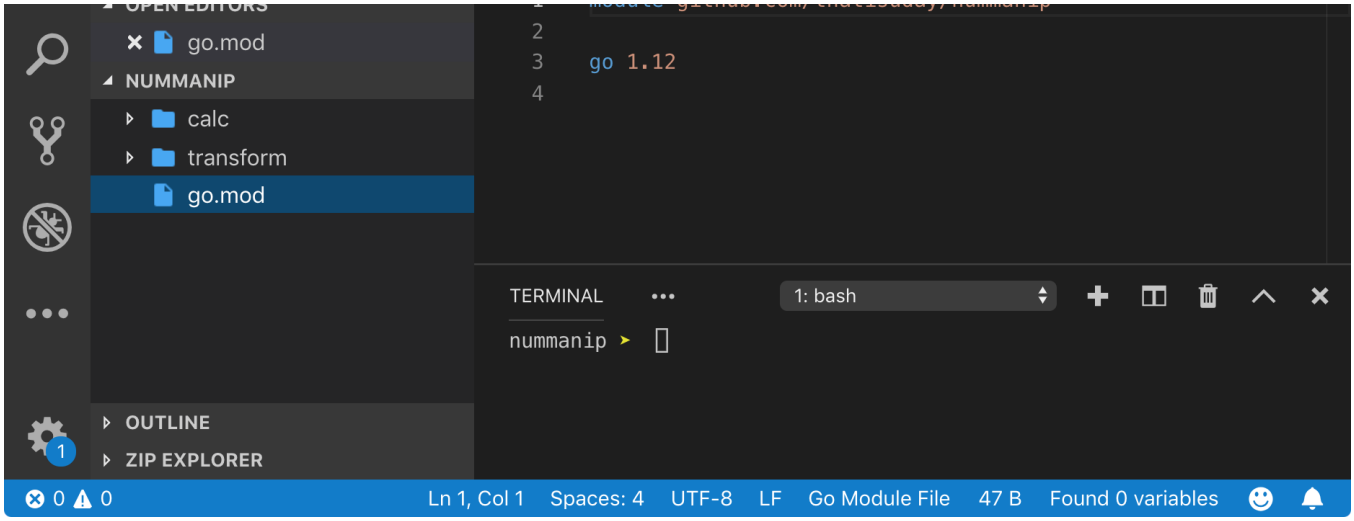


In the first step, we initialized a Git repository and pointed it to the Github repository we just created. In the second step, we initialized a Go module and specified module import path in the command.

```
go mod init github.com/thatisuday/nummanip
```

> ⚠ *Creating modules inside* `$GOPATH` *is disabled by default. You will get this error if you initiate a go module inside* `$GOPATH` → `go: modules disabled inside GOPATH/src by GO111MODULE=auto; see 'go help modules'` *. This is a preventive measure to deprecate* `$GOPATH` *in the near future. If you really need to create Go modules inside* `$GOPATH` *, set* `GO111MODULE` *environment variable to* `on` *.*

It created a `go.mod` file which contains the module import path and version of the Go this module was created on. The most difficult part is done and from here, we should focus on developing our packages.

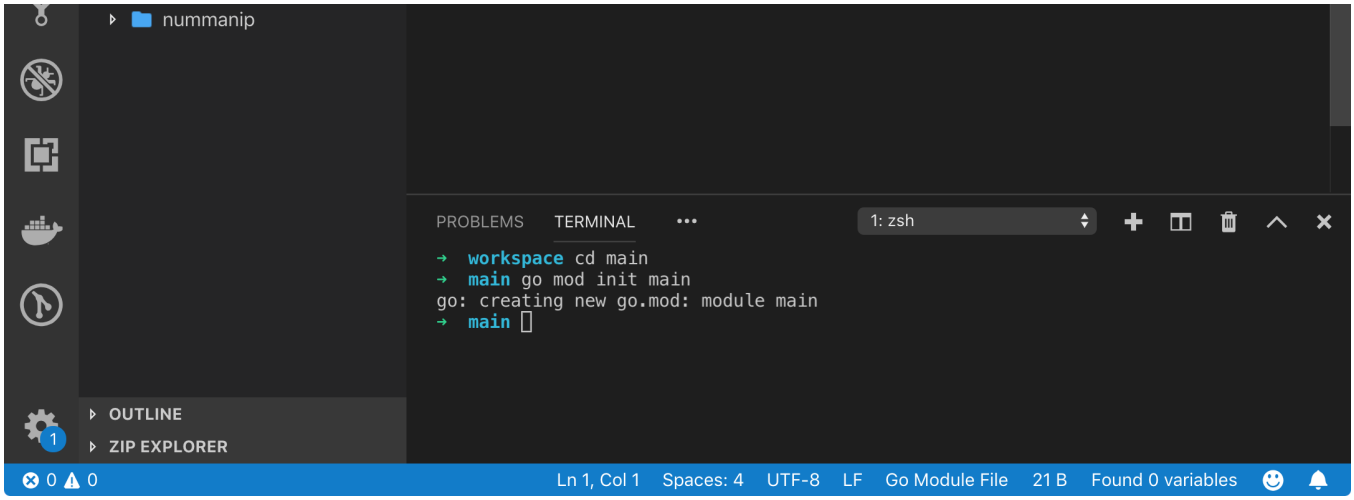


We have created two packages inside our module. Currently, these are empty directories but soon, we will put some code there. ***calc*** package will export utilities to calculate something with numbers while transform will be used to transform numbers or ***number***related data types (*like an array of numbers*).

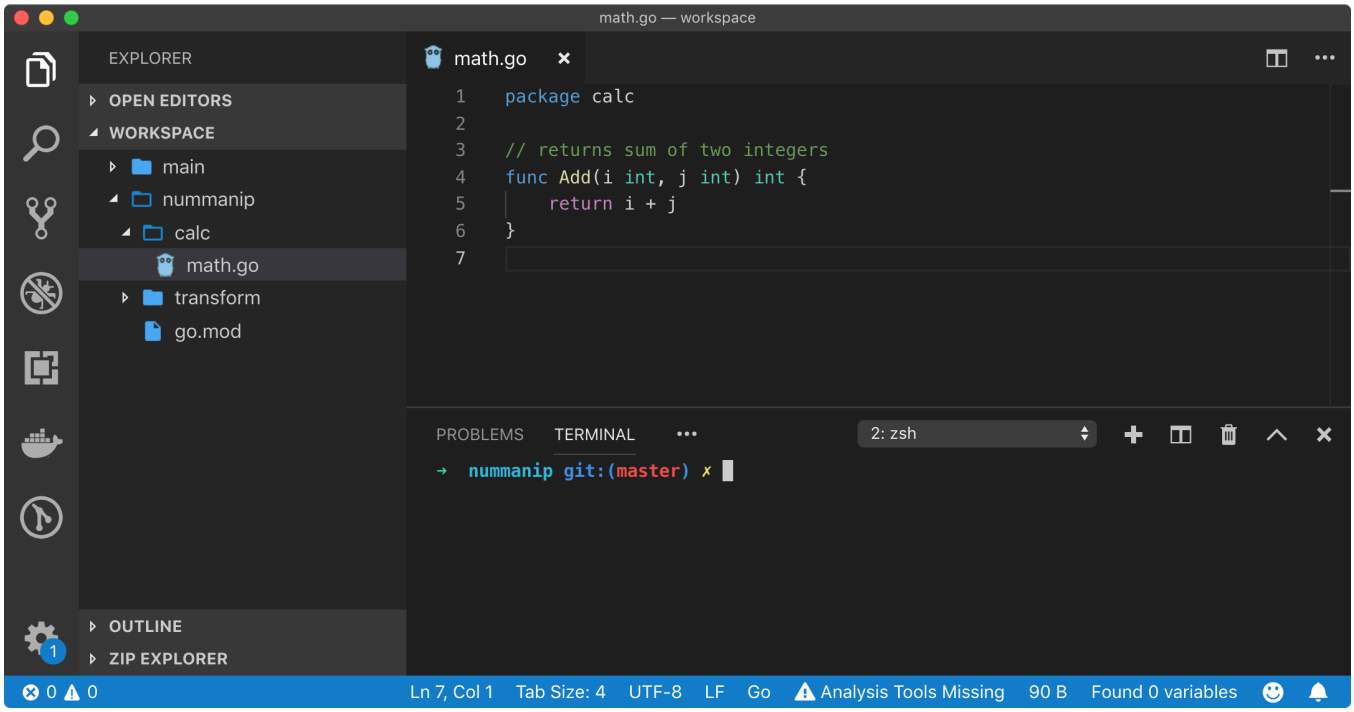
> ⚠ *Since we are providing multiple packages through our module, we have created directories for each of them. But if want to just provide single package, you don't need to create a separate directory for it. You can just write the package code inside the module directory (*where `go.mod` file is*). The only difference is how you import your package, which we will see later.*

At this point, we are not sure if our module is a fully-fledged **executable application** or a **distribution module with utility package(s)**. I always recommend keeping your reusable logic in separate packages and import them to your applications. So to test our module and packages, we are going to create another module which will consume `nummanip` module packages. We won't be publishing this test module as it is just for testing, hence we can init this module with a **non-URL** module name.

> *Go provide `go test` utility to test your packages with custom test suits. But that is a completely different topic which we will perhaps cover in upcoming tutorials.*



We have created `main` module for testing purpose using the command `go mod init main` which created `go.mod` file. Also for simplicity, we have opened both `main` and `nummanip` module in VSCode as workspace.



We have created `math.go` file inside `calc` package to provide a utility function to **return the sum of two numbers**. Focus on package declaration part, `package calc` signifies that `math.go` belongs to `calc` package and since this package is in a separate directory under our module, **package declaration has nothing to do with the module name**.

> *If our package code were inside the module directory, then package name should be same as the module name (in order to successfully resolve import statement).*

Let's publish our module. Publishing is just pushing your code to the remote **VCS**repository with a tag. But before we do that, we need to understand **semantic versioning** and **git tags**. So let's get into it.

Semantic Versioning or **SemVer** is a universally accepted format to tag your release packages/modules etc. It has v**X**.**Y**.**Z** format where **X** is a ***major*** version number, **Y** is a ***minor*** version number and **Z** is ***patch*** version number (*translated as* **v*Major.Minor.Patch***). For example, a package with version `1.2.0` has the ***major***version `1`, ***minor*** version `2` and ***patch*** version `0`. We bump only ***patch*** version when there is a small change in package, ***minor*** in case new or enhanced functionality is added. When there is a major difference between the old version and the new version, we bump up the ***major*** version number resetting ***minor*** and ***patch*** version to `0` (*for example* `2.0.0`).

> *Additional pre-release tags can be added as a suffix with release tag which goes like this* `x.y.z-rc.0` *which means it is a release with release-component* `0` *(*or `x.y.z-beta.1`*). This is helpful to those who want to test the beta version.*

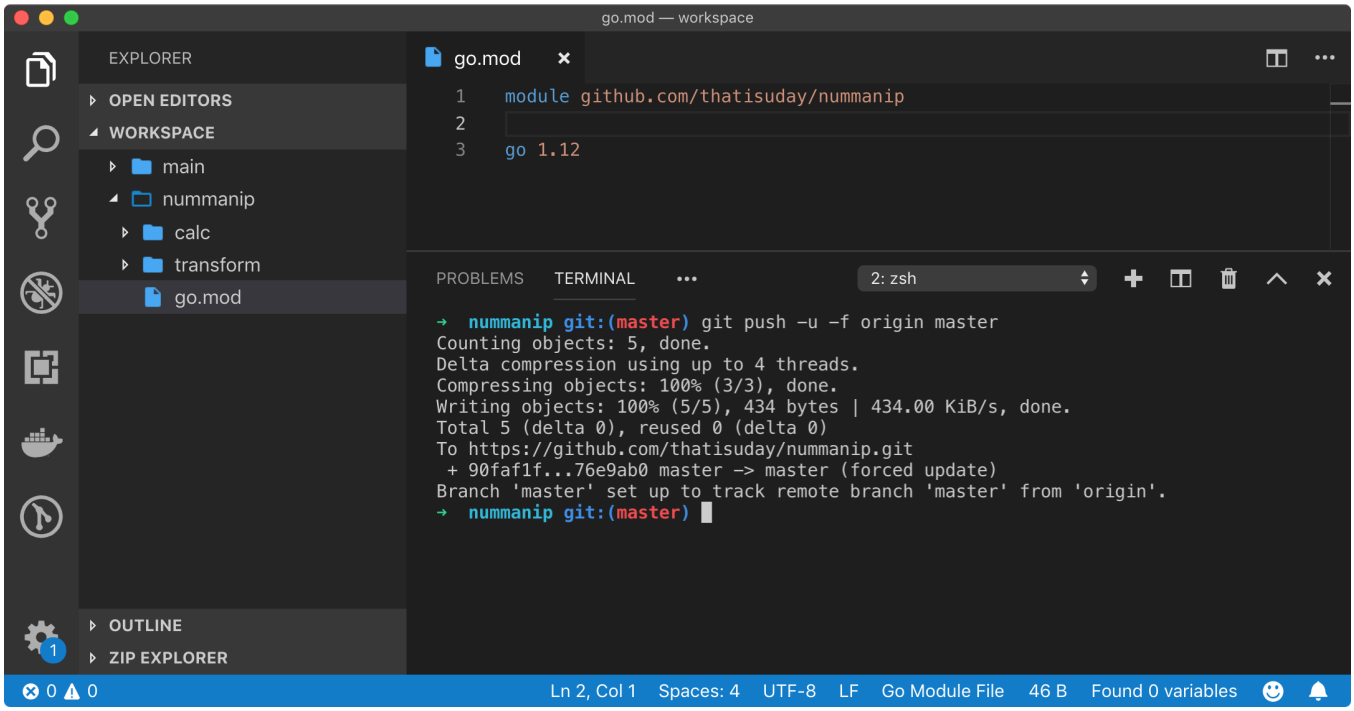
Go specifies that, when there is an incompatibility between the new version and the old version, then the new version should be a major release. When a major version is released, Go treats it as a different module which we will see later in this tutorial and discuss why that is important.

As we know, Git branch is nothing but a history of commits. Each commit has a unique identifier (*commit hash*) associated with them. At any specific commit, we can extract the state of the files in the repository. When you want to release anything for public use, you need to provide a commit hash where the state of the files in the repository is stable and can be used in the production. But what we can also do is create an alias for the commit hash with the SemVer. It's called **tagging**.

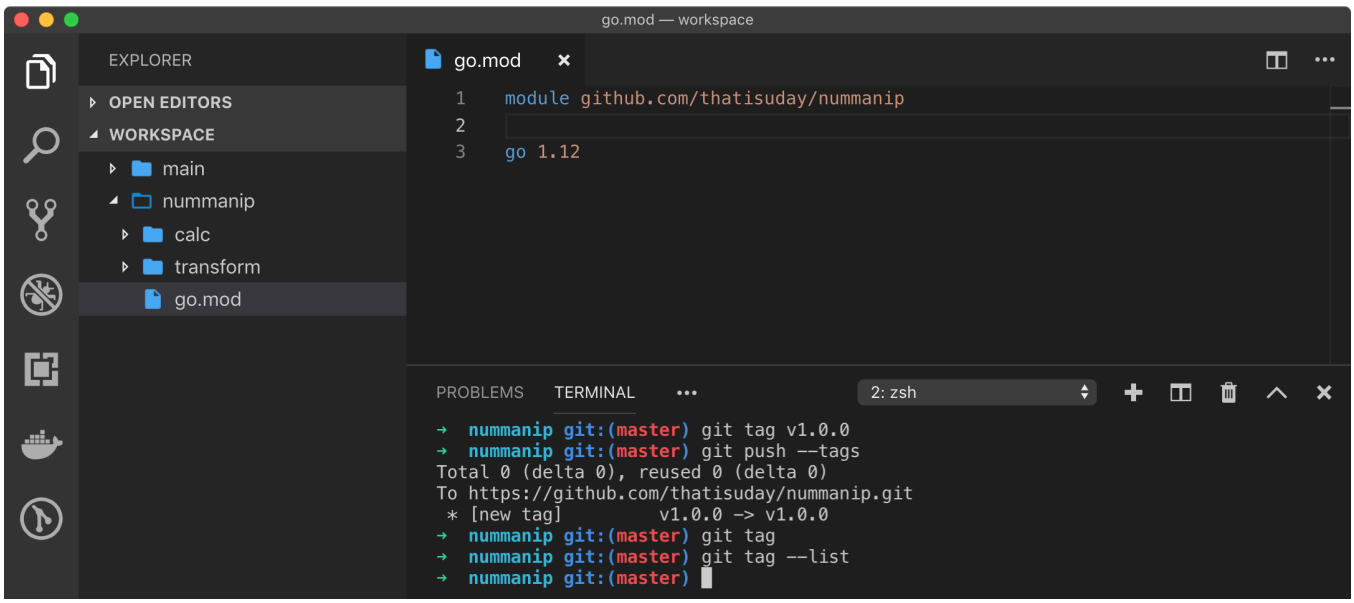
Git provides two types of tags. The **Lightweight Tag** is simply a pointer to commit in the Git History. While **Annotated Tag** is saved as a full object (*which contains additional information like Tagger's name, tag message, and other info*) in the Git

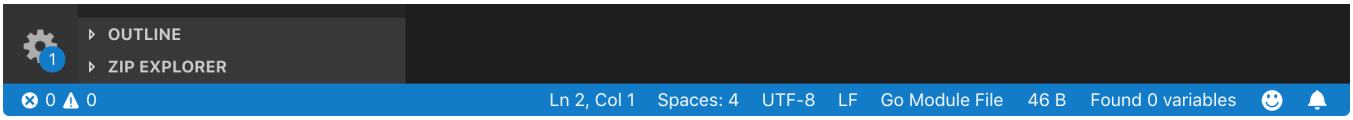
database. You can read about Annotated Tags ***here*** in depth because for simplicity, we will use lightweight tags.

Since we are releasing our module for the first time, we need to create a commit and push it to the remote repository. Then we will create a tag from the last commit (*which will be the one we pushed*) with a SemVer.

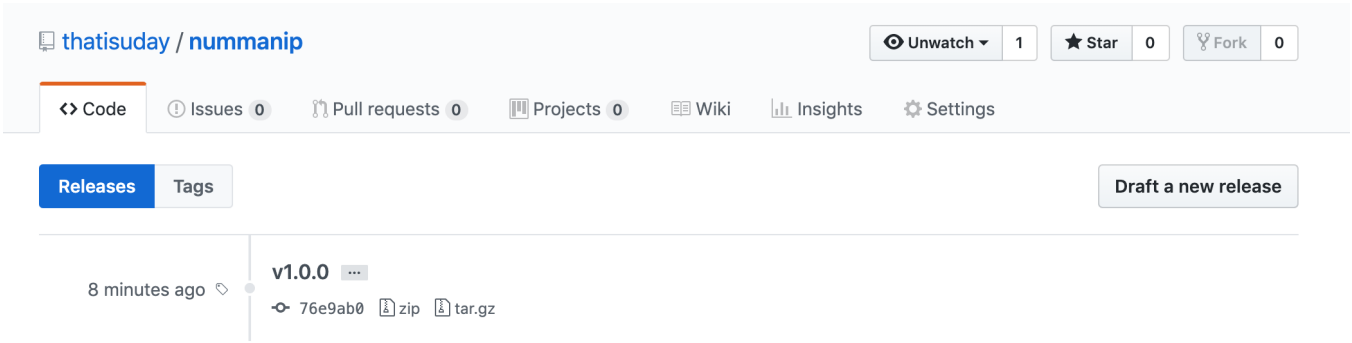


From above, we have created a commit and pushed it to the remote master branch. `-f` flag was used to forcefully push the local git history to the remote which is OK for the first ever commit. Let's create a Git Tag.

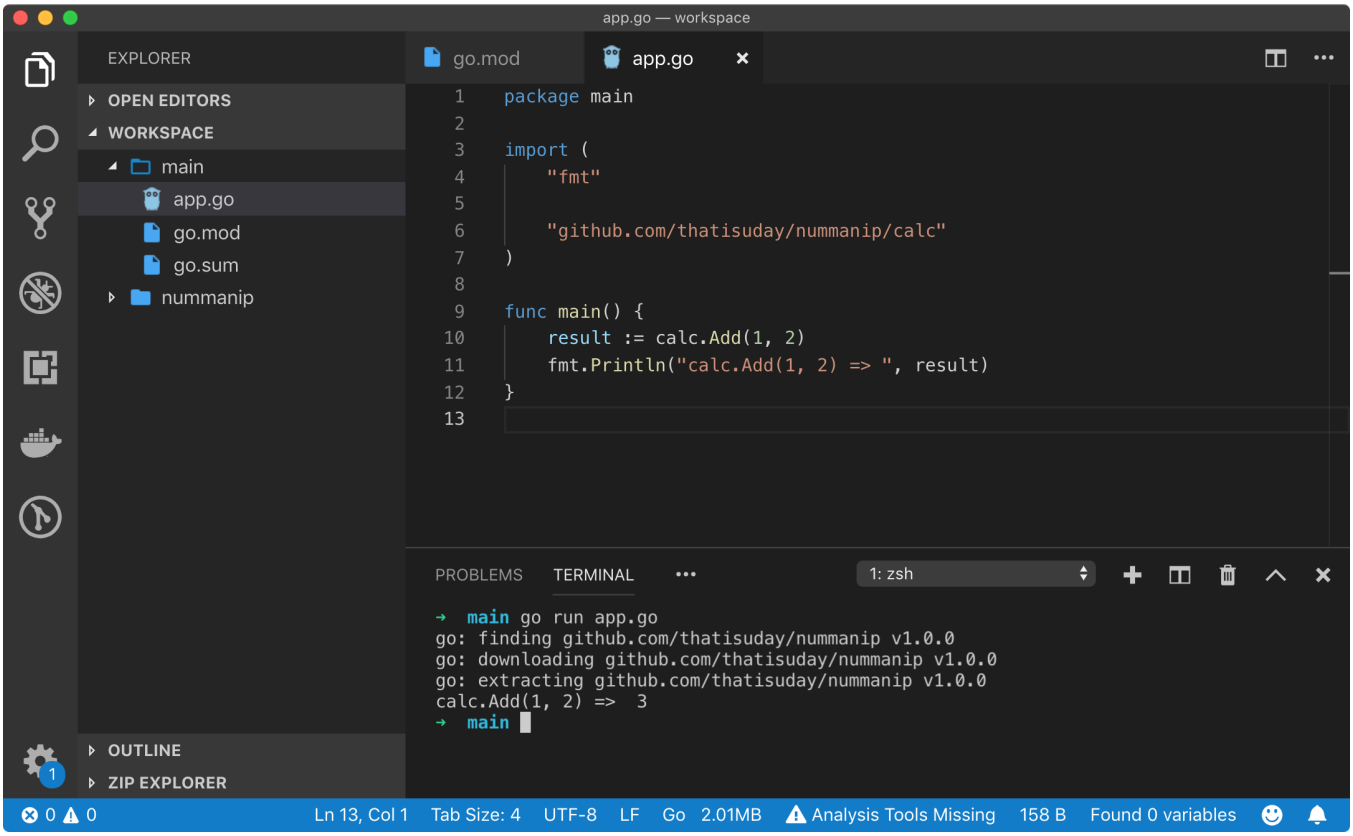




Since we are releasing the first version of our module, the SemVer should be `v1.0.0`. When you are releasing a Go module, your SemVer tag must start with a lowercase `v` (*for successful version resolution*). When you create a Git Tag, you need to push it to the remote repository with `git push --tags`.



GitHub provides information about tag inside the Releases section.



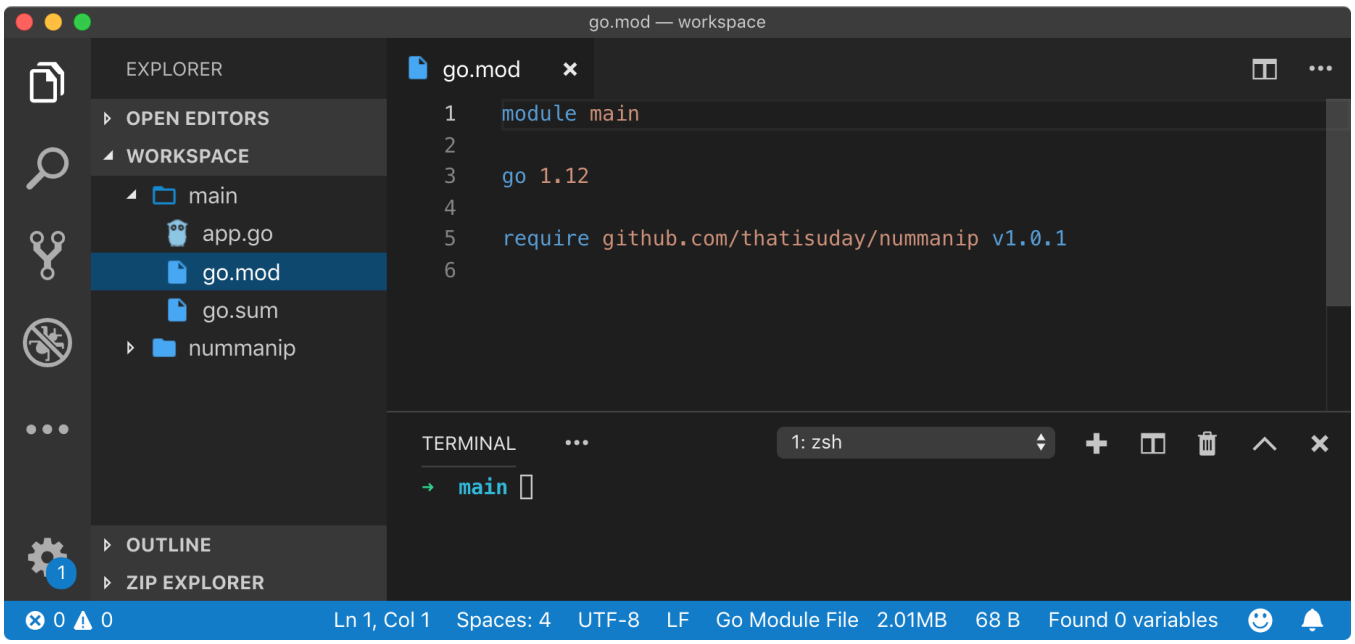
I have created `app.go` file inside `main` module to test if everything is working fine. Basically, we are importing `calc` package

from `github.com/thatisuday/nummanip` module to utilize `Add` method. Since we know the import path of the module (*and package in it*), we can import directly in our code with full URL path to the package.

> *If our package code were inside module directory itself, we would have imported it with `github.com/thatisuday/nummanip` only and we would have needed to use `nummanip` as the package variable to run `nummanip.Add()`.*

Notice that until now, we haven't installed this module yet using `go get`. Yes, to install a module, you can also use `go get` command. When we run `app.go` using the command `go run <file>`, Go fetches latest version `v1.x.x` (*explained later*) of `nummanip` module. When Go successfully downloads the module, it also updates the `go.mod` file to save the downloaded dependency.

> *This way, we do not need to tell other people which dependencies to install. Go can just look at `go.mod` file for dependencies needed by the module.*



go.mod file

> ⚠ *You might ask, how Go resolves the package import URL as `https://github.com/thatisuday/nummanip/calc` returns **404 Not Found** page. I couldn't find exact documentation on it but my guess is that since GitHub is one of the recognized code hosting sites, it knows where the package is located and **this** documentation suggests that.*

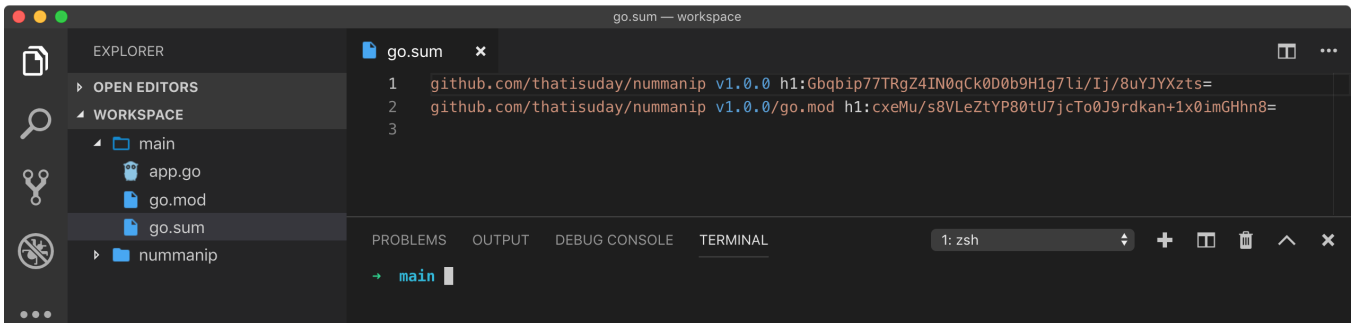
Now, there are many questions popping in our heads. When go will install the module automatically? What's the use of `go get` then? Where are the modules stored on the system?

Go modules are stored inside `$GOPATH/pkg/mod` directory (***module cache directory***). Seems like we haven't been able to get rid of `$GOPATH` at all. But Go has to **cache** modules somewhere on the system to prevent repeated downloading of the same modules of the same versions.

When we execute the command `go run` or other Go commands like `test`, `build`, Go automatically checks the third party import statements (*like our module here*), and clones the repository inside module cache directory.

`$GOPATH/pkg/mod`

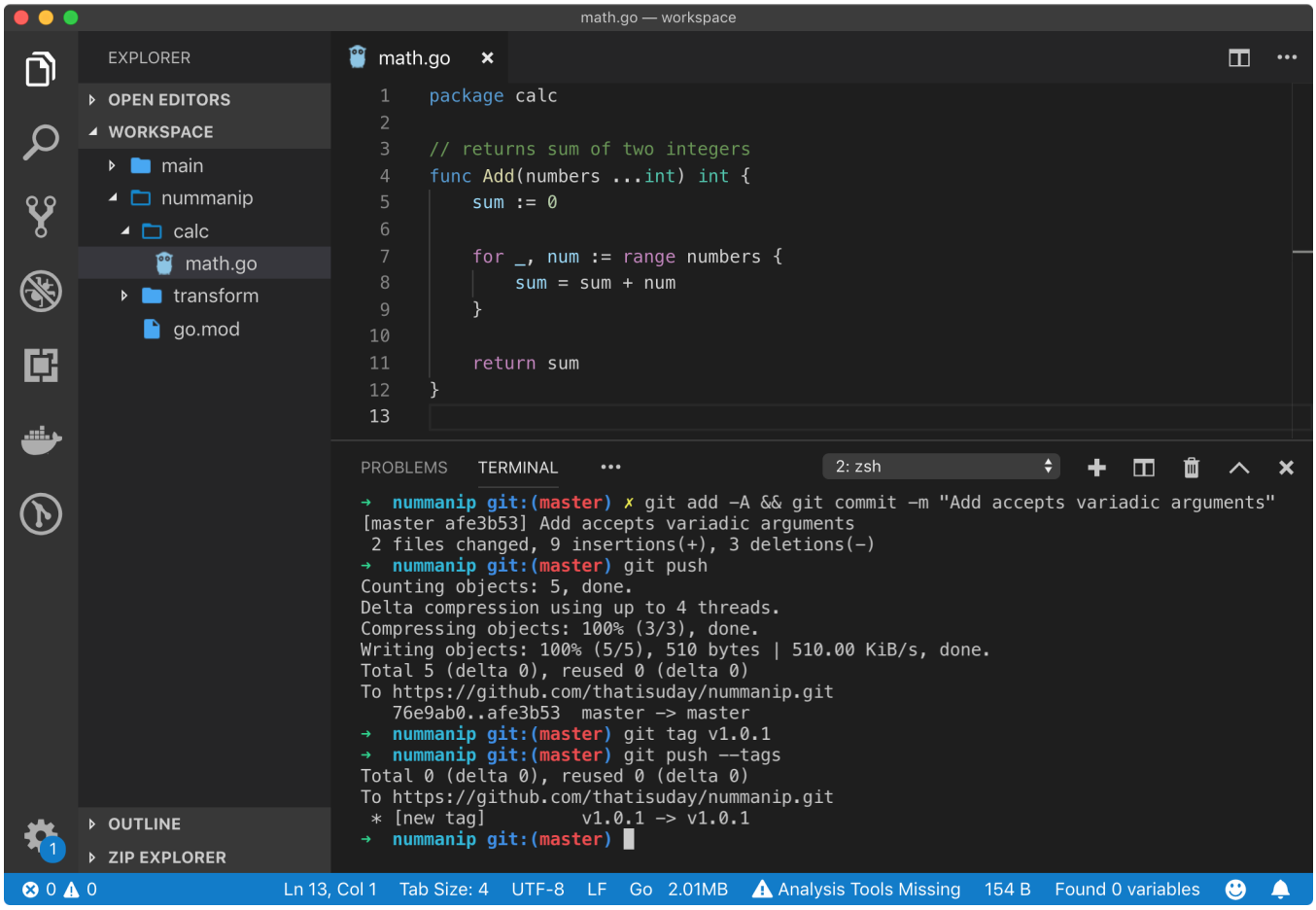
We can see the cache directory has `nummanip` module tagged with version `v1.0.0` to make import resolution simple. Go creates `go.sum` file to save checksums of the downloaded **direct and indirect dependencies' content**.



`go.sum` file

Unlike `package-lock.json` file in `npm` which stores versions of the currently installed direct and indirect dependencies for **100% reproducible builds**, `go.sum` file is not a lock file and but it should be committed along with your code in case your module is a VCS repository (***explained here***). When you shared your module, `go.sum` plays an important role to ensure **100% reproducible builds** by validating the checksum of each module.

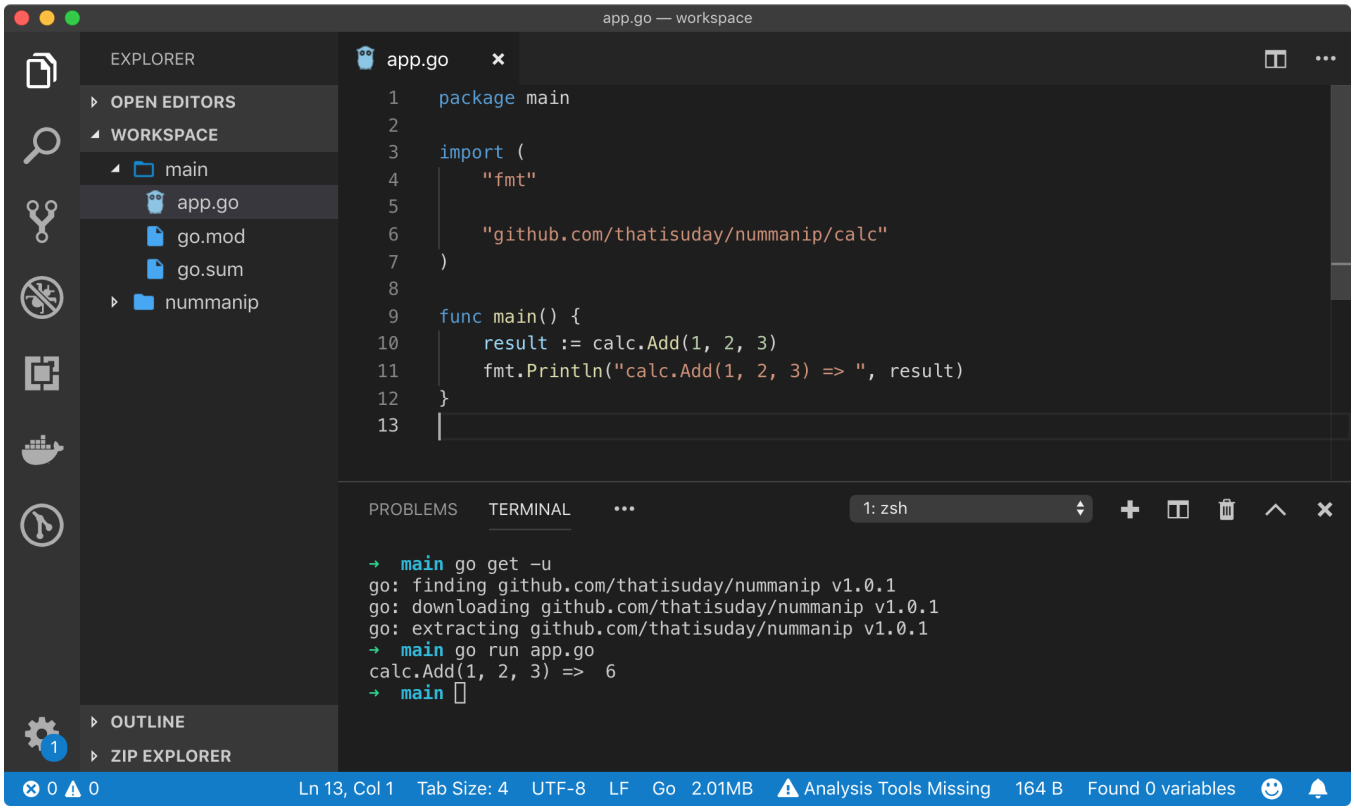
Let's add some ***patch*** version level functionality to our `calc` package.



Add is not variadic function

What we changed in Add function is the ability to accepts multiple arguments using a variadic function argument (***read my tutorial***). We then pushed new commit with tag `v.1.0.1`.

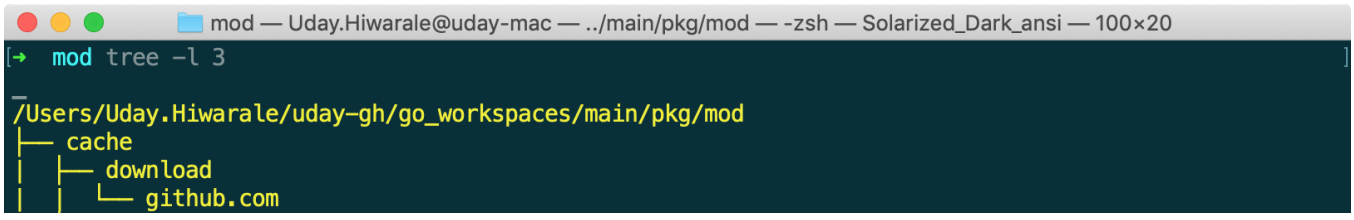
Let's implement newly improved `Add` function inside `main` module. But there is a gotcha here. Since Go already have our module, `go run` or other commands will not fetch the newer version. To do that, we need to update our module **manually** (*in worst case reinstall it*).

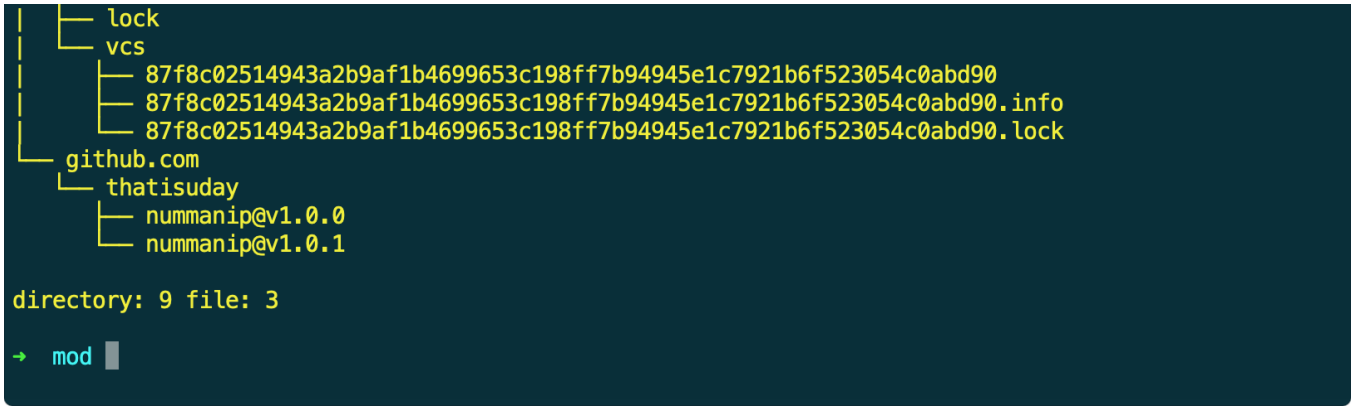


update Go modules

To update Go modules present inside a module with `go.mod` file, use `go get -u` command. This command will update all the modules with the latest ***minor*** or ***patch***version of the given **major** version (***explained later***). If you want to update only patch version even if the newer minor version is available, use `go get -u=patch` command instead.

If you need a precise version of a dependency module, you can use `go get module@version` command, for example, in our case to install `v1.1.2`, we should use `go get github.com/thatisuday/nummanip/calc@v1.1.2`.





`$GOPATH/pkg/mod`

As you can see from the above screenshots, Go have downloaded the latest version `v1.0.1` and saved in the module cache directory. This is a great way to manage dependencies on a global level, where multiple modules on the system can access different versions of the dependency.

Upgrading to a Major Version

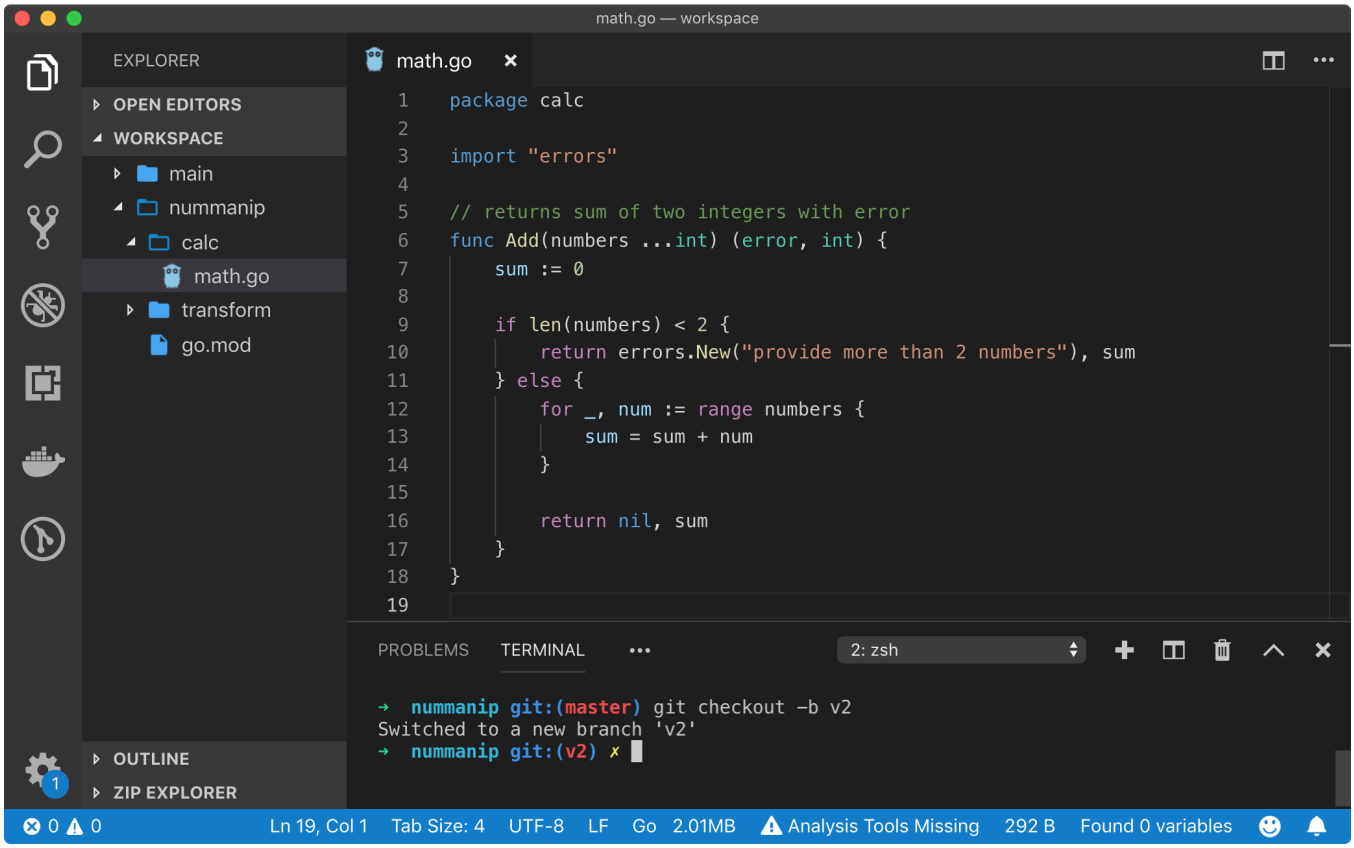
Now the time has come to explain why Go only automatically resolves only `v1.x.x` versions of a dependency module or why `go get -u` updates only modules within the current major version.

It is generally accepted that a major version of a dependency is needed when there is a substantial change in how given dependency works. For example, **Angular v1** is very different from **Angular v2**. They are both works in very different ways. Also, when a major version of a dependency is imported, it could lead to incompatibilities which means your old code might now work with the new major released version of the dependency.

Hence, if `go get -u` would have updated the dependency module to a major version for example, from `v1.0.2` to `v2.0.0` then our application might now have worked/built properly. Then how Go solves this issue?

Go says, when you update your module to a newer major version, technically it's a different module as it's not backward compatible. So module with `v1.x.x` is different package compared to `v2.x.x`. That means the consumer who has imported your module must manually install module with the new major version using `go get` and specify the **new import path**.

So what's the deal with **new import path**? As we have `go.mod` file which specifies the module import path at the top, we need to change to something else. But don't worry, the only modification we need is to append major version code (v**X**) to the import path so that consumer can import multiple versions of the same package. Let's see that in action.



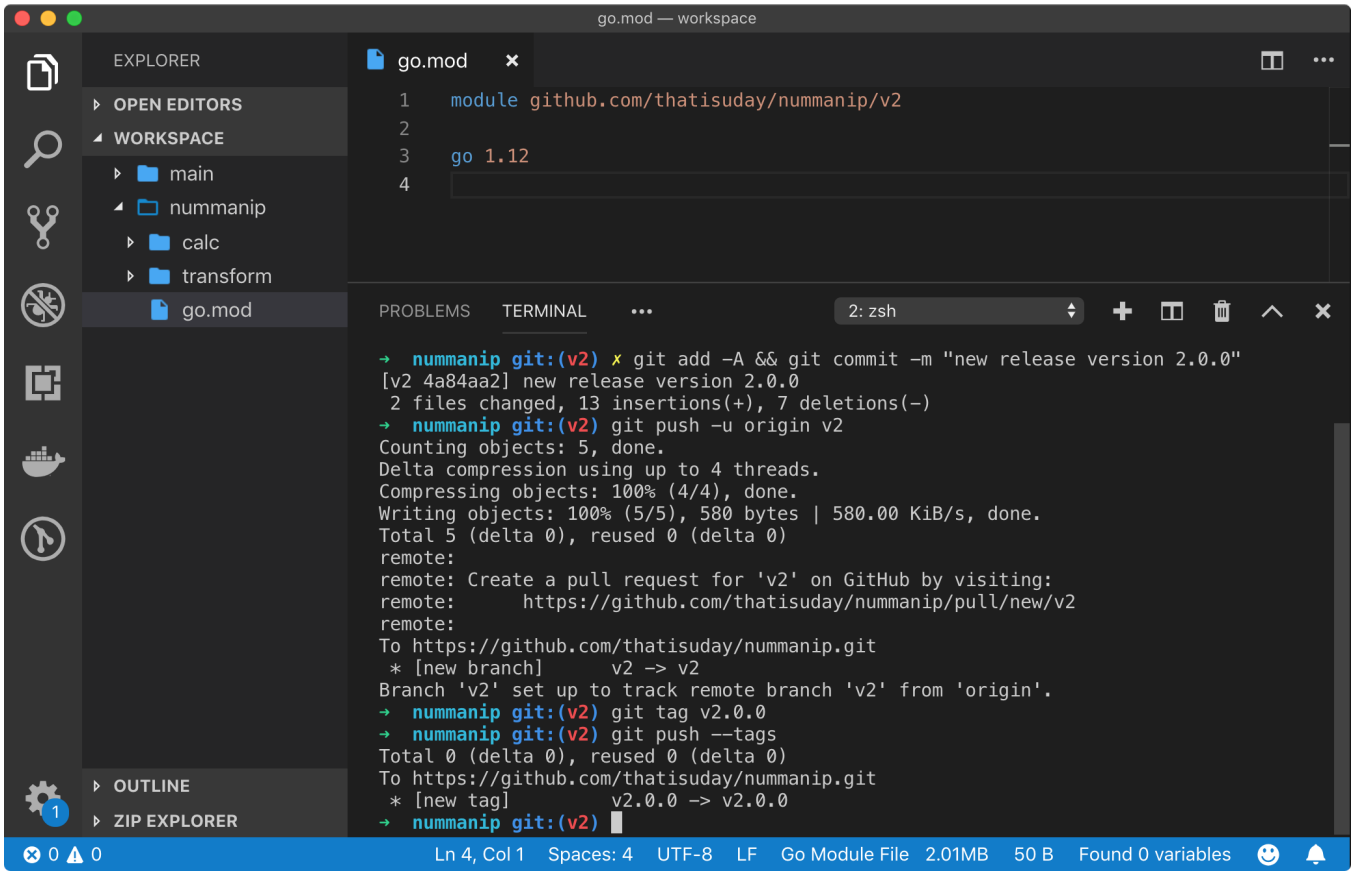
incompatible changes in the Add function

From above we can see that, we added the functionality inside the function `Add` to check if arguments are greater than 2. If arguments passed is less than 2, then `Add` will return an error as its first return value, else `sum` as its second return value, which also makes sense.

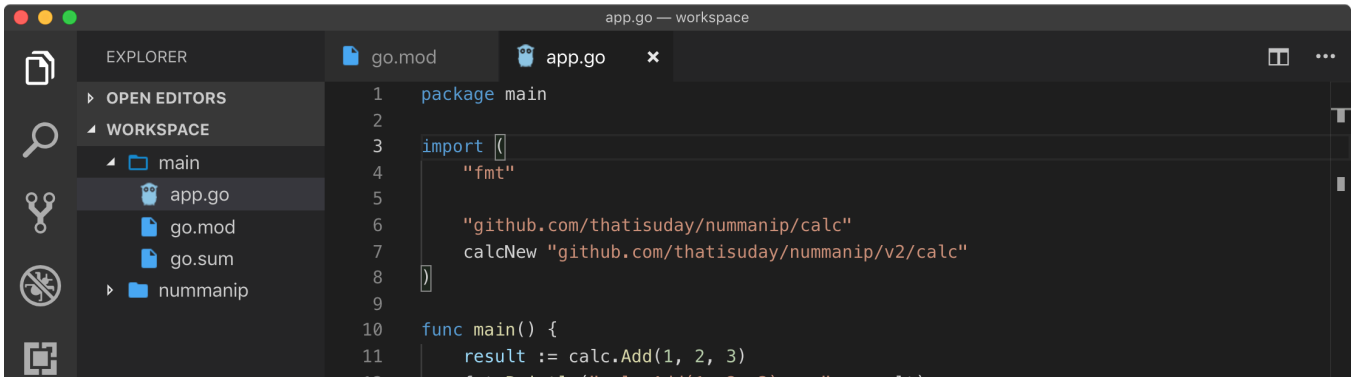
This implementation of the function `Add` will not work with previous code as `Add` in version `1.x.x` used to return only one value. Which means, our code has officially become backward-incompatible (*not something to cheer for though*). Hence, it's fair to assume that the next release version should be `v2.0.0` as it is a different package overall and Go is not wrong about that.

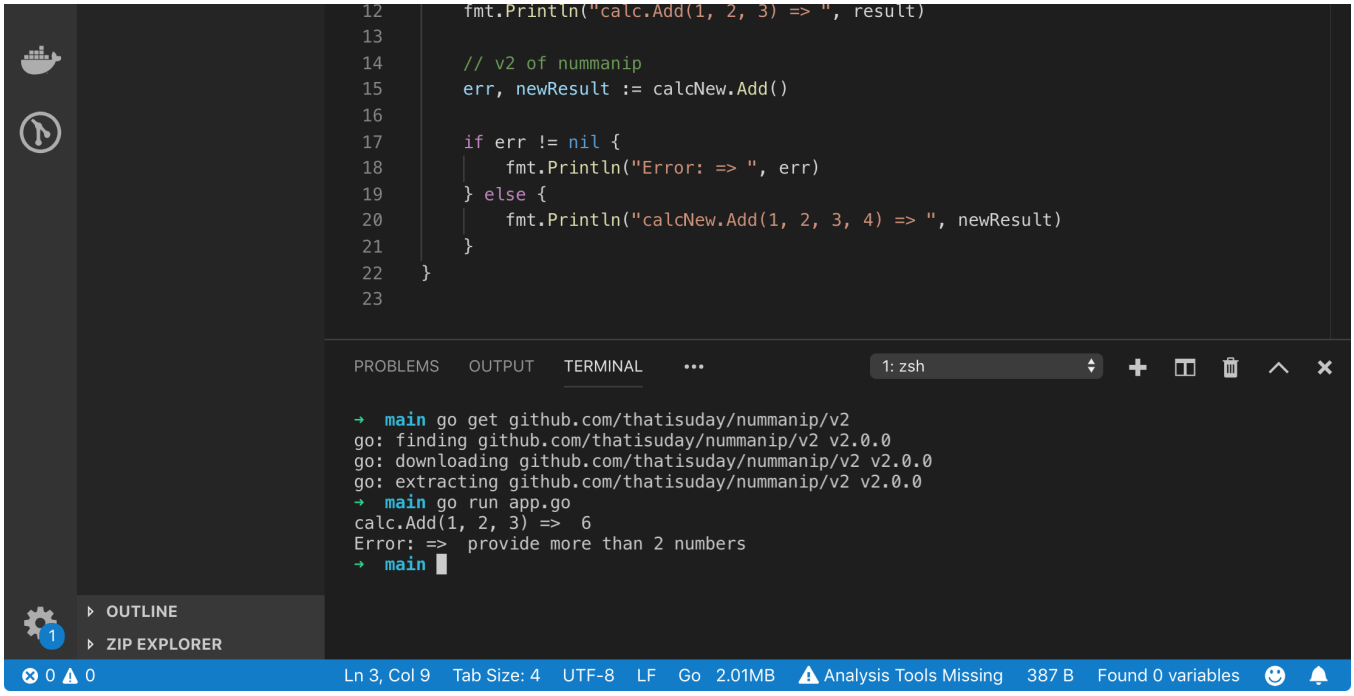
If you have noticed, we created a different branch **v2** for the new major release. This will make thing easier to handle since we need to also support **v1** in case some bugs or enhancement is needed there.

Now the most important thing is to update our `go.mod` file to add version prefix in the module import statement.



Go understands the `vX` syntax and successfully resolve the module import despite this confusing import path. `vX` syntax is not flexible, which means it should be precise with the released **SemVer *major*** version release number, for example, **v2** for **v2.x.x** release.

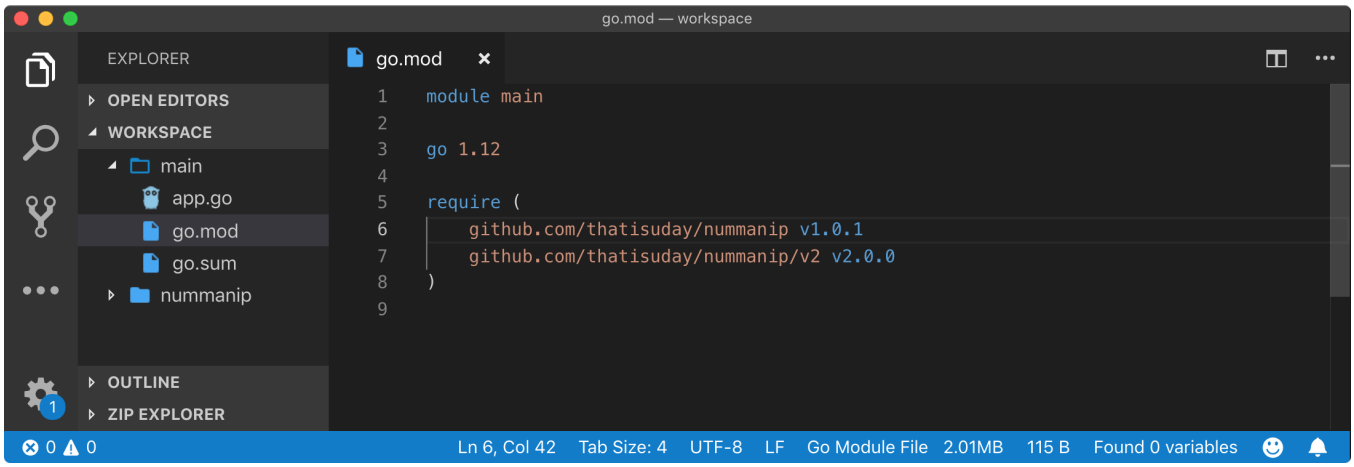




installing v2

In the above example, we have installed a newer major release version of `nummanip` module. Since it's a major release, we need to use `go get` command to manually install it. Since this newer release also has new import syntax, we need to use that too.

The annoying things could be aliasing your package. Since we are importing two different versions of the same package, we need to alias either one of them to resolve the same package name variable conflict. Hence, we aliased v2/calc package to `calcNew` and used it to access `Add` function.



go.mod file

module cache

You can also see that, `go get` command has added module entry to `go.mod` file and saved the newer version of the module inside the cache directory.

Run and Build Executable Module

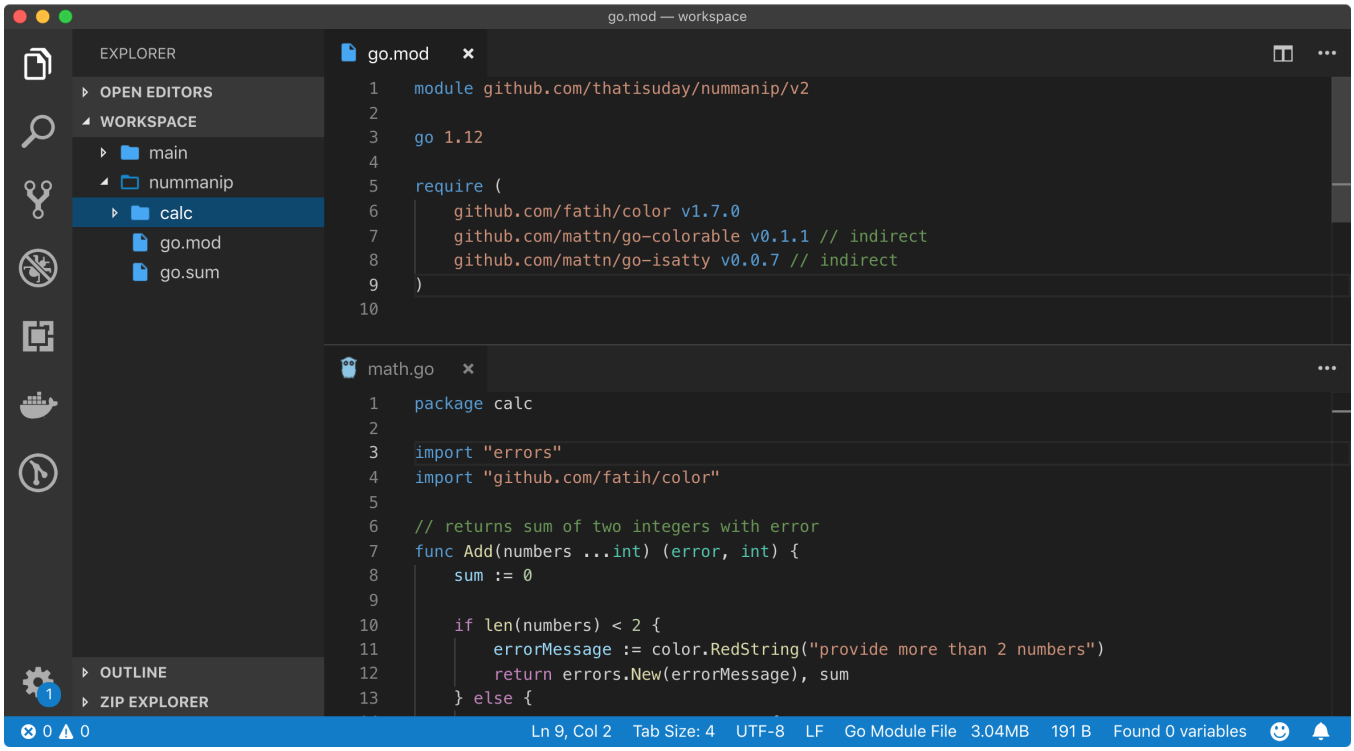
When you are working on an executable module, you can run the module by using `go run <filename>` or `go run path/*.go` which I have explained in my **packages** tutorial. When it comes to build, you can build, you can either use `go build` which builds the module and outputs the binary in the current directory or use `go install` which installs (*outputs*) the module binary inside `$GOPATH/bin` *(which is inside the **PATH**).*

> *When it comes to non-executable modules like `nummanip` module in our case, unlike **pre-go1.11**, you can't create package archives using `go install` command. This is because `go install` can't predict the module version (SemVer) of a module from the Git history or Git tags. Also, Go Modules do not save packages as binary archives as explained in my **packages**tutorial.*

Indirect Dependencies

We have used "**Indirect Dependencies**" term before but haven't explained yet. Well, Indirect Dependencies as they sound are not direct dependencies to our module. Direct dependencies are those which our module imports while indirect dependencies are those which direct dependencies import.

`go.mod` files note down both direct and indirect dependencies and mark them using using trailing comment as shows in the below program.



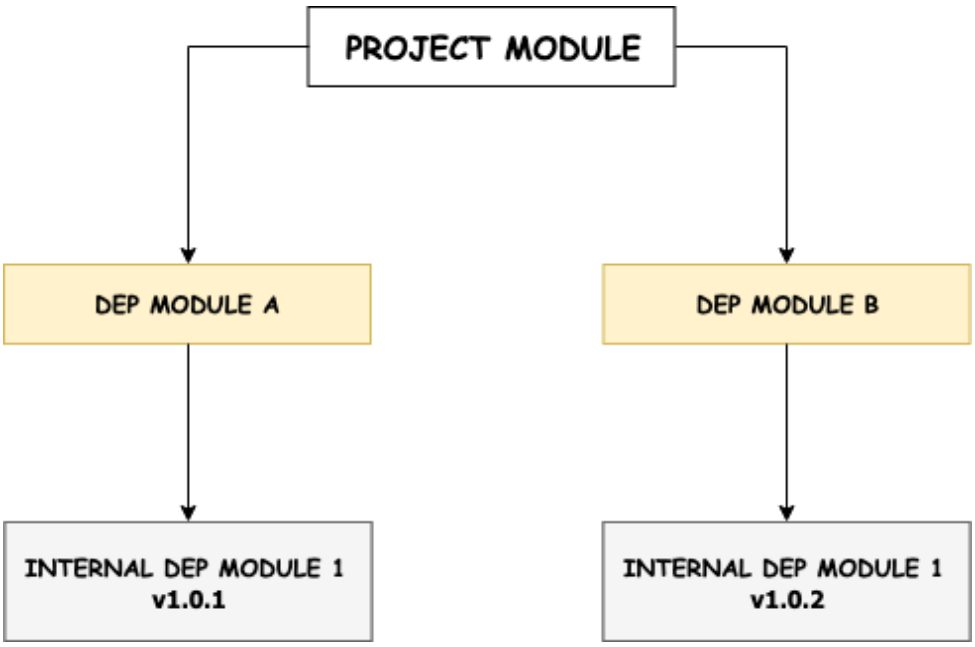
(direct vs indirect dependencies)

From above program, we know that `github.com/fatih/color` is a direct dependencies since we have imported it inside our code. When we ***run*** or ***compile*** the module, Go update `go.mod` file and adds `indirect` comment to the dependencies which are not direct.

I hope this might have cleared a lot of things about Go Modules, but one question remains is that when there is no version suffix is present in the module import declaration (*inside `go.mod` file of the dependency module*), which version Go fetched automatically? It's the latest version of `v1.x.x` because, by default, the version suffix is **v1**.

Minimal Version Selection

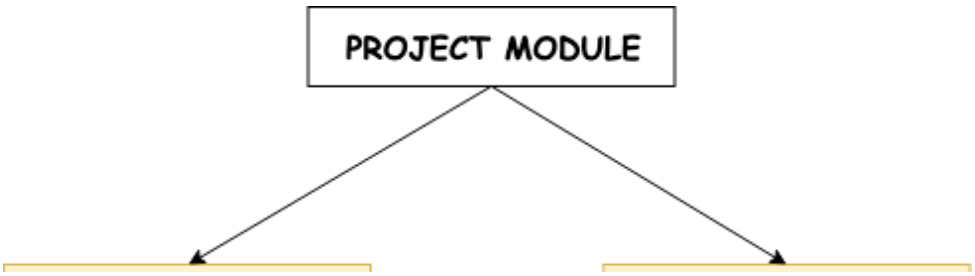
From what we have learned so far, we can import multiple versions of the same dependency modules but as long as they are major release version. There is no way to import multiple modules whose version differ by only **minor** or **patch** version (*because there is no different module import statement for* **minor** *or* **patch** *releases*).

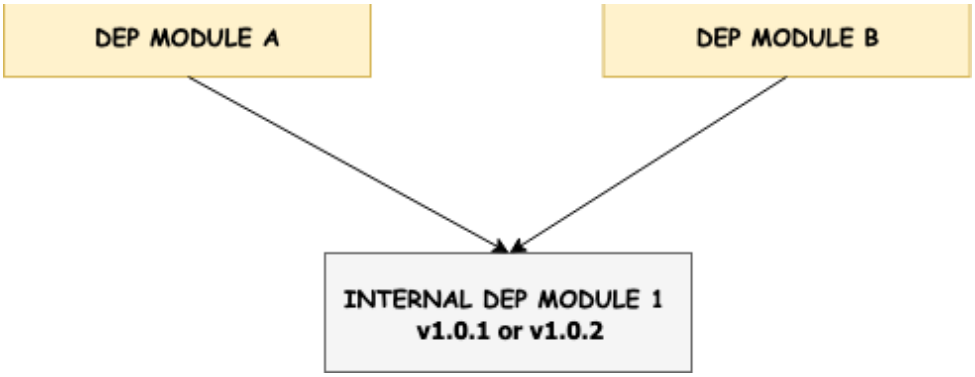


Dependency Copy Problem

Let's say, you have a project module which has dependency modules `A` and `B`. These are completely different modules but they both have a dependency on the same module which is **Module 1**. But the gotcha is, **Module A** requires version `v1.0.1` while **Module B** requires version `v1.0.2`. So each module has defined the **Minimal Version** of the dependency they need in order to work properly. So, if we use `v1.0.1` in the final build, there is a possibility that **Module B** will give unexpected results or fails to build at all.

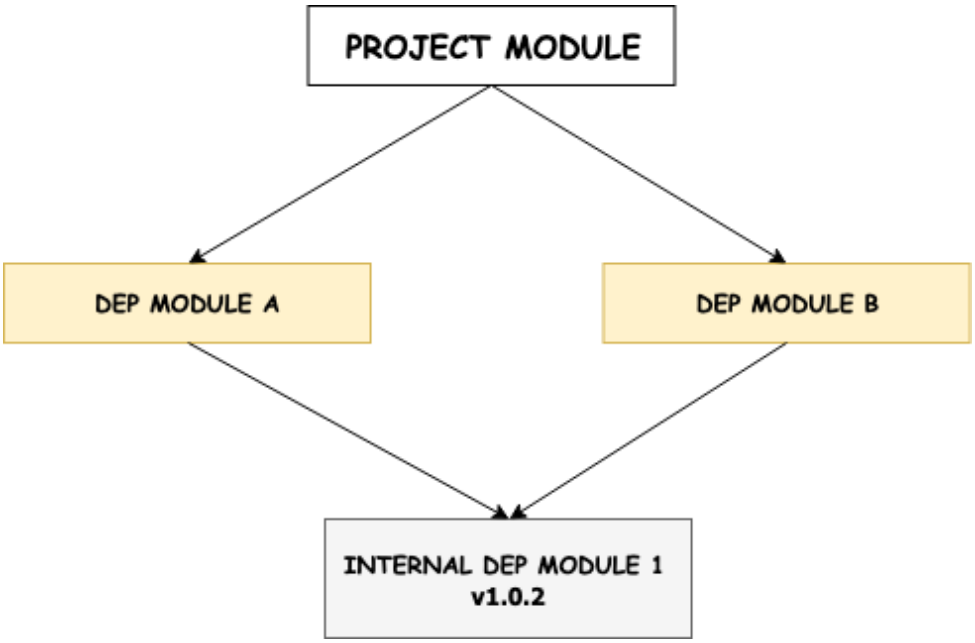
Since in the build, we can deploy only one version of this dependency, we have a ***Diamond Dependency Problem*** *(as shown below).*





Diamond Dependency

As per Go's recommendation, multiple versions of the same dependency with a common ***major*** version should be backward compatible. Hence, if we use `v1.0.2` in our final build, Module 1 will function OK with `v1.0.2` as it contains all the functionality of `v1.0.1`. Go calls this **Minimal Version Selection** (which also means to select **maximal version** between the dependencies). Minimal version selection is **explained here in details**.



Final Verdict with Minimal Version Selection

So finally, we ended up with `v1.0.2` of the dependency **Module 1**.

Go modules are in beta stage and there might be changes in the future. Since I generally do not always get the time

to follow up on these things, please drop a private note on this article if anything new comes up. This article became possible due to a private note about Go Modules only. So, any help is appreciated :)

BONUS TIPS

1. If you have a module which you wish to publish but its `go.mod` file does not track the dependencies of your module's source code has, then you can run `go build ./...` command. `./...` pattern matches all the packages in the module and downloads the dependencies which might not have been downloaded before. This way, you can make sure, `go.mod` file has all the dependencies enlisted before you publish your module.

2. If you think `go.mod` file has some dependencies that your module does not require anymore, then you can use `go mod tidy` command which will remove unwanted dependencies.

3. Sometimes, when you are running automated tests for your project (*executable module like `main` in our case*), there is a chance that your test machine may face some network issues downloading dependencies. In that case, you need to provide dependencies beforehand. This is called **vendoring**. You can use the command `go mod vendor` to output all the dependencies inside `vendor` directory (*in module directory*). When you use `go build`, it looks for the dependencies inside module cache directory but you can force Go to use dependencies from `vendor` directory of the current directory using the command `go build -mod vendor`.

4. `go mod graph` shows you the dependencies of your module.

5. Watch **this amazing video** at GopherCon 2018 to understand **Go Dependency Management with versioning** in simple language.

6. The official Go documentation on Modules is available **here**.