

This documentation will get end users with a basic understanding of coding and programming, familiar with Go Language and utility. Understanding Go Language is a process, yet this guiding document will provide the links and basic knowledge to get any beginner and or intermediary programmer comfortable with and operating Go in a short amount of time.

According to <https://golang.org/>, Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.

Typically, there are Binary distributions of the program available for Linux, macOS, Windows, etc. Go is supported by Google, and is also used by some established companies like IBM, Intel, Medium, and Adobe.

Go is expressive, concise, clean, and efficient. It's concurrency mechanisms make it easy to write programs that get the most out of multicore and networked machines. Its subsequent rudimentary novel type system allows for flexible and modular program construction.

Go compiles rapidly to encode. Go also has the convenience of a [garbage collection](#) and the power of [run-time reflection](#). In its current incarnation, it is a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language.

Download the Program:

Download and install

- I. [Go Download Here](#)
- II. [Go Install Here](#)
- III. [Go Code Here](#)

For other content on installation, you might be interested in:

[Managing Go installations](#) -- How to install multiple versions and uninstall.

[Installing Go from source](#) -- How to check out the sources, build them on your own machine, and run them.

Download and install Go quickly with the steps provided below:

1. Download [here](#). This link leads to a downloads page with more intricate instructions.
 - a. If you do not see your operating system listed, try one of the [other downloads](#).
 - b. **Note:** By default, the go command downloads and authenticates modules using the Go module mirror and Go checksum database run by Google. You can learn more [here](#).
2. Select the tab for your computer's operating system below, then follow its installation instructions.

- a. **Linux:** Extract the archive you downloaded into /usr/local, creating a Go tree in /usr/local/go.
 - i. Important: This step will remove a previous installation at /usr/local/go, if any, prior to extracting. Please back up any data before proceeding.
 - ii. For example, run the following as root or through sudo: `rm -rf /usr/local/go && tar -C /usr/local -xzf go1.14.3.linux-amd64.tar.gz`
 - iii. Add /usr/local/go/bin to the PATH environment variable.
 - iv. You can do this by adding the following line to your \$HOME/.profile or /etc/profile (for a system-wide installation): `export PATH=$PATH:/usr/local/go/bin`
 - v. **Note:** Changes made to a profile file may not apply until the next time you log into your computer. To apply the changes immediately, just run the shell commands directly or execute them from the profile using a command such as `source $HOME/.profile`.
 - vi. Verify that you've installed Go by opening a command prompt and typing the following command: `$ go version`
 - vii. Confirm that the command prints the installed version of Go.
 - b. **MacOS:** Open the package file you downloaded and follow the prompts to install Go.
 - i. The package installs the Go distribution to /usr/local/go. The package should put the /usr/local/go/bin directory in your PATH environment variable. You may need to restart any open Terminal sessions for the change to take effect.
 - ii. Verify that you've installed Go by opening a command prompt and typing the following command: `$ go version`
 - iii. Confirm that the command prints the installed version of Go.
 - c. **Windows:** Open the MSI file you downloaded and follow the prompts to install Go.
 - i. By default, the installer will install Go to Program Files or Program Files (x86). You can change the location as needed. After installing, you will need to close and reopen any open command prompts so that changes to the environment made by the installer are reflected at the command prompt.
 - ii. Verify that you've installed Go.
 - iii. In Windows, click the Start menu.
 - iv. In the menu's search box, type `cmd`, then press the Enter key.
 - v. In the Command Prompt window that appears, type the following command: `$ go version`
 - vi. Confirm that the command prints the installed version of Go.
3. You should be successfully configured and able to operate Go. You can visit the [Getting Started tutorial](#) to write some simple Go code. It takes about 10 minutes to complete.

Getting into Training:

1. After you have successfully downloaded the program, it's not time to get familiar with operating the tool.
2. The tools you will need are as follows:
 - a. Basic programming experience. The code here is simple, however, it helps to know something about functions.
 - b. A code or text editor. Any text editor you have will work. Most text editors have good support for Go. The most popular are VSCode (free), GoLand (paid), and Vim (free).
 - c. A command terminal. Go works well using any terminal on Linux and Mac, and on PowerShell or cmd in Windows.

3. To begin, open a command prompt and cd to your home directory.



```
cd
```

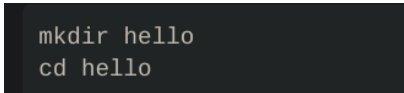
- a. On Linux or Mac:



```
cd %HOMEPATH%
```

- b. On Windows:

4. Next create a hello directory for your first Go source code. For example, use the following commands:



```
mkdir hello  
cd hello
```

5. Enable dependency tracking for your code.
6. When code imports packages contained in other modules, you manage those dependencies through your code's own module. Module is then defined by a go.mod file that tracks the modules that issue the packages. The go.mod file stays with the code, including in the source code in the repository.
7. In order to enable dependency tracking for the code by creating a go.mod file, run the [go mod init command](#), and name it after the module your code will be in. Like other programs, the name is the module's module path. This will be the repository location where your source code will be kept, such as github.com/mymodule. If you plan to publish your module for others to use, the module path must be a location from which Go tools can download your module.
8. For the purposes of this tutorial, just use example.com/hello:

```
$ go mod init example.com/hello
go: creating new go.mod: module example.com/hello
```

9. In a text editor, create a file hello.go in which to write code.
10. Paste the following code into your hello.go file and save the file:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

11. This is your Go code. In this code, you must do the following:
 - a. Declare a main package (a package is a way to group functions, and is made up of all the files in the same directory).
 - b. Import the popular [fmt package](#), which contains functions for formatting text, including printing to the console. This package is one of the [standard library](#) packages included in the Go installation.
 - c. Implement a main function to print a message to the console. A main function executes by default when you run the main package.

```
$ go run .
Hello, World!
```

12. Then run the code:
 - a. The [go run command](#) is one of many go commands you'll use to get things done with Go. Use the following command to get a list of the others:

```
$ go help
```

13. Now you will implement the code in an external package. In the case that you require your code to perform an action that might have been implemented by another programmer, you can look for a package that has functions you can use in your code.
14. You can make your printed message a little more interesting with a function from an external module.
 - a. Visit [pkg.go.dev](#) and [search for a "quote" package](#).
 - b. Locate and click the [rsc.io/quote](#) package in search results, if you see [rsc.io/quote/v3](#), ignore it for now.
 - c. In the Documentation section, under Index, record the list of functions you can call from your code. You'll use the `Go` function.

- d. As mentioned earlier and listed atop the documentation, please observe that the package *quote* is included in the *rsc.io/quote* module.
 - e. You can use the pkg.go.dev site to find published modules whose packages have functions you can use in your own code. Packages are published in modules like: *rsc.io/quote*. This is where other programmers can use and share modules. Modules are improved with newer versions over time, and you can upgrade your code to use the improved versions via this function as well.
15. In your Go code, import the *rsc.io/quote* package and add a call to its Go function. After adding the highlighted lines, your code should display:

```
package main

import "fmt"

import "rsc.io/quote"

func main() {
    fmt.Println(quote.Go())
}
```

16. Now you must add new module requirements and sums. Go will add the *quote* module as a requirement, as well as a *go.sum* file for use in authenticating the module. For more information, see [Authenticating modules](#) in the Go Modules Reference:

```
$ go mod tidy
go: finding module for package rsc.io/quote
go: found rsc.io/quote in rsc.io/quote v1.5.2
```

17. Then run your code to see the message:

```
$ go run .
Don't communicate by sharing memory, share memory by communicating.
```

18. You should notice that your code calls the *Go* function, issuing a decisive message about communication. When you ran the *go mod tidy*, it located and downloaded the *rsc.io/quote* module that contains the package you imported. By default, it downloaded the latest version -- *v1.5.2*.

19. Feel free to experiment and author more code. With this quick introduction, you got Go installed and learned some of the basics. To write some more code with another tutorial, take a look at [Create a Go module](#)

20. You are now familiar with Go! You may tour and use the practice module at the [Go Tour](#) page.

Below is a useful Glossary of Go Terms and other programmer terminology:

[Glossary:](#)

I. build constraint: A condition that determines whether a Go source file is used when compiling a package. Build constraints may be expressed with file name suffixes (for example, `foo_linux_amd64.go`) or with build constraint comments (for example, `// +build linux,amd64`).

II. build list: The list of module versions that will be used for a build command such as `go build`, `go list`, or `go test`. The build list is determined from the main module's `go.mod` file and `go.mod` files in transitively required modules using minimal version selection. The build list contains versions for all modules in the module graph, not just those relevant to a specific command.

III. canonical version: A correctly formatted version without a build metadata suffix other than `+incompatible`. For example, `v1.2.3` is a canonical version, but `v1.2.3+meta` is not.

IV. current module: Synonym for main module.

V. deprecated module: A module that is no longer supported by its authors (though major versions are considered distinct modules for this purpose). A deprecated module is marked with a deprecation comment in the latest version of its `go.mod` file.

VI. direct dependency: A package whose path appears in an import declaration in a `.go` source file for a package or test in the main module, or the module containing such a package. (Compare indirect dependency.)

VII. direct mode: A setting of environment variables that causes the `go` command to download a module directly from a version control system, as opposed to a module proxy. `GOPROXY=direct` does this for all modules. `GOPRIVATE` and `GONOPROXY` do this for modules matching a list of patterns.

VIII. go.mod file: The file that defines a module's path, requirements, and other metadata. Appears in the module's root directory.

IX. import path: A string used to import a package in a Go source file. Synonymous with package path.

X. indirect dependency: A package transitively imported by a package or test in the main module, but whose path does not appear in any import declaration in the main module; or a module that appears in the module graph but does not provide any package directly imported by the main module. (Compare direct dependency.)

XI. main module: The module in which the go command is invoked. The main module is defined by a go.mod file in the current directory or a parent directory.

XII. major version: The first number in a semantic version (1 in v1.2.3). In a release with incompatible changes, the major version must be incremented, and the minor and patch versions must be set to 0. Semantic versions with major version 0 are considered unstable.

XIII. major version subdirectory: A subdirectory within a version control repository matching a module's major version suffix where a module may be defined. For example, the module example.com/mod/v2 in the repository with root path example.com/mod may be defined in the repository root directory or the major version subdirectory v2. See Module directories within a repository.

XIV. major version suffix: A module path suffix that matches the major version number. For example, /v2 in example.com/mod/v2. Major version suffixes are required at v2.0.0 and later and are not allowed at earlier versions. See the section on Major version suffixes.

XV. minimal version selection (MVS): The algorithm used to determine the versions of all modules that will be used in a build. See the section on Minimal version selection for details.

XVI. minor version: The second number in a semantic version (2 in v1.2.3). In a release with new, backwards compatible functionality, the minor version must be incremented, and the patch version must be set to 0.

XVII. module: A collection of packages that are released, versioned, and distributed together.

XVIII. module cache: A local directory storing downloaded modules, located in GOPATH/pkg/mod.

XIX. module graph: The directed graph of module requirements, rooted at the main module. Each vertex in the graph is a module; each edge is a version from a require statement in a go.mod file (subject to replace and exclude statements in the main module's go.mod file).

XX. module path: A path that identifies a module and acts as a prefix for package import paths within the module. For example, "golang.org/x/net".

XXI. module proxy: A web server that implements the GOPROXY protocol. The go command downloads version information, go.mod files, and module zip files from module proxies.

XXII. module root directory: The directory that contains the go.mod file that defines a module.

XXIII. module subdirectory: The portion of a module path after the repository root path that indicates the subdirectory where the module is defined. When non-empty, the module subdirectory is also a prefix for semantic version tags. The module subdirectory does not include the major version suffix, if there is one, even if the module is in a major version subdirectory.

XXIV. package: A collection of source files in the same directory that are compiled together. See the Packages section in the Go Language Specification.

XXV. package path: The path that uniquely identifies a package. A package path is a module path joined with a subdirectory within the module. For example "golang.org/x/net/html" is the package path for the package in the module "golang.org/x/net" in the "html" subdirectory. Synonym of import path.

XXVI. patch version: The third number in a semantic version (3 in v1.2.3). In a release with no changes to the module's public interface, the patch version must be incremented.

XXVII. pre-release version: A version with a dash followed by a series of dot-separated identifiers immediately following the patch version, for example, v1.2.3-beta4. Pre-release versions are considered unstable and are not assumed to be compatible with other versions. A pre-release version sorts before the corresponding release version: v1.2.3-pre comes before v1.2.3. See also release version.

XXVIII. pseudo-version: A version that encodes a revision identifier (such as a Git commit hash) and a timestamp from a version control system. For example, v0.0.0-20191109021931-daa7c04131f5. Used for compatibility with non-module repositories and in other situations when a tagged version is not available.

XXIX. release version: A version without a pre-release suffix. For example, v1.2.3, not v1.2.3-pre.

XXX. repository root path: The portion of a module path that corresponds to a version control repository's root directory.

XXXI. retracted version: A version that should not be depended upon, either because it was published prematurely or because a severe problem was discovered after it was published.

XXXII. semantic version tag: A tag in a version control repository that maps a version to a specific revision.

XXXIII. selected version: The version of a given module chosen by minimal version selection. The selected version is the highest version for the module's path found in the module graph.

XXXVIII. vendor directory: A directory named vendor that contains packages from other modules needed to build packages in the main module. Maintained with go mod vendor.

XXXV. version: An identifier for an immutable snapshot of a module, written as the letter v followed by a semantic version.

Resources: [Go Docs](#), [Go Installation for OS](#), [Go Tour](#), [Learning Go Programming](#), [Define Programming Garbage Collection](#), [Define Reflection Programming](#)

Tags: downloadgo, golang, golangue, gotraining, goforbeginners, goprogram, gotutorial, linux, mac, macos, windows, windowsos, goforlinux, goformac, goformacos, goforwindows, goforwindowsos, module, syntax, texteditor