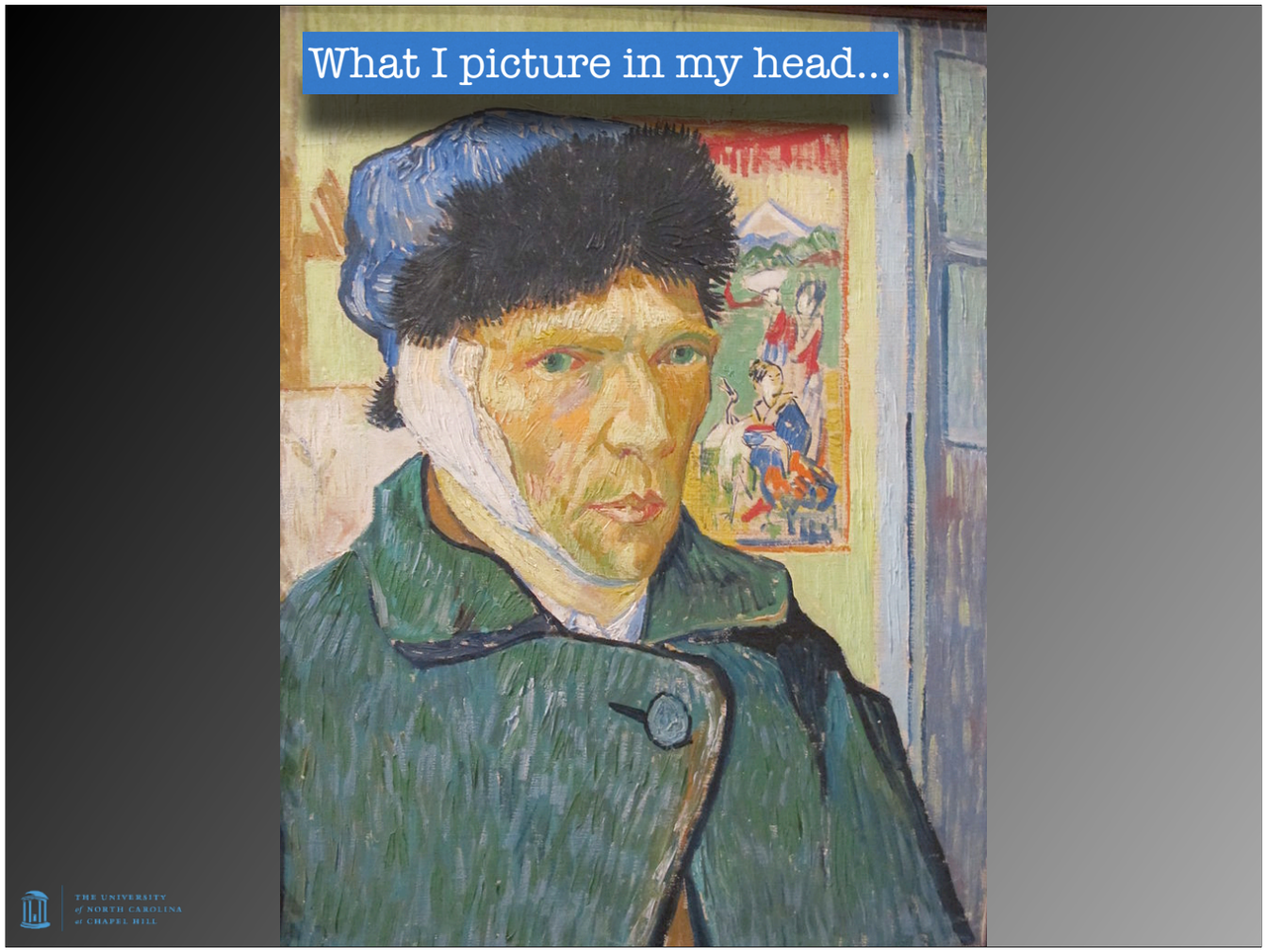
I have 20 minutes to rip through this, and I see that I am standing between you and lunch. So strap in. Here we go…

Good morning. My name is…

First, a message from our sponsor...



When I work on presentations, this is what I picture in my head.

Unfortunately, THIS is what it typically ends up looking like.

Just managing expectations.

Who am I?

Frank Seesink

- Senior Network Engineer, UNC Chapel Hill
- Part of network DevOps group
- Involved in network automation for years
- Love languages, both human & computer
- Programming since I was 12 years old
- Formally B.S. in Computer Science with all coursework for an M.S. in C.S.
- JOAT - databases, OSes, networking,...

That reminds me.
For anything useful, credit goes to UNC Chapel Hill for allowing me to attend.
For any mistakes/errors/etc., that all falls on me.

Story Time...

In 2022 I taught myself & fell in love with Go.

At TechEx 2023 I did a session titled “When You are Ready to GO Beyond PYTHON” explaining the “WHY”.
For full details, see

https://frank.seesink.com/presentations/Internet2TechEx-Fall2023/

This session is intended to cover the “HOW”.

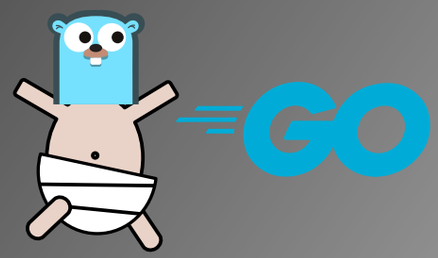
TL;DR of “Why”

- Compilation vs. interpretation
  - performance
  - single binary executable
  - no external dependencies
  - cross-compile to other OS/architectures
- Static typing / Inference typing
- Concurrency
- Go benefits from 30+ years of observations into what makes an effective language

Created at Google in the mid-2000s by many of the same folks behind the C programming language, Go benefits from more than 30 years of observations into what makes an effective language. From a very fast compiler allowing for quick iteration during development (very much like Python) while providing all the benefits of a compiled language such as static type checking/etc., to the built-in concurrency support and module management setup, Go offers the “sweet spot” between interpreted languages like Python and low level compiled languages such as C and Rust.

History of Programming Languages

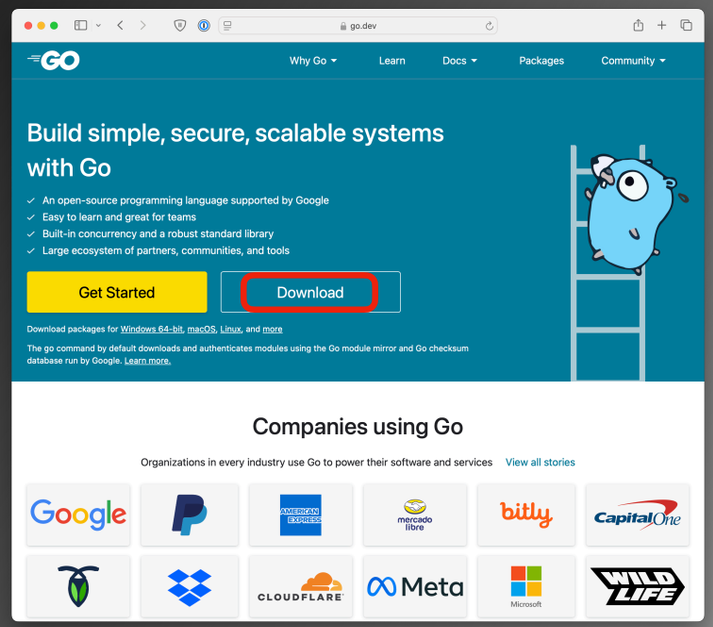
C



Getting Started with

Go.dev

Option #1



https://go.dev/

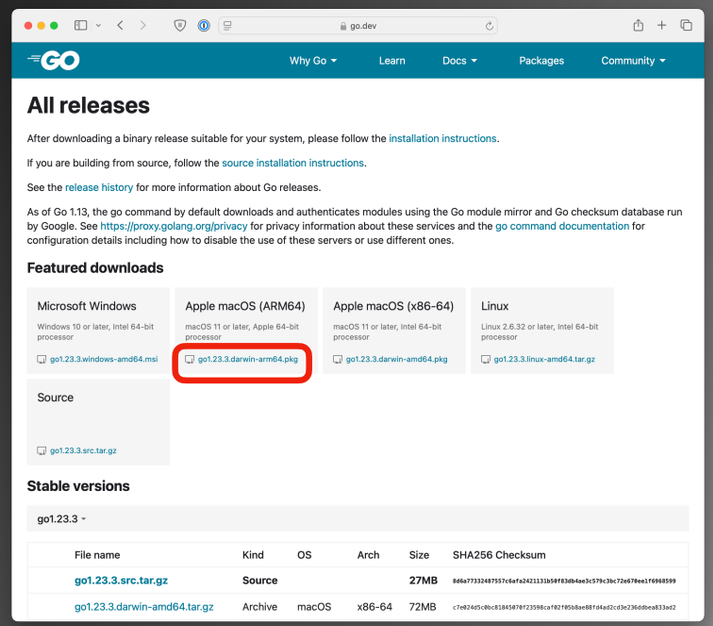
THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

Now in order to program in Go, you need to have the Go compiler installed. Much as I did in my other 2023 TechEx session “Network Automation Tapas - Getting Started with Python”, let’s quickly cover getting Go installed on the various OSes.

The official Go site is Go.dev, and from here you simply click on the “Download” button to find your installer.

Go.dev

Option #1



mac OS

https://go.dev/

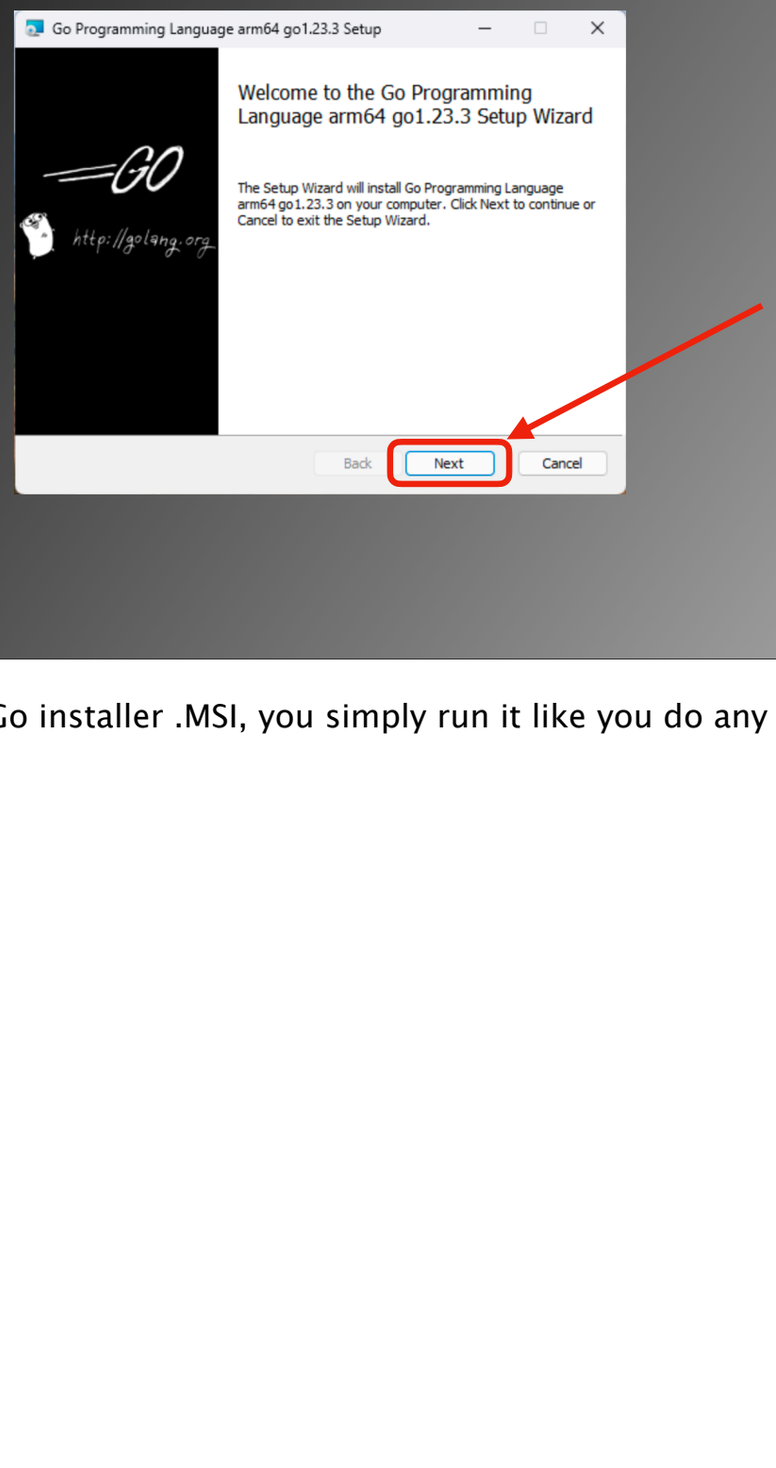
THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

Here you can see that they offer installers for all the major OSes (and quite a few minor ones as well). Each installer is aimed at a particular OS/architecture combination. I have an M3 MacBook Pro, so here you can see I have highlighted the macOS ARM64 installer package.

However, if you don't see your particular OS/arch at the top, simply scroll down a little and you will find plenty more. For example, if you wanted to install the Go compiler on a Raspberry Pi, here is the Linux ARM64 installer. And even further down, if you click on "Other Ports", you will be taken to yet more installers, including ones for FreeBSD, NetBSD, OpenBSD, the Windows ARM64 installer, and even versions that run on the RISC V chip. We will cover this cross-compilation aspect a bit more later.

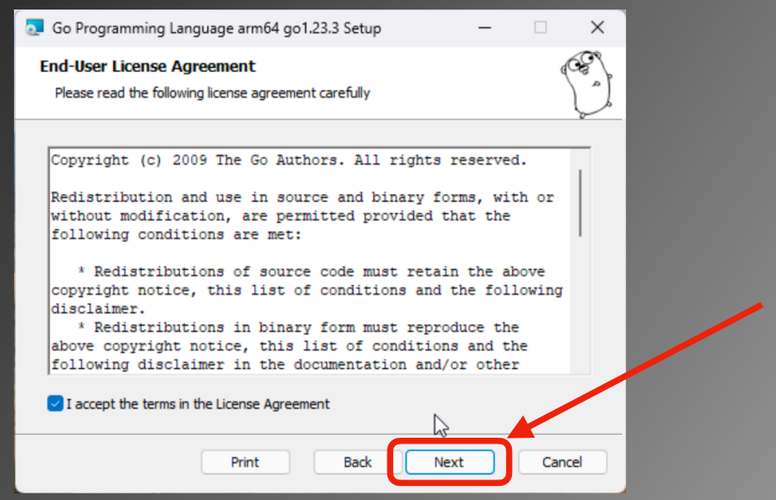

Installing GO
for Windows
THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

Install GO - Windows

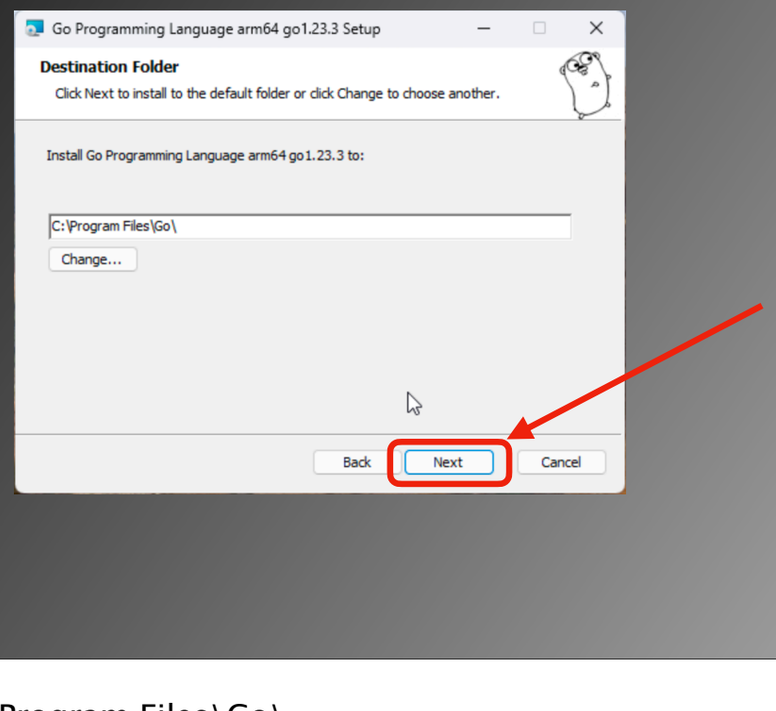


To install the Go compiler on Windows using the official Go installer .MSI, you simply run it like you do any other Windows installer. The steps here are pretty self-explanatory.

Install GO - Windows

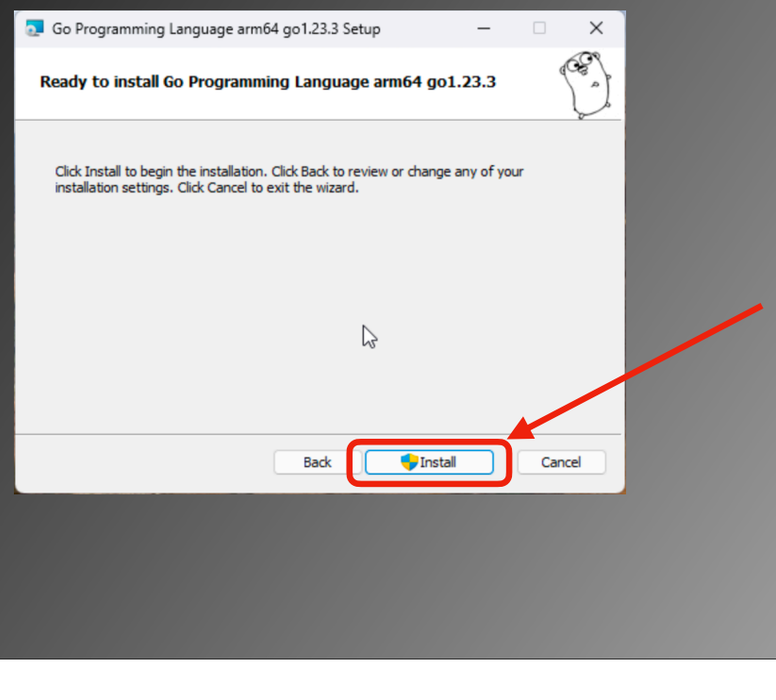


Install Go - Windows



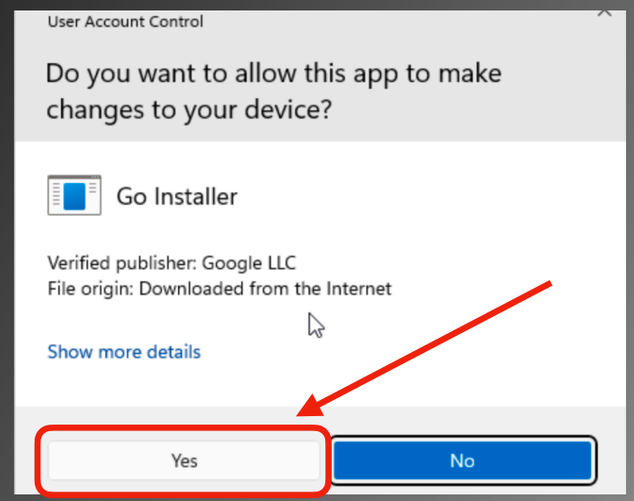
Notice that the default location for the Go compiler is C:\Program Files\Go\.

Install GO - Windows



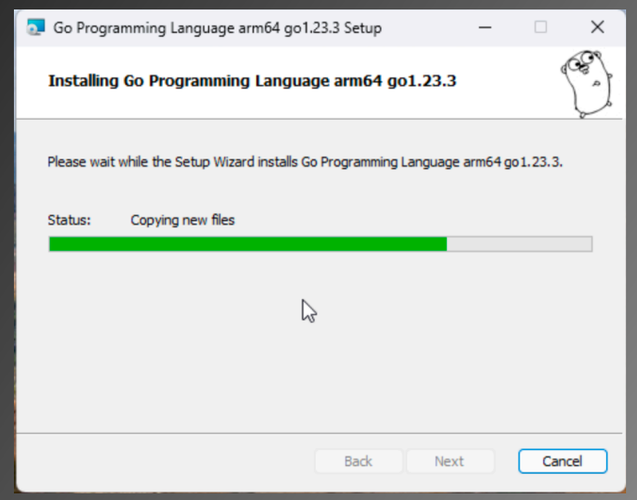
At this point you should notice the shield symbol, indicating that you'll need to have admin level access to install.

Install GO - Windows

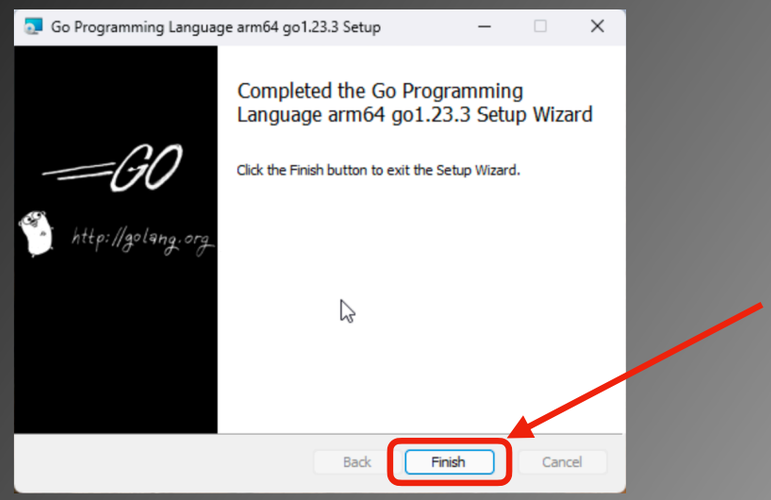


Confirm that YES, you want to allow this app to make changes.

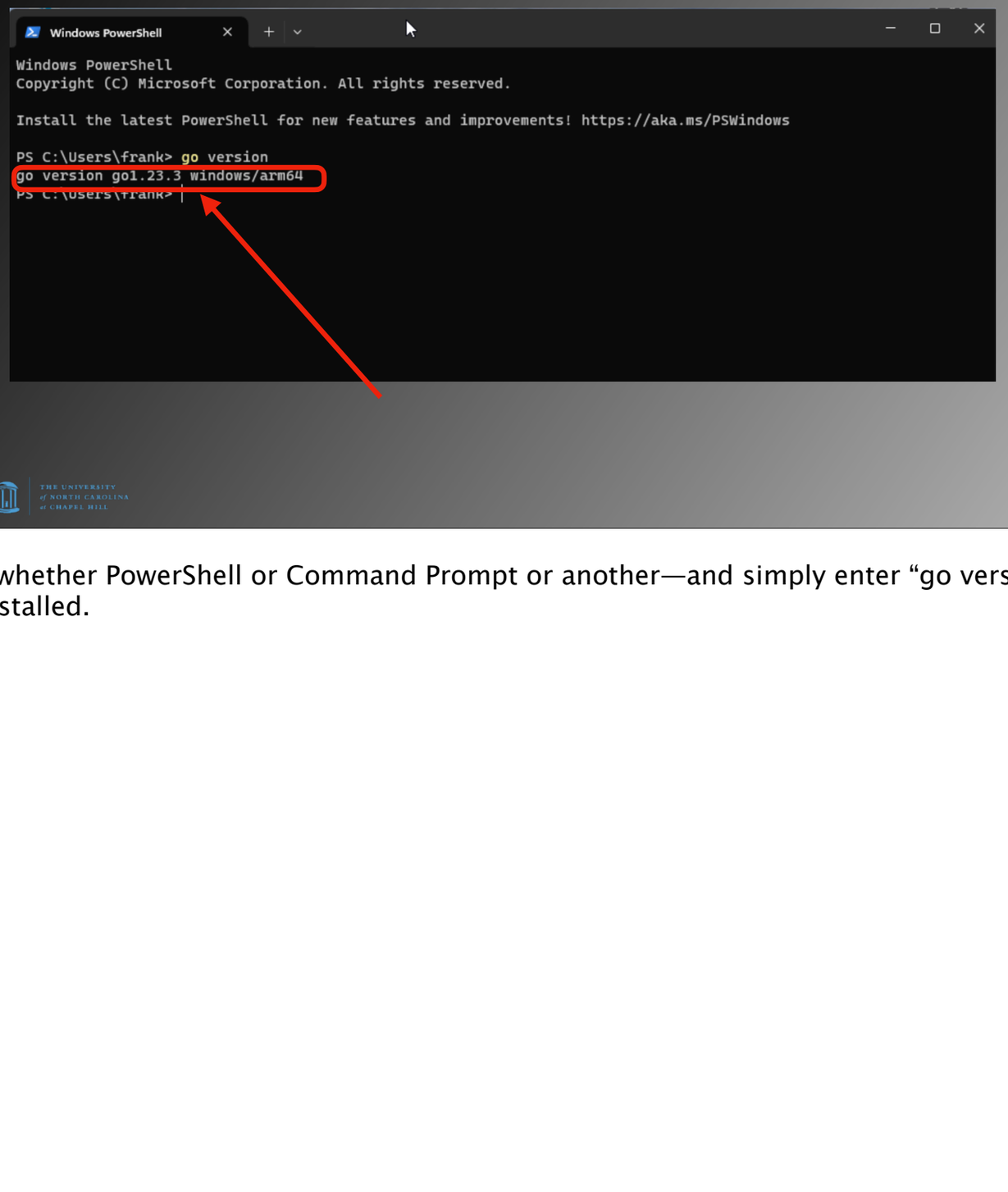
Install GO - Windows



Install GO - Windows



Install GO - Windows



Once done, you can open a terminal—whether PowerShell or Command Prompt or another—and simply enter “go version” to see whether you have access to the go binary and what version is installed.

Install  - Windows

Go.dev Windows Installer installs Go by default in

```
C:\Program Files\Go\
```

Go modules (e.g., seen using **`go get <module>`**) are located in

```
C:\Users\<user>\go\pkg\
```

Go apps (e.g., seen using **`go install <app>`**) are located in

```
C:\Users\<user>\go\bin\
```

If, like me, you like to know where programs put their files, I provide this just as a quick reference should you ever need to go in and remove the setup. I will try to show this for every installation approach.

Install GO - Windows

Option #2: Chocolatey

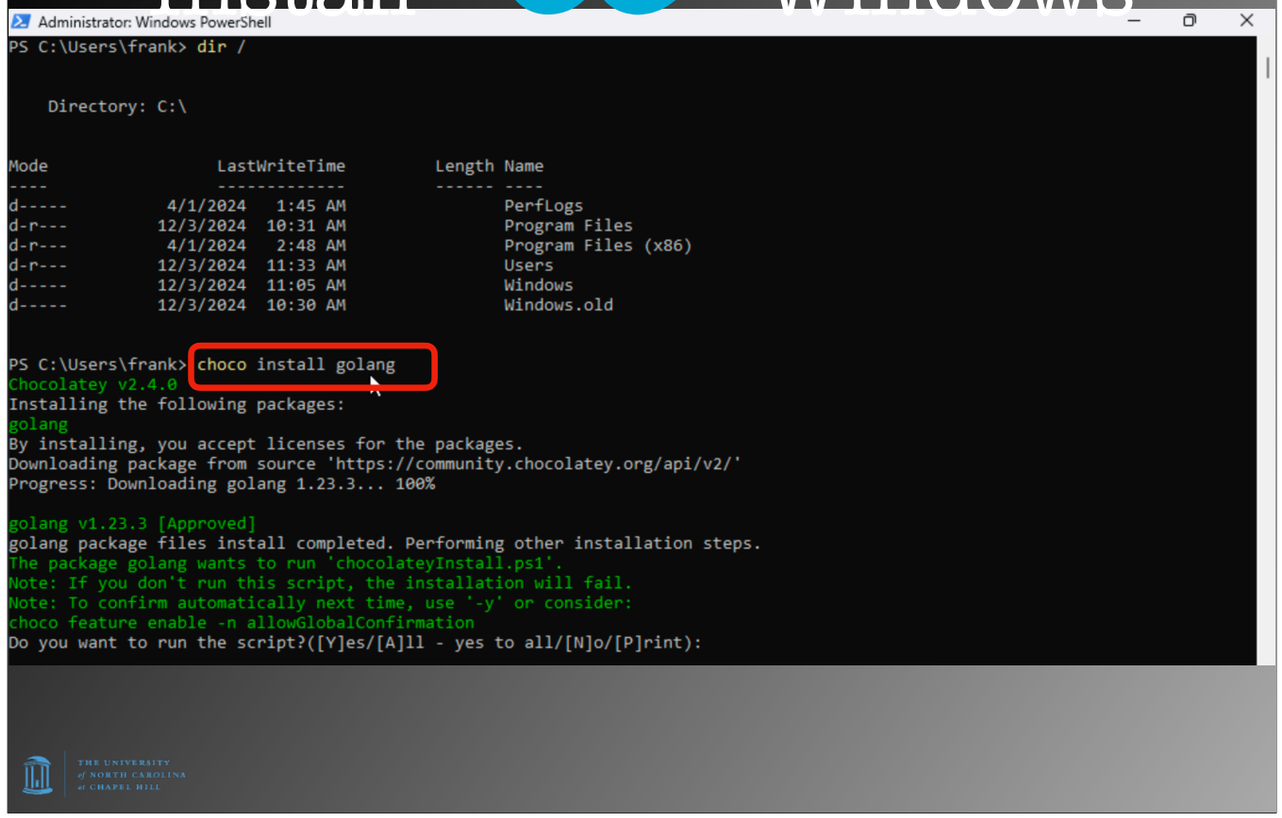
The Package Manager for Windows

https://chocolatey.org/

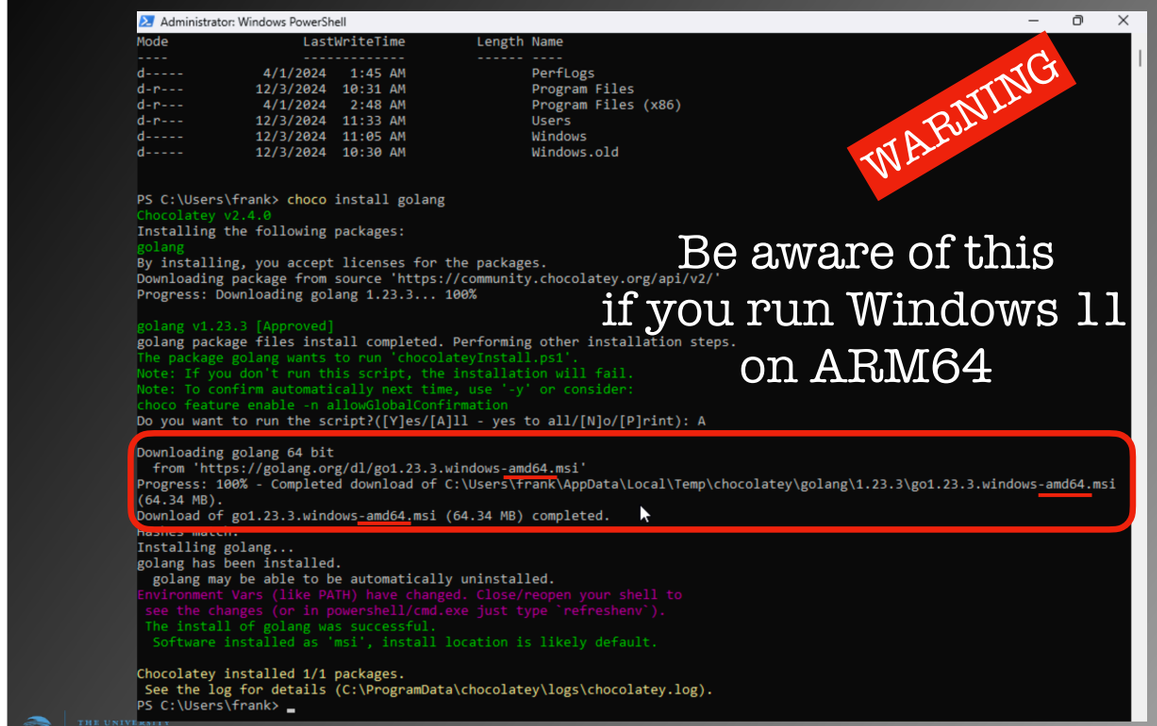
Simply open PowerShell as an administrative shell (i.e., “Run as Administrator”) and enter

```
choco install golang
```

Install GO - Windows



Install GO - Windows



Be warned if you are running Windows 11 ARM64—as I was doing for taking these screenshots—that, at least as of 3 Dec 2024, Chocolatey appears to be downloading/installing the x64 version of the Go compiler, NOT the ARM64! This in turn means you will be running your Go compiler through Windows' emulation layer. So not ideal. I do not recommend Chocolatey at this time.

Install  - Windows

Chocolatey uses Go.dev installer, so also installs Go in

```
C:\Program Files\Go\
```

Go modules (e.g., seen using **`go get <module>`**) are located in

```
C:\Users\<user>\go\pkg\
```

Go apps (e.g., seen using **`go install <app>`**) are located in

```
C:\Users\<user>\go\bin\
```

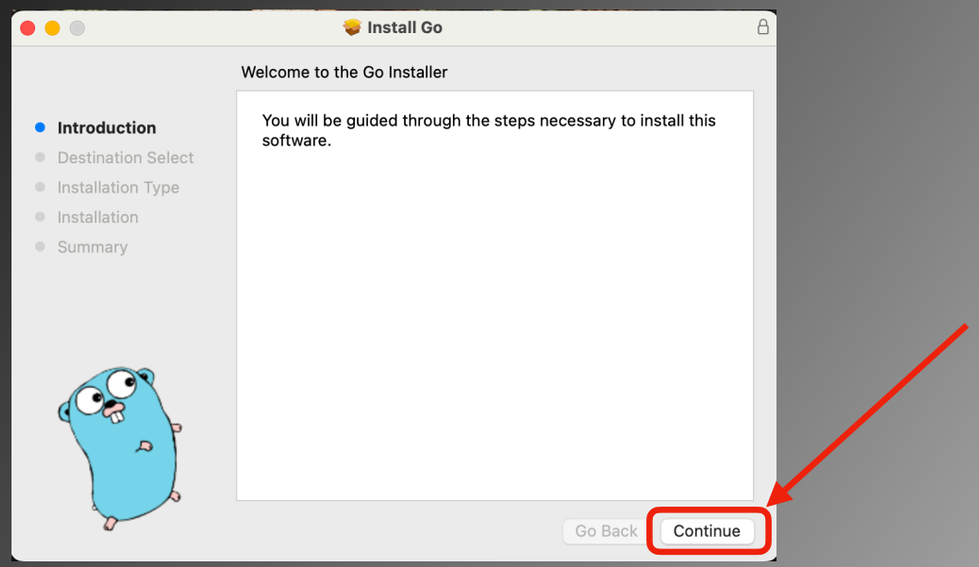
THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

Since Chocolatey simply uses the official Go.dev installers behind the scenes, the paths are the same.

Installing GO

for macOS

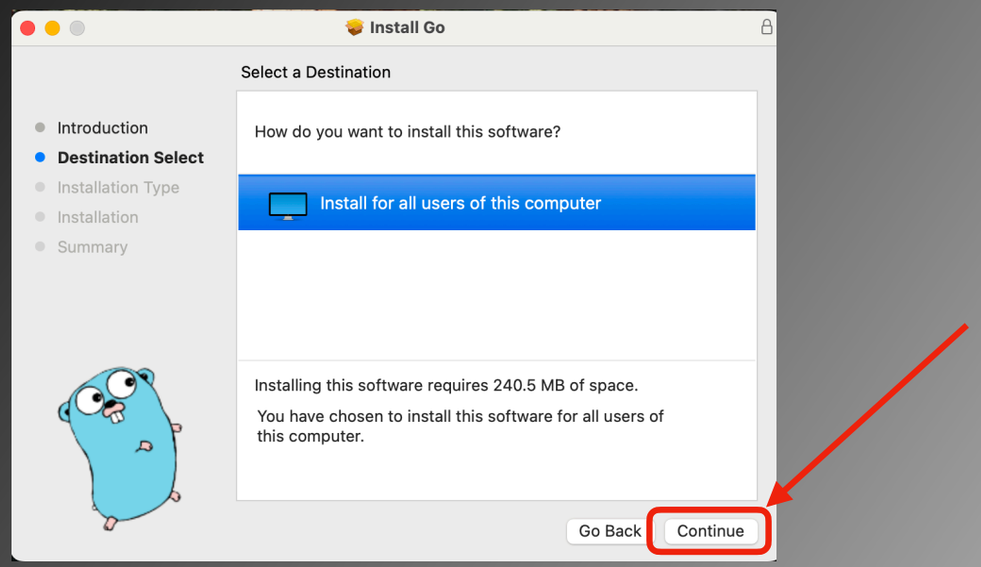
Install GO - macOS



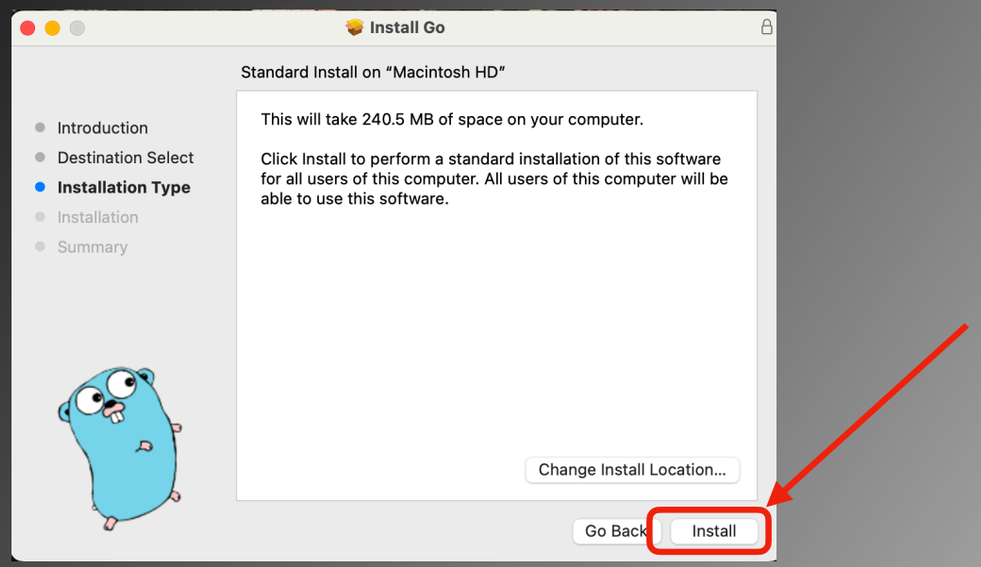
THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

For macOS, this is also your typical .PKG installer.

Install GO - macOS

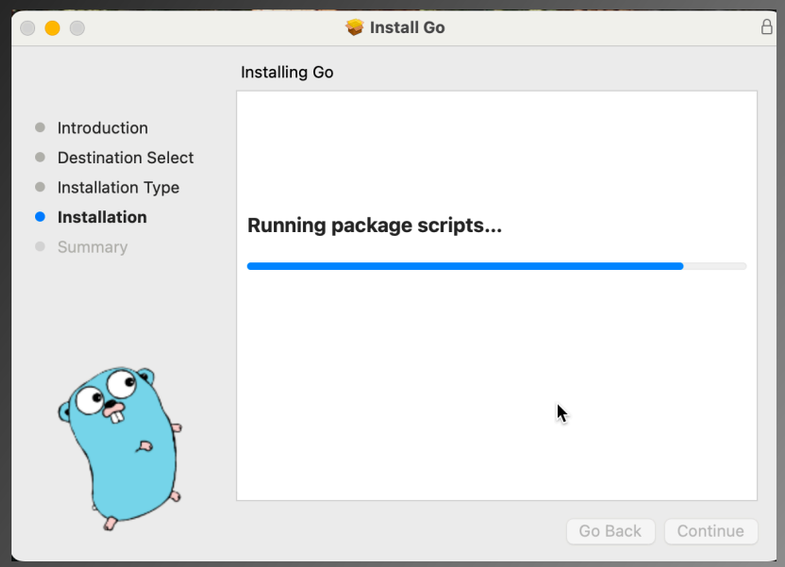


Install GO - macOS

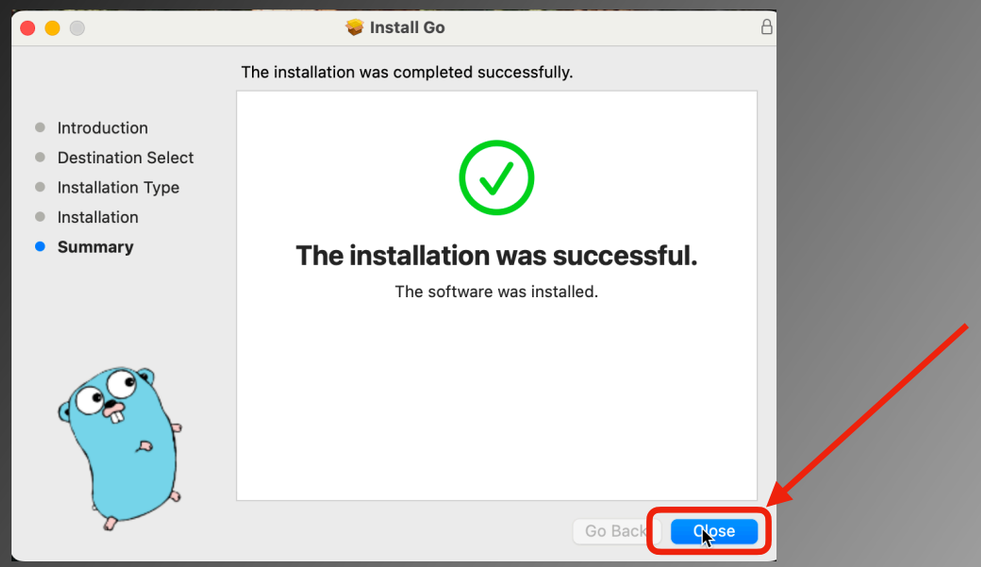


Here be sure to enter your Mac user password so the installation can proceed.

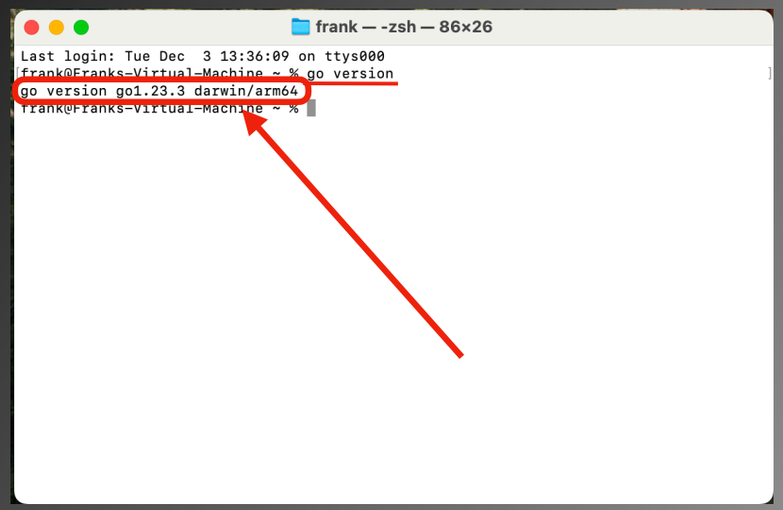
Install GO - macOS



Install GO - macOS



Install GO - macOS



THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

Once finished, simply open Terminal (or whichever terminal program you use such as iTerm2, Wave, etc.) and enter “go version” to see if the Go compiler is installed and what version it is.

Install GO - macOS

Go.dev macOS Installer installs Go in

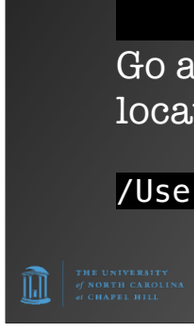
```
/usr/local/go/
```

Go modules (e.g., seen using **`go get <module>`**) are located in

```
/Users/<user>/go/pkg/
```

Go apps (e.g., seen using **`go install <app>`**) are located in

```
/Users/<user>/go/bin/
```



Since macOS is basically UNIX, you can expect to find the Go files in the usual places. The one thing to note is that as you download Go packages and/or apps (similar to using ‘pip install’ for Python), these are placed in a “go” directory within your user’s home directory. Should you ever truly need to clean house, simply deleting this directory removes everything user-specific that you have for Go.

Install GO - macOS

Option #2: Homebrew

The Missing Package Manager for macOS (or Linux)

https://brew.sh/

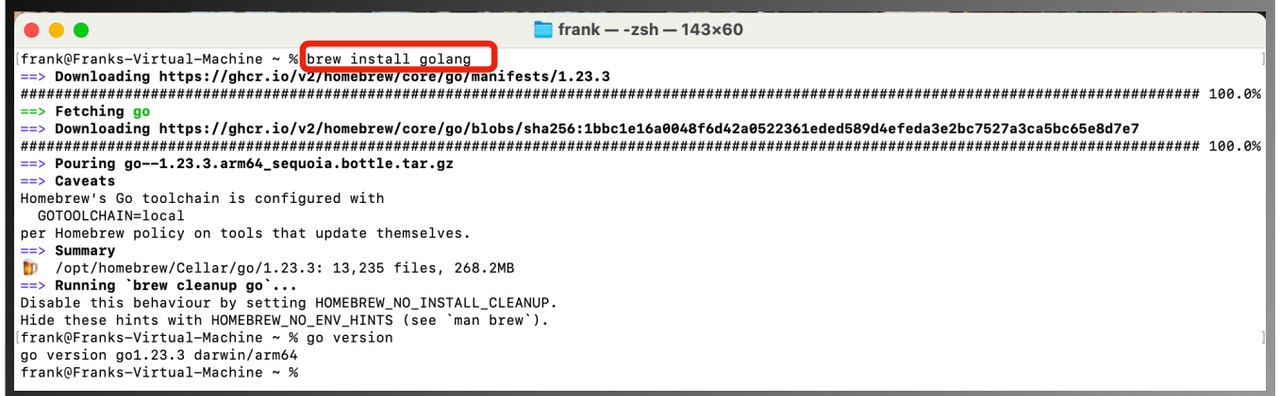
Simply open Terminal and enter

```
brew install golang
```

[SIDE NOTE: Simply having Homebrew installed provides you with a version of Python3 (it comes with the XCode Command Line Tools that Homebrew installs).]

Though I am not a Homebrew user, the steps for installing the Go compiler with Homebrew are pretty straightforward.

Install GO - macOS



Install GO - macOS

Homebrew macOS Installer installs Go in

```
/opt/homebrew/bin/go/
```

Go modules (e.g., seen using **`go get <module>`**) are located in

```
/Users/<user>/go/pkg/
```

Go apps (e.g., seen using **`go install <app>`**) are located in

```
/Users/<user>/go/bin/
```

Here the only thing to note is that the Homebrew version of the Go compiler installs under Homebrew's directory.

Install GO - macOS

Option #3: MacPorts

An open-source community initiative to design an easy-to-use system for compiling, installing, and upgrading either command-line, X11 or Aqua based open-source software on the Mac operating system

https://www.macports.org/

To install Go, simply open Terminal and enter

```
sudo port install go
```

THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

For those who use MacPorts, installing the Go compiler is also quite easy.

Install GO - macOS

frank — -zsh — 125×40

```
frank@Franks-Virtual-Machine ~ % sudo port install go
Password:
--->  Fetching archive for go
--->  Attempting to fetch go-1.23.3_0.darwin_24.arm64.tbz2 from https://packages.macports.org/go
--->  Attempting to fetch go-1.23.3_0.darwin_24.arm64.tbz2 from https://mirrors.mit.edu/macports/packages/go
--->  Attempting to fetch go-1.23.3_0.darwin_24.arm64.tbz2 from http://bos.us.packages.macports.org/go
--->  Fetching distfiles for go
--->  Attempting to fetch go1.23.3.src.tar.gz from https://distfiles.macports.org/go
--->  Attempting to fetch go1.23.3.darwin-arm64.tar.gz from https://distfiles.macports.org/go
--->  Attempting to fetch go1.23.3.darwin-arm64.tar.gz from https://storage.googleapis.com/golang/
--->  Verifying checksums for go
--->  Extracting go
--->  Configuring go
--->  Building go
--->  Staging go into destroot
--->  Installing go @1.23.3_0
--->  Activating go @1.23.3_0
--->  Cleaning go
--->  Scanning binaries for linking errors
--->  No broken files found.
--->  No broken ports found.
frank@Franks-Virtual-Machine ~ % go version
go version go1.23.3 darwin/arm64
frank@Franks-Virtual-Machine ~ %
```

Install GO - macOS

MacPorts installs Go in

```
/opt/local/lib/go/
```

(with symlinks in /opt/local/bin/ to **go** and **gofmt**).

Go modules (e.g., seen using **go get <module>**) are located in

```
/Users/<user>/go/pkg/
```

Go apps (e.g., seen using **go install <app>**) are in

```
/Users/<user>/go/bin/
```

THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

This was one of the more unique installations, in that the Go compiler/etc. were installed down under /opt/local/lib/, and then symlinks were created in /opt/local/bin/ that pointed to the “go” and “gofmt” binaries.

Installing GO

for Linux

THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

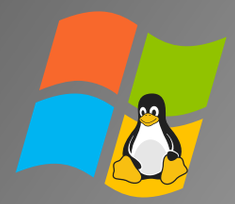
Install GO - Linux

RHEL/CENTOS/Rocky/Alma Linux

```
rpm/yum/dnf install golang
```

Ubuntu/Debian Linux

```
apt install golang
```



WSL

THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

The easiest way to install the Go compiler on Linux is simply to download and decompress the respective .tar.gz file to your setup. That should go into the usual “/usr/local/go” directory. That said, if you prefer using a package manager, the respective ones work just fine, though you will often find that the version of Go on the repositories can be a bit behind.

Install GO - Linux

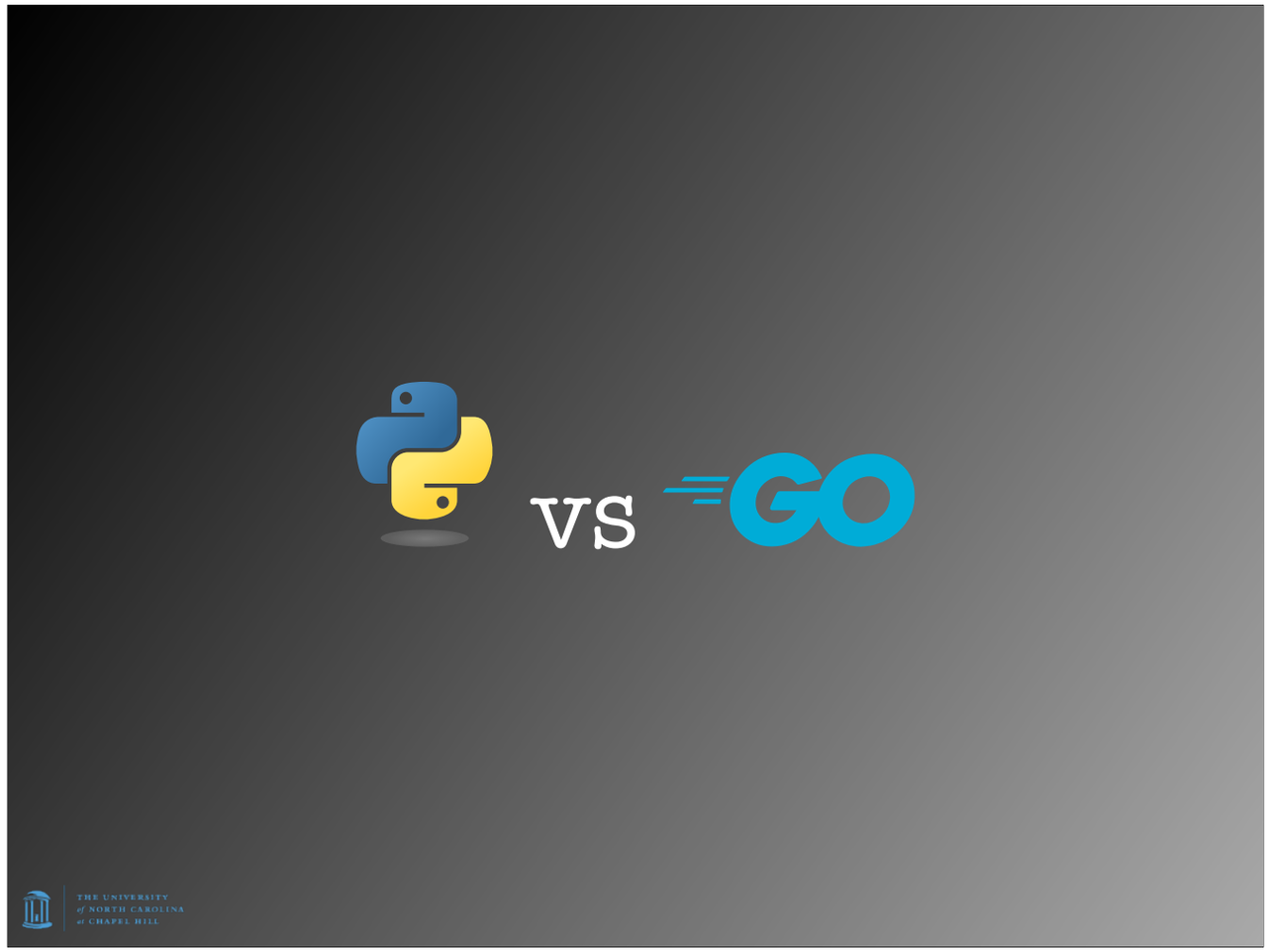
Install in your own user account;e.g.,

```
$ cd ~
$ mkdir install
$ cd install
$ wget https://go.dev/dl/go1.23.3.linux-
amd64.tar.gz
$ cd ..
$ tar zxvf install/go1.23.4.linux-
amd64.tar.gz
```

This installs in ~/go/. Note this may require modifying your GOPATH; e.g., adding this to your shell profile:

```
export GOPATH=$HOME/go.my
```


Another option that I use is to simply download and decompress the relevant .tar.gz file right in my Linux user account on those systems where I do not have root access. The reality is you can apply this technique to ANY of the OSes including Windows. You simply need to be sure you modify your PATH variable and a few other environment variables so that Go knows where to look. Then it “just works.”



That covers getting you setup.

Now it’s time to get into the reason you’re really here.

Comments

Python	Go

`# Single line comment`
 	
`""" Multiline strings can`
`be written using three`
`"s, and are often used`
`as documentation.`
`"""`
	`// Single line comment`
	
	`/* Multi-`
	`line comment */`
	
	`/* A build tag is a line`
	`comment starting with`
	`“//go:build”`
	`and can be executed by`
	`go build -tags="foo bar"`
	`command. Build tags are`
	`placed before package`
	`clause near top of file`
	`followed by blank line`
	`or other comments. */`
	`//go:build prod || dev`

Comments in Go are similar to C. Single line comments use double slashes, while multi-line comments use /* and */. The most unique thing in Go are build tags, which are a line comment starting with “//go:build” followed by a boolean logic of tags. These work in conjunction with the “go build -tags=“…”” command to handle conditional compilation, such as when you have, for example, a free app, a pro app, and an enterprise app where varying features are included in the final binary.

Primitives and Operators

Python	Go
`1`	`1`
`-2`	`-2`
`1.2`	`1.2`
`(1+2)-3*4/5`	`(1+2)-3*4/5`
`int(6)`	`int(6)`
`True`	`true`
`False`	`false`
`not True`	`!true`
`1 == 1`	`1 == 1`
`2 != 1`	`2 != 1`
`2 > 1`	`2 > 1`
`1 < 2 and 2 <= 3`	`1 < 2 && 2 <= 3`
`1 < 2 or 2 <= 3`	`1 < 2 || 2 <= 3`
`"Frank"`	`"Frank"`

Primitives and operators in Go are similar to most programming languages. Numbers—integers and floats—along with math operators like +, -, *, /, etc. Boolean values and operators. Even strings are very similar.

The key thing to note is that Go is strongly typed, so once a value is set to be of a certain type, you need to typecast/convert variables/values to match in order to work on them together (e.g., if you want to add an integer and a float in Go, you need to convert both to integers or both to floats, then add).

Variables (declared)

Python	Go

`# Declare and assignment`
`name = "Frank"`
`day = 12`
	`// Declare and assignment`
	`var name string = "Frank"`
	`var day int = 12`

THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

Variable naming in Go is similar to most languages. That said, there are some interesting differences. Here we see an example of both defining a variable, setting its type, and assigning it a value.

Also note that I had to use print statements in Go to use the variables assigned, as the Go compiler will complain if you do things like define a variable and then never use it, or include a package but never use anything from it, etc.

Variables (inferred)

Python	Go

`// Same as before`
`name = "Frank"`
`day = 12`
 	
`day = "Fred"`
	`// Inference`
	`name := "Frank"`
	`day := 12`
	
	`day = "Fred"` ← ✗

Here we see an example of using ":="—known as the "short variable declaration operator"—which allows you to both define and assign a variable a value in one step, where you let the Go compiler "infer" the type of the variable based on the value being assigned. Here "name" is inferred to be a string variable while "day" is inferred to be an integer.

UNLIKE Python, where you can easily set the value of a variable one moment as an integer and then later as a string, the Go compiler does not allow this. Go will complain that you are trying to set an integer variable as a string and simply will not compile. This can prevent a multitude of issues.

In fact, if you use the various extensions available in such IDEs as Microsoft's Visual Studio Code, all of this will be pointed out to you right in the editor, preventing you from making many common mistakes even before trying to compile.

For those who use things like the Flake8 extension for Python, this should be familiar. The difference is that while Flake8 can help point this out while coding, if you are not in the editor and you simply run the Python code, the Python interpreter will gladly do so. The Go compiler will never let you compile such code.

Packages

Python	Go

`import os`
 	
`import os, math`
 	
`from math import exp`
	`import "os"`
	
	`import (`
	`  "os"`
	`  "math"`
	`  "github.com/google/uuid"`
	`)`

When the time comes to import packages, again things are similar though not the same. Where Python lets you import multiple modules on the same line separating each with a comma, in Go you enclose them in parentheses and separate them with whitespace/newlines. While Python lets you import a single function from a module, Go does not. This is not important, as when you compile your Go code, only the relevant bits from each module are compiled into the final executable.

Finally, where Python relies on modules installed using something like ‘pip’, in the Go world there is no centralized package authority like PyPi. Instead, anyone can host a Go package wherever they like. To access it, you simply reference the URL to reach the source code as shown here. This can be both good (no single “supply chain attack” can take out the language’s module repository) and bad (there is no central location to scan for viruses/etc.).

Functions

Python

```python
#!/usr/local/bin/python3


def fn(first):
    full = first + " Seesink"
    return full, 12


def main():
    fullname, day = fn("Frank")

    print("Hello", fullname)
    print("It is Dec.", day)


if __name__ == "__main__":
    main()
```

Go

```go
package main

import "fmt"

func fn(first string) (full string, age int) {
    full = first + " Seesink"
    return full, 12
}

func main() {
    fullname, day := fn("Frank")

    fmt.Println("Hello", fullname)
    fmt.Println("It is Dec.", day)
}
```

Conditionals (if)

Python	Go

`age = 15`
 	
`if age > 21:`
`    print("You can drink")`
`elif age > 12:`
`    print("You can watch TV")`
`else:`
`    print("Go to bed")`
	`age := 15`
	
	`if age > 21 {`
	`    fmt.Println("You can drink")`
	`} else if age > 12 {`
	`    fmt.Println("You can watch TV")`
	`} else {`
	`    fmt.Println("Go to bed")`
	`}`

Conditionals (switch)

Python

```python
#!/usr/local/bin/python3


def main():
    x = 42
    if x == 0:
        pass
    elif x == 1 or x == 2:
        print("So low.")
    elif x == 42:
        print("The meaning of life.")
    elif x == 44:
        pass
    else:
        # Default case
        pass

if __name__ == "__main__":
    main()
```

Go

```go
package main

import "fmt"

func main() {
    x := 42
    switch x {
    case 0:
    case 1, 2:
        // Can have multiple matches on one case
        fmt.Println("So low.")
    case 42:
        fmt.Println("The meaning of life.")
        // Cases don't "fall through".
        // There is a `fallthrough` keyword,
        // however. See:
        //  https://go.dev/wiki/Switch#fall-through
    case 44:
        // Unreached.
    default:
        // Default case is optional.
    }
}
```

Error Handling

Python	Go

`#!/usr/local/bin/python3`
 	
`def countdown(num):`
`  if num < 0:`
`  return 0, "Countdown < 0."`
`  print("Counting down from", num)`
`  return num - 1, None`
 	
`def main():`
`  num, err = countdown(5)`
`  if err:`
`  print(err)`
`  print(num, "good.")`
 	
`  num, err = countdown(-1)`
`  if err:`
`  print(err)`
 	
`if __name__ == "__main__":`
`  main()`
	`package main`
	
	`import (`
	`  "errors"`
	`  "fmt"`
	`)`
	
	`func countdown(num int) (int, error) {`
	`  if num < 0 {`
	`  return 0, errors.New("Countdown < 0.")`
	`  }`
	`  fmt.Println("Counting down from", num)`
	`  return num - 1, nil`
	`}`
	
	`func main() {`
	`  num, err := countdown(5)`
	`  if err != nil {`
	`  fmt.Println(err)`
	`  }`
	`  fmt.Println(num, "good.")`
	
	`  num, err = countdown(-1)`
	`  if err != nil {`
	`  fmt.Println(err)`
	`  }`
	`}`

Since we discussed functions and conditionals, let's quickly comment about error handling in Go. In Python, exception handling is typically done using try/except blocks, the idea being that if the Python interpreter hits on something it can't handle, it will dump a stack trace.

In Go, the expectation is that you handle errors at each point in the program where they may occur. Typically you will see code similar to the following, where a function call is made, and the return values include whatever the function's purpose is, along with an extra value for any errors returned. So if the function encounters an error, instead of blowing up the program, it passes back the error to the calling function. That function, in turn, is expected to either handle the error or yet again pass it back to its calling function.

Of course, "the buck stops here" at the main function. So either you handle the error, or your program goes BOOM!

The underscore (“_”)

Python	Go

`#!/usr/local/bin/python3`
 	
`def main():`
`    mydict = {`
`        "first": "Frank",`
`        "last":  "Seesink",`
`        "year":  “2024"`
`    }`
`    for key, val in mydict.items():`
`        print(val)`
 	
`if __name__ == "__main__":`
`    main()`
	`package main`
	
	`import "fmt"`
	
	`func main() {`
	`    mydict := map[string]string{`
	`        "first": "Frank",`
	`        "last":  "Seesink",`
	`        "year":  "2024",`
	`    }`
	`    for _, val := range mydict {`
	`        fmt.Println(val)`
	`    }`
	`}`

A quick explanation of the use of the underscore (“_”) in Go.

Sometimes you don’t need a value that is returned by a function. For example, maybe you don’t care if an error occurred and you wish to ignore it altogether. For such cases you have the underscore. It is a placeholder that says “Yeah, we know something goes here, but we don’t care.”

Now here I quickly show you both the map type—which is the Go equivalent of a dictionary in Python—and the use of the underscore. And much as in Python if you use the .items() function which returns both the key and the value of each item, in Go the ‘range’ keyword does the same for a map. But maybe we only care about the value. While in Python you COULD simply use the .values() function to ONLY return values from a dictionary (just as you could use the .keys() function to only return dictionary keys), typically in Go you would do something like this, where you simply use the “_” in the place where the key is returned.

Loops (for)

Python	Go

`#!/usr/local/bin/python3`
 	
`def main():`
`    for i in ["dog","cat"]:`
`        print(i)`
 	
`    for i in range(5):`
`        print(i)`
 	
`    x = 0`
`    while x < 5:`
`        print(x)`
`        x += 1`
 	
`if __name__ == "__main__":`
`    main()`
	`package main`
	
	`import "fmt"`
	
	`func main() {`
	`    for _, val := range []string{"dog","cat"} {`
	`        fmt.Println(val)`
	`    }`
	
	`    for i := 0; i < 5; i++ {`
	`        fmt.Println(i)`
	`    }`
	
	`    x := 0`
	`    for x < 5 {`
	`        fmt.Println(x)`
	`        x++`
	`    }`
	`}`

While the “for” loop in Python is more like an iterator method, in Go it is more traditional in nature. Here you can see examples of how Python would iterate over a list of strings or a range of numbers, vs. how Go would do the same thing.

Go’s “for” loops are more like what is seen in C-like languages, where you have an init statement that is executed before entering the loop, a boolean condition expression that is evaluated before each iteration (and when false causes the loop to end), and a post statement executed after each iteration, all separated by semicolons (“;”).

Also note that Go has no “while” keyword. A “while” loop is really just a “for” loop that does not have an init or post statement. It solely has a boolean condition expression that determines when the loop ends.

Go routines

Python	Go

`import concurrent.futures`
 	
`…`
`# Use ThreadPoolExecutor for concurrent execution`
`with concurrent.futures.ThreadPoolExecutor() as executor:`
`    # Map sites to the ping_site function`
`    results = executor.map(pingSite, sites)`
 	
`# -- OR --`
 	
`import asyncio`
 	
`async def pingSite():`
`    print("pingSite started")`
`    await asyncio.sleep(5)`
`    print("pingSite done")`
 	
`async def main():`
`    # fire and forget pingSite()`
`    asyncio.ensure_future(pingSite())`
`    print('Do some actions 1')`
`    await asyncio.sleep(5)`
`    print('Do some actions 2')`
 	
`loop = asyncio.get_event_loop()`
`loop.run_until_complete(main())`
	`go pingSite(site)`

THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

Here comes the real power of Go, especially when writing programs which have to deal with many things (such as network devices) at a time.

Python was developed in a time of single-core CPUs. Famously Python has the GIL (Global Interpreter Lock), a feature that Guido van Rossum put into Python to simplify execution. The challenge is that the GIL also heavily restricts Python's ability to leverage modern, multi-core CPUs, as it only lets one thing run at a time in essence. In order to do better, you either have to leverage something like concurrent.futures (which takes its name from its Java-based counterpart)—a module that lets you run either a pool of threads or a pool of processes—or something like asyncio, which provides cooperation multitasking a la OSes like Windows 3.1/95 in days of yore.

Go was built in the time of multi-core CPUs. So foundational to how it works, Go has built-in support for what are called "Go routines", or more generically, "green threads." These are super lightweight processes that the Go runtime handles for you, letting you focus on your coding.

The keyword you need to remember is simply "go." Add this in front of any function call, and now that function is a Go routine. These Go routines will run either until they finish executing, or when the main thread of the program exits, whichever comes first. This is KEY to understand. If you write a Go routine to do something, and it is in the middle of executing when the main thread reaches its end, that Go routine dies. Which brings us to…

WaitGroups & Channels

Python

```python
#!/usr/local/bin/python3

import concurrent.futures
import subprocess

def pingSite(site):
    try: # Use '-n' below instead of '-c' on Windows
        result = subprocess.run(
            ["ping", "-c", "1", site],
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True
        )
        if result.returncode == 0:
            return f"{site} is reachable"
        else:
            return f"{site} is not reachable"
    except Exception as e:
        return f"Error pinging {site}: {e}"

def main():
    # List of sites to ping
    sites = ["google.com", "github.com", "nonexistent.website"]

    # Use ThreadPoolExecutor for concurrent execution
    with concurrent.futures.ThreadPoolExecutor() as executor:
        # Map sites to the ping_site function
        results = executor.map(pingSite, sites)

    # Print the results
    for site, result in zip(sites, results):
        print(result)

if __name__ == "__main__":
    main()
```

Go

```go
package main
import (
    "fmt"
    "os/exec"
    "sync"
)
var wg sync.WaitGroup // To synchronize goroutines

// List of sites to ping
var sites = []string{"google.com", "github.com",
"nonexistent.website"}

// Buffered channel to store results
var results = make(chan string, len(sites))

func pingSite(site string) {
    defer wg.Done()
    cmd := exec.Command("ping", "-c", "1", site) //'-n' on Windows
    if err := cmd.Run(); err != nil {
        results <- fmt.Sprintf("%s is not reachable", site)
        return
    }
    results <- fmt.Sprintf("%s is reachable", site)
    return
}

func main() {
    for _, site := range sites {
        wg.Add(1)
        go pingSite(site)
    }

    wg.Wait() // Wait for all goroutines to finish
    close(results)

    // Print the results
    for result := range results {
        fmt.Println(result)
    }
}
```

Only prints results once all threads finish

Language Similarities

Python	Go

`import os`
 	
`def itsvalid():`
`  print("Valid day of the month")`
`  cwd = os.getcwd()`
`  print(cwd)`
 	
`def main():`
`  # Variable assignment`
`  name = "Frank"`
`  day = 12`
 	
`  if day >= 1 and day < 31:`
`  itsvalid()`
 	
`if __name__ == "__main__":`
`  main()`
	`package main`
	
	`import (`
	`  "fmt"`
	`  "os"`
	`)`
	
	`func itsvalid() {`
	`  fmt.Println("Valid day of the month")`
	`  cwd, _ := os.Getwd()`
	`  fmt.Println(cwd)`
	`}`
	
	`func main() {`
	`  // Variable assignment`
	`  name := "Frank"`
	`  day := 12`
	
	`  if day >= 1 && day < 31 {`
	`  itsvalid()`
	`  }`
	
	`  fmt.Println(name)`
	`}`

Here is a quick comparison showing how Python and Go look side-by-side. This example has a main function where variables are defined and used, along with another function that is called. There is a conditional with some boolean operators, along with a comment.

As you can see, while the syntax differs, there is more that they have in common than they do in difference. But let's delve into the specifics here.

Workflow

go mod init

```
$ go mod init github.com/fseesink/GoTest
go: creating new go.mod: module github.com/
fseesink/GoTest
go: to add module requirements and sums:
        go mod tidy
```


When starting a new project, you typically perform a “go mod init” to create a go.mod file that contains information like where your Go module/app’s code can be found, along with what version of the Go compiler was used.

go mod init

```
module github.com/fseesink/GoTest

go 1.23.4
```


Here's a simple example of the output of a new go.mod file. Over time, as you add in other packages/modules, this file will contain this information including version numbers used.

go mod tidy

As you work on your project, you may need to perform a “go mod tidy” command to tidy up the go.mod file so that it is up-to-date. This will trigger Go going through your code making sure all the dependencies are accounted for, and downloading any and all packages that you don’t have yet or that are out-of-date.

This is a bit like using “pip3 freeze > requirements.txt” combined with “pip3 install -r requirements.txt”.

Workflow

Python	Go

`#!/usr/local/bin/python3`
 	
`print("Hello world")`
	`package main`
	
	`import "fmt"`
	
	`func main() {`
	`  fmt.Println("Hello world")`
	`}`

`$ python3 helloworld.py`
 	
or if permissions set, simply
 	
`$ helloworld.py`
	`$ go run helloworld.go`
	or
	`$ go run .`
	to run interactively.
	
	Compile and run executable with
	`$ go build .`
	`$ helloworld`

Ok let's discuss workflow.
When you write code in Python, you typically write code in an editor, save the file, then execute the file from a terminal session.
When you write code in Go, you typically do the same, only you have to compile your code first before running it. Go makes this easy in fact by offering the "go run" command, which performs both duties in one shot. This makes your workflow very similar to Python.

Workflow Performance

Python

```
#!/usr/local/bin/python3

print("Hello world")
```

```
$ time python3 helloworld.py
Hello world
python3 helloworld.py  0.02s
user 0.02s system 36% cpu
0.111 total
```

Time to run executable binary

Go

```
package main

import "fmt"

func main() {
    fmt.Println("Hello world")
}
```

Time to compile AND run the program (when developing)

```
$ time go run helloworld.go
Hello world
go run helloworld.go  0.14s
user 0.29s system 49% cpu
0.860 total
$ go build helloworld.go
$ time ./helloworld
Hello world
./helloworld  0.00s user 0.00s
system 2% cpu 0.135 total
```


Now let's talk performance. Admittedly this is too simplistic an example. But using this example, you can see that executing the Python "Hello World" program takes .02s. The Go "Hello World" program, when you use "go run", takes .14s. However, once you are done developing, you simply compile your Go program one time. After this, you just run the binary. And as you can see here, the binary executes so quickly that it is listed as .00s. So you get nearly the performance of Python while developing, and much better performance once you truly compile to a binary.

Also note the CPU usage in each case. The Python script consumed 36% CPU, while the Go binary only required 2%. This is significant.

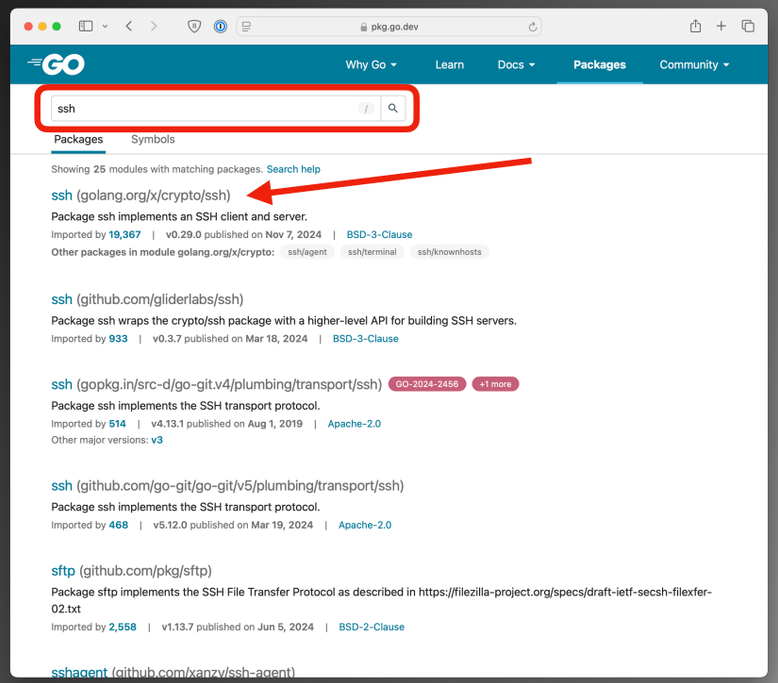
Go Cross-Compilation

- Go creates binary executables specific to an OS/architecture (e.g., x64 Windows, ARM64 Linux)
- Go can cross-compile to ANY supported OS/architecture combination FROM any supported OS/architecture. Simply set GOOS and GOARCH environment variables.

```
$ GOOS=linux GOARCH=arm64 go build .
```

Finding Go packages is as simple as visiting this site and typing in what you are looking for.

Finding GO Packages

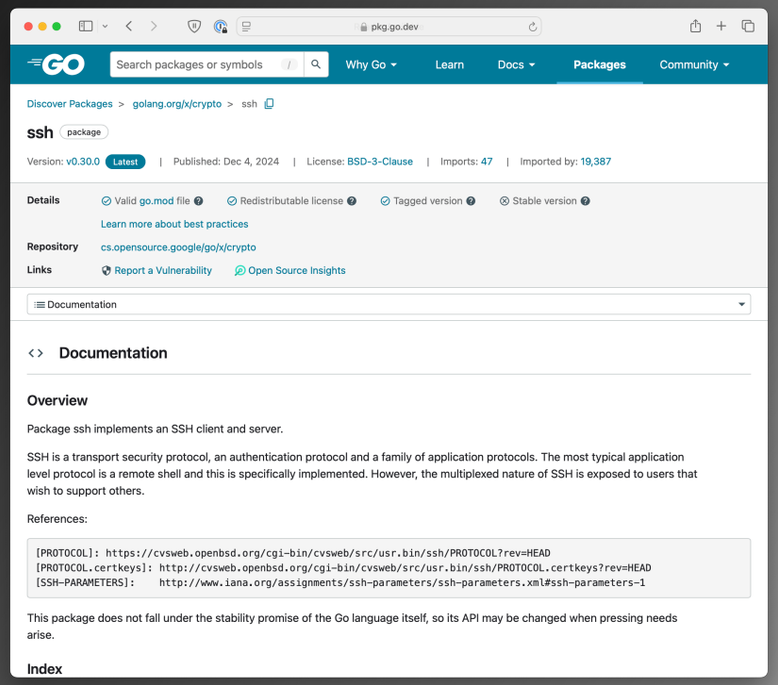


THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

https://pkg.go.dev/

For example, Go has SSH support built-in, though note that it's location (golang.org/x/crypto/ssh) tells you that it is under the Go project but outside the main Go tree. That means that they are developed under looser compatibility requirements than the Go core. But still, there is no need for an external library such as netmiko/etc. to perform SSH functions.

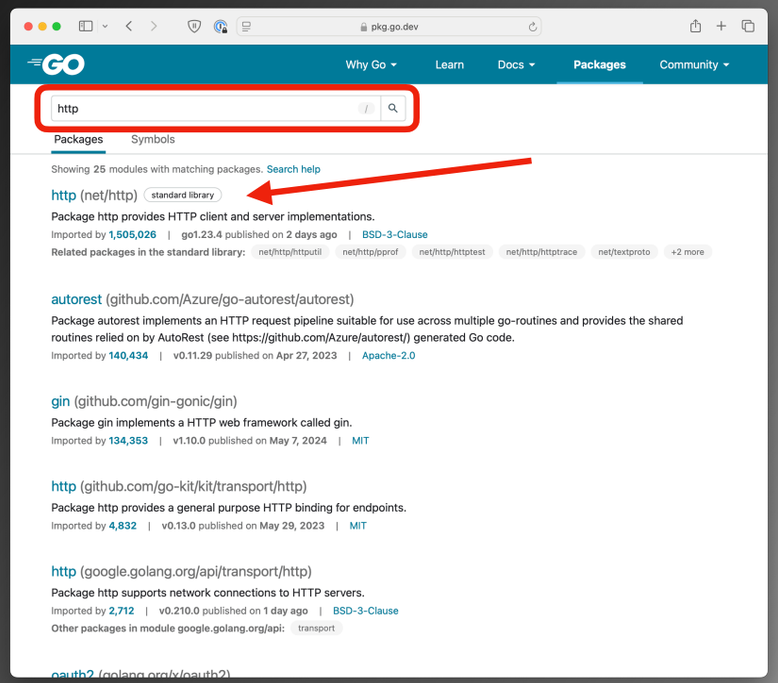
Finding GO Packages



THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

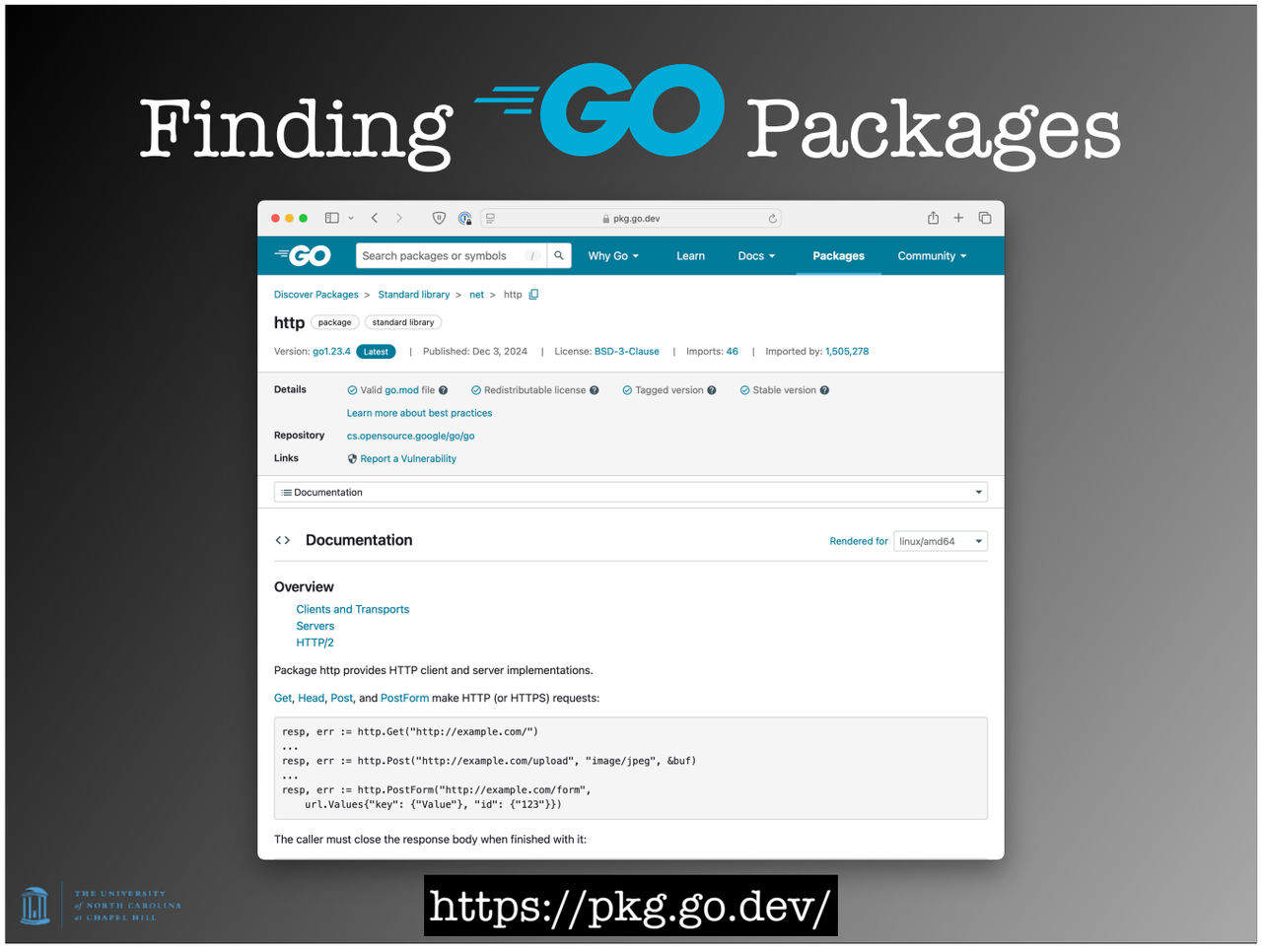
https://pkg.go.dev/

Finding GO Packages

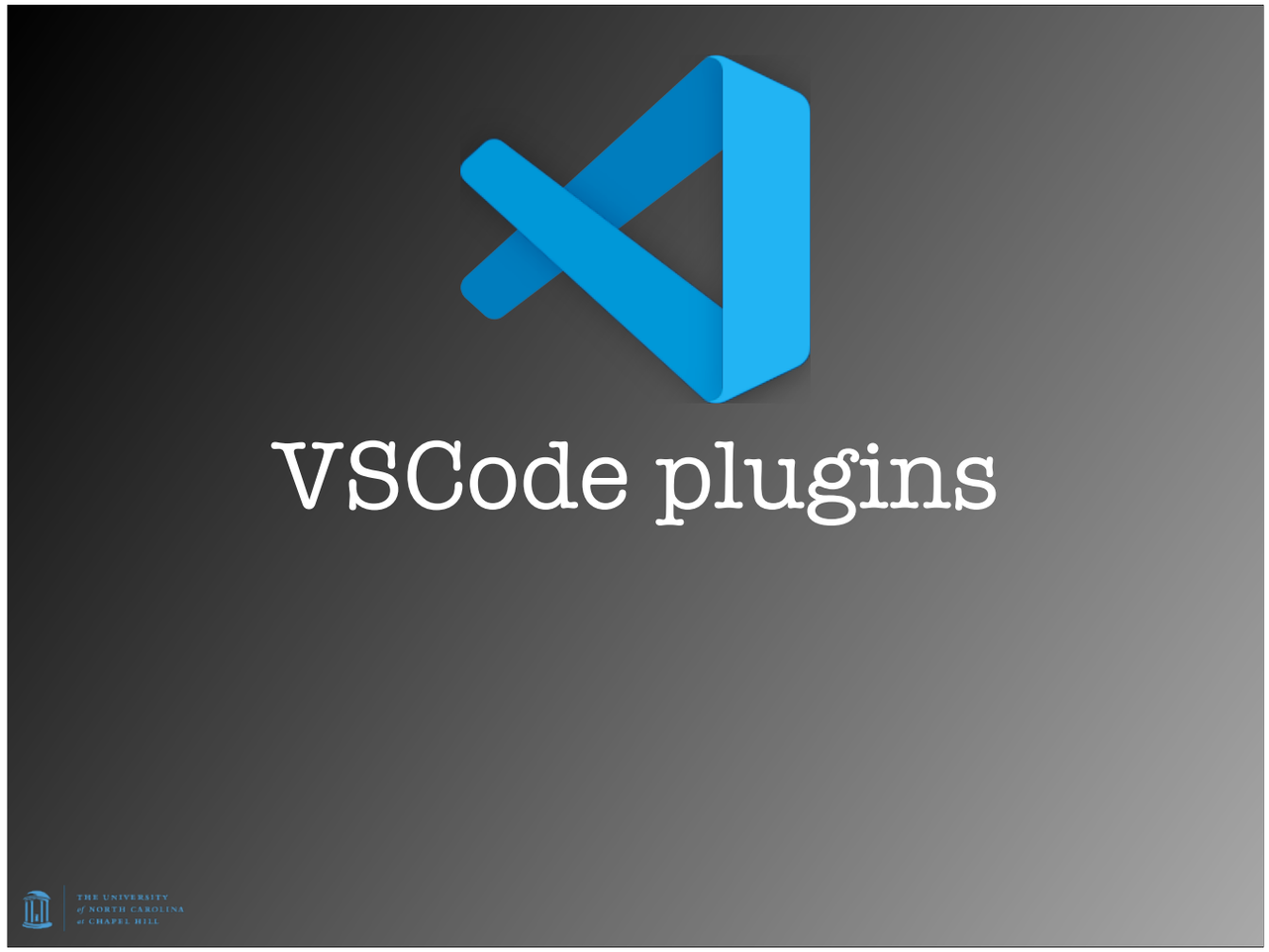


https://pkg.go.dev/

Another package you might want to leverage is the http package.



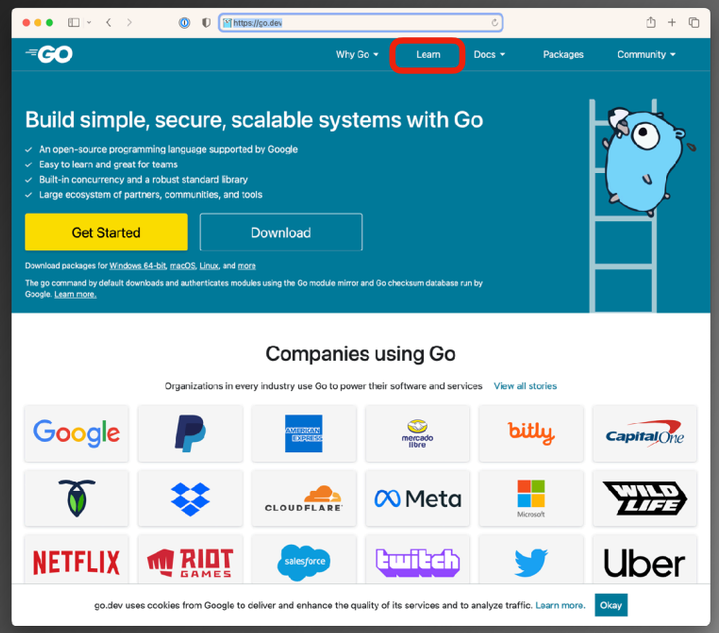
Unlike SSH, however, the HTTP package IS part of the Go core. So Go has full HTTP support built-in, meaning no need for something like the “requests” library in Python.



When developing code, if you use any kind of Integrated Development Environment (IDE) such as Visual Studio Code (VSCode), I strongly encourage you to look for extensions that support Go. I use VSCode, and the Golang extensions are absolutely fantastic. With them installed, many things which would require compiling to trigger a warning/error are shown right in the code editor. And on each save, the extension runs “go fmt” in the background, guaranteeing that your code will always be formatted according to the Go standard.

To Learn More...

Go (Golang)



https://go.dev/

On the main page of Go.dev, click on the “Learn” link at the top to access educational resources for learning Go.

Go (Golang)

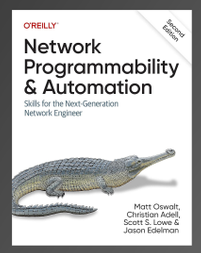
- **Learning Go**
  https://www.linkedin.com/learning/learning-go
- **Go for Python Developers**
  https://www.linkedin.com/learning/go-for-python-developers
- https://**learnxinyminutes.com**/docs/go/

Here are just a few examples of online courses you could take to learn more about Go.

That last site is very handy for quickly refreshing yourself on a language.

Books

https://www.amazon.com/Network-Automation-operations-applications-programming/dp/1800560923



https://www.amazon.com/Network-Programmability-Automation-Next-Generation-Engineer/dp/1098110838

There are also several books out there on just Go or on using Go in network automation specifically. Here are two that I have which I can highly recommend.

Once you get past writing CLI tools, if you wish to write a GUI application, know that in Go you have packages like the fyne.io library. If you are familiar with things like GTK, Qt, Tcl/Tk, wxWidgets, or Tcl/Tk, Fyne provides you with similar features while staying in Go. That is, when you're done, once again you have a single binary executable for a given OS/architecture that provides a GUI application.

If you like both

https://www.amazon.com/Go-Gopher-Tee-Developer-Umbrella/dp/B0DJZK5DXT

If like me you enjoy both Python and Go, there’s even a shirt out there with both now!

Thank You

https://frank.seesink.com/presentations/Internet2TechEx-Fall2024/

Frank Seesink
frank@seesink.com
frank@unc.edu

THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL