
Go for Python Programmers Documentation

Release 0.1a

Jason McVetta

October 16, 2014

1	About This Book	3
1.1	Copying	3
1.2	Resources	3
1.3	Classroom Instruction	3
1.4	Related Courses	3
1.5	Author	3
1.6	Payment	4
2	Introduction	5
2.1	What is Go?	5
2.2	Searching	6
2.3	Who’s Using Go?	6
2.4	Background Assumptions	6
2.5	Requirements	7
2.6	Further Reading	7
3	Installation & Setup	9
3.1	Installing Go	9
3.2	Environment Variables	10
3.3	IntelliJ IDE	10
4	Hello, World!	13
4.1	Curly Braces	14
4.2	Package	14
4.3	Import	14
4.4	Functions	14
4.5	Return	14
4.6	Variables	15
4.7	main()	16
4.8	Compiling	16
5	Documentation	19
5.1	Self-Documenting Code	19
5.2	godoc tool	21
5.3	GoDoc.org	21
6	The Go Toolchain	25
6.1	go build	25

7	Project Layout	27
7.1	Hello World	27
7.2	Dependencies	28

This book is intended to provide a solid introduction to the [Go](#) language for experienced Python programmers.

Contents

About This Book

1.1 Copying

All source code used in this book is Free Software, released under the [terms of the GPL v3](#).

The remainder of the book is released under a [Creative Commons Attribution-ShareAlike 3.0 Unported License](#).

Sing along with [Richard Stallman](#), founder of the Free Software movement!

1.2 Resources

The [Sphinx source code](#) for this book is available on on Github.

The latest version of this book is published at [Read the Docs](#). It is also available in [PDF](#) and [ePub](#) formats.

1.3 Classroom Instruction

Classroom instruction in Go, Python, SQL, REST API development, cloud deployment, and continuous integration is available from the author in Silicon Valley, or on demand anywhere in the world, for groups of five to twelve students. [Contact the author](#) for more information.

1.4 Related Courses

Todo

Add related courses.

1.5 Author

[Jason McVetta](#) is an independent consultant, teacher, and Free Software activist based in beautiful-but-too-cold San Francisco. He studied government and philosophy in college, but was unwilling to sell his soul to the Demopublicat-ican Party, so he went to work in software. After 12 years as a professional Pythonist he met his new love, Go. He is able to fly without mechanical assistance.

1.6 Payment

This is a Free book, but you are more than welcome to pay me for it! You are under no obligation, legal or moral, to do so. But I can always use a couple bucks for coffee. =)

Citations

Introduction

Go is like C and Python had a kid, who inherited Python's good looks and pleasant demeanor, along with C's confidence and athletic ability.

2.1 What is Go?

Go is an open source, compiled, garbage-collected, concurrent system programming language. It was first designed and developed at Google Inc. beginning in September 2007 by Robert Griesemer, Rob Pike, and Ken Thompson.¹

2.1.1 Minimalist

The creators of Go advocate a minimalist approach to language design. This minimal elegance is often contrasted with the “kitchen sink” approach of C++.

2.1.2 Opinionated

Like Python, in Go there is often “one right way to do things”, although that phrase is not so commonly used as in the Pythonist community. The compiler is renowned for its strictness. Things that would at most be warnings in other languages - e.g. unused imports, unused variables - are hard compiler errors in Go. At first that may seem like a draconian buzzkill - but soon you will begin to appreciate how clean and cruft-free it keeps your code.

2.1.3 Fast

No question about it - Go is blazing fast. Despite a still young compiler with minimal speed optimizations, Go regularly scores at or near the top of inter-language benchmark comparisons. That's because Go is statically typed, compiled - and although garbage-collected, allows the programmer some degree of control over memory usage.

2.1.4 Batteries Included

To a large degree Go follows Python's “batteries included” philosophy. Although there are not yet as many 3rd party libraries as there are for Python, nevertheless Go's standard library provides a solid foundation for many modern programming tasks. Built-in support for serving HTTP, de/serializing JSON, template rendering, and strong cryptography make Go ideal for modern web services development.

¹ [https://en.wikipedia.org/wiki/Go_\(programming_language\)](https://en.wikipedia.org/wiki/Go_(programming_language))

2.1.5 Productive

Go is frequently - and favorably - compared with Python and Ruby for programmer productivity. It's clean elegant syntax, unobtrusive static typing, optional duck typing, batteries-included standard library, lightning-fast compilation, and support for a variety of modern continuous integration tools make Go a productivity champion.

2.2 Searching

Since “go” is such a common English word, it is completely useless as a search term when googling. Instead search for “golang”. It is standard practice in the community that all Go-related blog posts, Github repos, tweets, job postings, etc be tagged with “golang”.

2.3 Who's Using Go?

Google Home of the Go authors and core team, significant parts of the Google infrastructure are thought to be written in Go. The company however does not publicly disclose which Google services are written in which languages. Youtube is known to be powered by [Vitess](#), an open source, massively scalable database multiplexing layer written in Go.

Heroku Maintainers of the [pq](#) driver for PostgreSQL, significant parts of Heroku's infrastructure are said to be written in Go. One component that has been open sourced is [Doozer](#), a consistent distributed data store.

Canonical The company behind Ubuntu Linux recently [rewrote](#) its [Juju](#) devops management application from Python to Go, with pleasing results.

Soundcloud Internet music thought leaders SoundCloud use a bespoke build and deployment system called [Bazooka](#), designed to be a platform for managing the deployment of internal services. It is promised that Bazooka will be open sourced soon.

Bitly Significant parts of [Bitly](#) are written in Go, including [nsq](#), a realtime distributed message processing system.

Cloudflare Large volumes of internet traffic are served by Cloudflare's [Railgun](#) web optimization software, designed to speed up the delivery of content that cannot be cached.

Iron.io Cloud infrastructure company and core Go community supporters [Iron.io](#) replaced parts of their Ruby code-base with Go, allowing them to handle a greater load with 2 workers than with 30 Ruby workers.

TinkerCAD Cloud-based CAD application [TinkerCAD](#) is written in Go.

Drone.io Continuous integration service [Drone.io](#) is written in Go.

2.4 Background Assumptions

- Familiarity with Python or a similar dynamic, interpreted language.
- Comfortable working on the UNIX command line.
- Basic understanding of internet protocols such as HTTP(S).

2.5 Requirements

Command line examples were written on an [Ubuntu](#) 12.10 desktop environment. Everything we do in this book can also be done on other flavors of Linux, on Mac OSX - probably even on Windows.

In addition to Go language basics, this book also covers [cloud](#) deployment and [continuous integration](#) tools & techniques popular in the Go community. You will need free accounts on the following services:

- [Github](#)
- [Drone](#)
- [Coveralls](#)
- [Heroku](#)

2.6 Further Reading

- [Effective Go](#) by the creators of the language. Perhaps the single most valuable documentation resource for Go programmers.
- [An Introduction to Programming in Go](#) by Caleb Doxsey
- [Go for Pythonists](#) by Aditya Mukurjee

Citations

Installation & Setup

3.1 Installing Go

The first thing we need is a working Go installation on our system. The following instructions were derived from the official [Getting Started](#) documentation.

3.1.1 Binary Packages

Binary distributions of Go are available for the FreeBSD, Linux, Mac OS X, NetBSD, and Windows operating systems from the [official downloads page](#).

3.1.2 Building from Source

Many developers prefer to install Go directly from the source code. Building is easy and quick - the Go compiler and tool chain can be compiled in a matter of minutes on even a modest workstation or laptop.

Install Build Tools

The Go tool chain is written in C. To build it, you need a C compiler installed. You will also need the Mercurial DVCS tool to download the source. On Ubuntu run the following command to install the necessary packages:

```
$ sudo apt-get install gcc libc6-dev mercurial
```

See [Installing Go from source](#) and [Install C tools](#) for build instructions on other platforms.

Fetch the repository

Go will install to a directory named `go`. Change to the directory that will be its parent and make sure the `go` directory does not exist. Then check out the repository:

```
$ hg clone -u release https://code.google.com/p/go
```

Build Go

To build the Go distribution, run

```
$ cd go/src
$ ./all.bash
```

If all goes well, it will finish by printing output like:

```
ALL TESTS PASSED
```

```
---
```

```
Installed Go for linux/amd64 in /home/you/go.
Installed commands in /home/you/go/bin.
*** You need to add /home/you/go/bin to your $PATH. ***
```

where the details on the last few lines reflect the operating system, architecture, and root directory used during the install.

3.2 Environment Variables

These instructions assume Go is installed at `~/go`. If you installed elsewhere, change the commands to reflect your actual installation path.

The Go tools look for several environment variables. Typically these are set in your shell profile. If you are using Bash, add the following lines to your `~/ .bashrc`:

```
export GOBIN=~/go/bin
export GOARCH=amd64
export GOOS=linux
export GOROOT=~/go
```

3.2.1 PATH

You probably also want to add the path to your `go/bin` directory to your `PATH`:

```
export PATH=$PATH:/home/you/go/bin
```

Reload your `~/ .bashrc` file to active the new environment variables. Once the `go` binary is on your `PATH`, you can confirm your install is working correctly by running the `go version` command.

```
$ go version
go version go1.1.1 linux/amd64
```

3.2.2 GOPATH

The `GOPATH` environment variable is used to specify directories outside of `$GOROOT` that contain Go source code. Typically `GOPATH` is set to the root of a workspace.

See *Project Layout* for more detail.

3.3 IntelliJ IDE

Go bindings are available for many popular programming editors and IDEs. As Go is still a relatively young language, none of these bindings are as full-featured as those for a mature language like Java. However a good [integrated development environment](#), even if not perfect, is still a valuable tool.

Currently (July 2013) the IDE with the best Go support is [IntelliJ IDEA](#).

3.3.1 Download

Download the Free community edition of IntelliJ IDEA appropriate for your platform from the official [download page](#).

Untar the downloaded tarball, descend into the resulting folder, and start IntelliJ by running:

```
$ ./bin/idea.sh
```

3.3.2 Install Go Plugin

Open the Settings window (File -> Settings) and go to Plugins under IDE Settings. Click on Browse repositories, and type “golang” into the search form. There should be only one result, “golang.org support”. Double-click on it to install, and close the settings. You will be prompted to restart IntelliJ.

If you like vi-keys, you may also wish to install the `IdeaVim` plugin.

Hello, World!

Let's start out with a Hello World example. Our program will greet the world with a hello and the current time.

In Python it looks something like this:

```
1  #!/usr/bin/env python
2
3  import time
4
5
6  def greeting() :
7      '''Returns a pleasant, semi-useful greeting.'''
8      return "Hello world, the time is: " + time.ctime()
9
10
11 def main() :
12     print greeting()
13
14
15 if __name__ == '__main__':
16     main()
```

It's not that different in Go:

```
1  // A "Hello World" program that prints a greeting with the current time.
2  package main
3
4  import (
5      "fmt"
6      "time"
7  )
8
9  // greeting returns a pleasant, semi-useful greeting.
10 func greeting() string {
11     return "Hello world, the time is: " + time.Now().String()
12 }
13
14 func main() {
15     fmt.Println(greeting())
16 }
```

4.1 Curly Braces

The first thing one may notice is that whereas Python uses whitespace to denote scope, Go uses curly braces. The Go authors did, however, take a lesson from Python's culture of readable code. Although it is certainly possible to write syntactically valid Go code without any indentation at all, nevertheless almost all Go code one sees is formatted consistently and readably. That's because, Go includes a code formatter along with the compiler. The formatter, `gofmt`, is considered canonically correct by the Go community - if you don't like how `gofmt` formats your code, *you are wrong*. It is customary that `gofmt` always be run before checking in code.

```
$ gofmt -w hello.go # -w option updates the file(s) in place
```

4.2 Package

In any `.go` file, the first non-comment line is a package declaration. The package declaration is mandatory - *every* `.go` file must begin with one. Every `.go` file in the same folder must have the same package name.

The package declaration is preceded by a comment, called the *package comment*. Automatic documentation tools like `godoc` extract the package comment as the description of your program.

```
// A "Hello World" program that prints a greeting with the current time.  
package main
```

4.3 Import

Next comes the `import` declaration. Although it is optional, most `.go` files will have one. Each package to be imported is listed on a separate line, inside quotation marks. The packages in our example, `fmt` and `time`, come from the standard library. By convention, 3rd-party packages are named after their repository URL. For example, my Neo4j library hosted on Github would be imported as `"github.com/jmcvetta/neo4j"`.

```
import (  
    "fmt"  
    "time"  
)
```

4.4 Functions

Our program has two functions, `greeting()` and `main()`.

The function `greeting()` takes no arguments, and returns a string. Unlike Java and C, in Go the type declaration *follows* the function name. Like all good function declarations, this one includes a doc comment describing what it does.

```
// greeting returns a pleasant, semi-useful greeting.  
func greeting() string {
```

4.5 Return

Every function that declares a return type, must end with a `return` statement. In this case we add a literal string to the output of `time.Now().String()`

Let's look at the documentation for `time`. We can see that `time.Now()` returns an instance of type `Time`. That instance, in turn, exports a `String()` method that unsurprisingly returns a `string`. Using the addition operator with two strings results in the strings being concatenated. If we had tried to add our greeting string to just `time.Now()` the code would not compile, due to type mismatch.

```
$ godoc time Now String
PACKAGE DOCUMENTATION

package time
    import "time"

[ ... ]

type Time struct {
    // contains filtered or unexported fields
}

A Time represents an instant in time with nanosecond precision.

Programs using times should typically store and pass them as values, not
pointers. That is, time variables and struct fields should be of type
time.Time, not *time.Time. A Time value can be used by multiple
goroutines simultaneously.

Time instants can be compared using the Before, After, and Equal
methods. The Sub method subtracts two instants, producing a Duration.
The Add method adds a Time and a Duration, producing a Time.

The zero value of type Time is January 1, year 1, 00:00:00.000000000
UTC. As this time is unlikely to come up in practice, the IsZero method
gives a simple way of detecting a time that has not been initialized
explicitly.

Each Time has associated with it a Location, consulted when computing
the presentation form of the time, such as in the Format, Hour, and Year
methods. The methods Local, UTC, and In return a Time with a specific
location. Changing the location in this way changes only the
presentation; it does not change the instant in time being denoted and
therefore does not affect the computations described in earlier
paragraphs.

func Now() Time
    Now returns the current local time.

func (t Time) String() string
    String returns the time formatted using the format string

    "2006-01-02 15:04:05.999999999 -0700 MST"

[ ... ]
```

4.6 Variables

Python is dynamically typed. That means there is no difference between variable assignment and declaration. For example, saying `x = 3` is equally valid if `x` was previously undeclared, or if it was already declared as an integer, or

even if it was already declared as e.g. a string.

Go, on the other hand, is statically typed. A variable must be declared as a specific type before a value can be assigned to it. Once declared, a variable may only be assigned values of its declared type. Go also provides an operator, `:=`, that combined declaration and assignment in the same statement.

Let's look at a needlessly complex version of our greeting function to see how variable declaration and assignment work.

```
1 // needlesslyComplexGreeting uses needlessly complex operations to return a
2 // pleasant, semi-useful greeting.
3 func needlesslyComplexGreeting() string {
4     var now string
5     now = time.Now().String()
6     msg := "Hello world, the time is: "
7     msg += now
8     return msg
9 }
```

On line 4 we used the `var` keyword to declare the variable `now` as a string. It is initially set to the `zero value` for string type, which is `""` the empty string. Line 5 assigns the (string) return value of `time.Now().String()` to `now`.

On line 6 we use the `short variable declaration` syntax to declare `msg` as a string, and immediately assign it the value of a literal string. The resulting variable is exactly the same as if we had declared it as `var msg string` on a separate line.

Line 7 appends the value of `now` to the value of `msg`, working in this case exactly like Python's `+=` operator. However unlike Python, *only* strings can be added to other strings, no automatic type coercion is ever attempted.

4.7 main()

Our Hello World program declares its package as `main`, and contains a special function `main()`. That tells Go to compile this code as an executable program, with the entry point at `main()`. The function `main()` has no arguments and no return value. Command line arguments - called "argv" in many languages - are available from the standard library, either raw from `os.Args`, or with higher level abstraction from the package `flag`.

4.8 Compiling

Python is an interpreted language - our program can be run immediately with the `python` command:

```
$ python hello.py
Hello world, the time is: Sat Jul 13 12:49:14 2013
```

Go on the other hand is a compiled language - our source code must first be compiled into an executable binary before we can run it. We will use the `go` command line tool to access the compiler.

4.8.1 go run

As a convenience, the `go` tool provides a `run` command that first compiles the specified file(s), then executes the resulting binary. Given Go's lightning-quick compile times, using `go run` can feel similar to working with an interpreted language like Python or Ruby.

```
$ go run hello.go
Hello world, the time is: 2013-07-13 13:01:23.837155926 -0700 PDT
```

Only programs that declare `package main` – in other words, those which can be compiled into an executable application – can be run with `go run`. Note that all files to be compiled must be specified on the command line, even tho they are all declared as part of the same package.

Running code with `go run` does *not* leave an executable binary file laying around - it is deleted immediately after it is run.

4.8.2 go build

We can use the `go build` command to compile our code into an executable binary. The binary is statically linked, and can be redistributed (on the same platform) just by copying it, with no dependencies. The binary file is created in the same folder that contains the source code.

```
$ ls
hello.go

$ go build -v .
hello

$ ls
hello*  hello.go

$ ./hello
Hello world, the time is: 2013-07-13 13:07:57.150564433 -0700 PDT
```

4.8.3 go install

The `go install` command works like `go build`, except instead of putting the binary file in the source code folder, it puts installs it to `$GOPATH/bin`. If `$GOPATH/bin` is on your `$PATH`, your program will be available as a command after running `go install`.

```
$ ls
hello.go

$ go install -v .
hello

$ ls
hello.go

$ ls $GOPATH/bin
hello*
```

Documentation

Good documentation is an essential part of any Go program. You will frequently need to read and understand the documentation for 3rd Party Open Source libraries. As you become a skilled Gopher you will probably also want to publish your own Open Source packages. A well-documented package attracts more users, and reflects well on its author.

5.1 Self-Documenting Code

Go code is *self-documenting*, meaning the source code explains itself without needing external documentation. If you include doc comments in your code as described below, tools like `godoc` will automatically generate useful documentation for your packages.¹

5.1.1 Package Comments

Every package should have a *package comment*, a block comment preceding the package clause. For multi-file packages, the package comment only needs to be present in one file, and any one will do. The package comment should introduce the package and provide information relevant to the package as a whole. It will appear first on the godoc page and should set up the detailed documentation that follows.

```
/*
Package regexp implements a simple library for regular expressions.
```

The syntax of the regular expressions accepted is:

```
regexp:
    concatenation { '/' concatenation }
concatenation:
    { closure }
closure:
    term [ '*' | '+' | '?' ]
term:
    '^'
    '$'
    '.'
    character
    '[' [ '^' ] character-ranges ']'
    '(' regexp ')'
```

¹ The following sections mostly copied from http://golang.org/doc/effective_go.html#commentary

```
*/  
package regexp
```

If the package is simple, the package comment can be brief.

```
// Package path implements utility routines for  
// manipulating slash-separated filename paths.  
package path
```

Comments do not need extra formatting such as banners of stars. The generated output may not even be presented in a fixed-width font, so don't depend on spacing for alignment—godoc, like gofmt, takes care of that. The comments are uninterpreted plain text, so HTML and other annotations such as `_this_` will reproduce verbatim and should not be used. One adjustment godoc does do is to display indented text in a fixed-width font, suitable for program snippets. The package comment for the `fmt` package uses this to good effect.

Depending on the context, godoc might not even reformat comments, so make sure they look good straight up: use correct spelling, punctuation, and sentence structure, fold long lines, and so on.

5.1.2 Doc Comments

Inside a package, any comment immediately preceding a top-level declaration serves as a *doc comment* for that declaration. Every exported (capitalized) name in a program should have a doc comment.

Doc comments work best as complete sentences, which allow a wide variety of automated presentations. The first sentence should be a one-sentence summary that starts with the name being declared.

```
// Compile parses a regular expression and returns, if successful, a Regexp  
// object that can be used to match against text.  
func Compile(str string) (*Regexp, error) {
```

If the name always begins the comment, the output of godoc can usefully be run through `grep`. Imagine you couldn't remember the name "Compile" but were looking for the parsing function for regular expressions, so you ran the command,

```
$ godoc regexp | grep parse
```

If all the doc comments in the package began, "This function...", `grep` wouldn't help you remember the name. But because the package starts each doc comment with the name, you'd see something like this, which recalls the word you're looking for.

```
$ godoc regexp | grep parse  
    Compile parses a regular expression and returns, if successful, a Regexp  
    parsed. It simplifies safe initialization of global variables holding  
    cannot be parsed. It simplifies safe initialization of global variables  
$
```

Go's declaration syntax allows grouping of declarations. A single doc comment can introduce a group of related constants or variables. Since the whole declaration is presented, such a comment can often be perfunctory.

```
// Error codes returned by failures to parse an expression.  
var (  
    ErrInternal          = errors.New("regexp: internal error")  
    ErrUnmatchedLpar    = errors.New("regexp: unmatched '('")  
    ErrUnmatchedRpar    = errors.New("regexp: unmatched ')'")  
    ...  
)
```


5.1.3 Uncommented Code Is Broken

Seriously, if your code doesn't have good doc comments, it is by definition lousy code.

Todo

Write a short diatribe on the importance of code commentary.

5.2 godoc tool

Godoc extracts and generates documentation for Go programs.²

It has two modes.

Without the `-http` flag, it runs in command-line mode and prints plain text documentation to standard output and exits. If both a library package and a command with the same name exists, using the prefix `cmd/` will force documentation on the command rather than the library package. If the `-src` flag is specified, godoc prints the exported interface of a package in Go source form, or the implementation of a specific exported language entity:

```
godoc fmt                # documentation for package fmt
godoc fmt Printf          # documentation for fmt.Printf
godoc cmd/go             # force documentation for the go command
godoc -src fmt           # fmt package interface in Go source form
godoc -src fmt Printf     # implementation of fmt.Printf
```

With the `-http` flag, it runs as a web server and presents the documentation as a web page.

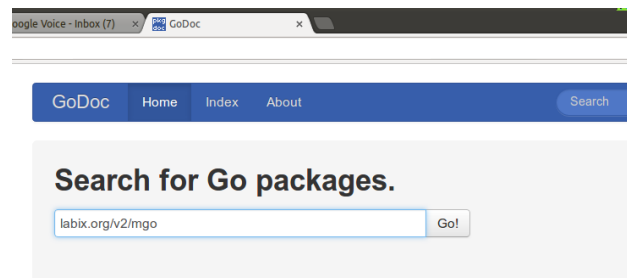
```
godoc -http=:6060
```

5.3 GoDoc.org

GoDoc is an open source web application that displays documentation for Go packages on Bitbucket, Github, Launchpad and Google Project Hosting. It is similar to the `godoc` command, but can display documentation for any open source Go module specified by its import URL.

5.3.1 Using GoDoc.org

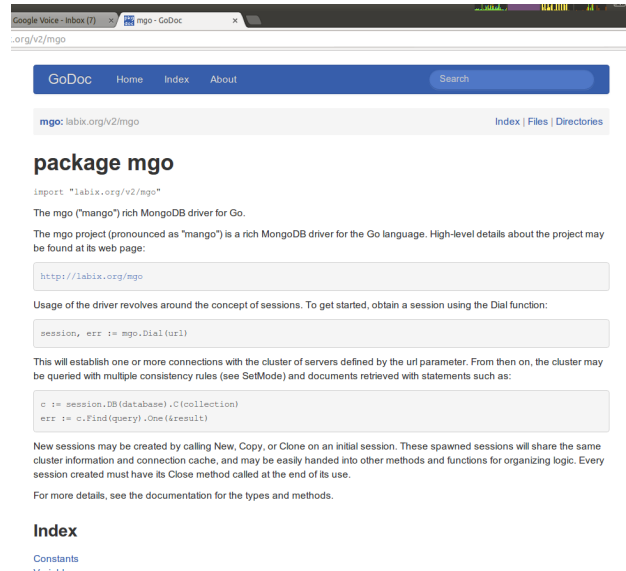
To view a package's documentation on GoDoc.org you enter its import path. Let's check out the docs for `mgo`, the popular MongoDB database driver. Its import path is `labix.org/v2/mgo`.



² The following section copied from the package comment of `godoc`

We could also have typed in just “mgo” and GoDoc would show us packages with that string in their name. However it does not attempt to rank results by relevance or popularity, so the actual `mgo` driver would not be one of the first few results.

Click “Go!” and GoDoc will display the documentation for our package. The documentation starts with the package’s name, import path, and package comment.



The screenshot shows the GoDoc website interface. At the top, there's a navigation bar with "GoDoc", "Home", "Index", "About", and a search bar. Below the navigation bar, the URL "mgo: labix.org/v2/mgo" is displayed. The main content area is titled "package mgo" and contains the following text:

```
import "labix.org/v2/mgo"
```

The mgo ("mango") rich MongoDB driver for Go.

The mgo project (pronounced as "mango") is a rich MongoDB driver for the Go language. High-level details about the project may be found at its web page:

```
http://labix.org/mgo
```

Usage of the driver revolves around the concept of sessions. To get started, obtain a session using the Dial function:

```
session, err := mgo.Dial(url)
```

This will establish one or more connections with the cluster of servers defined by the url parameter. From then on, the cluster may be queried with multiple consistency rules (see SetMode) and documents retrieved with statements such as:

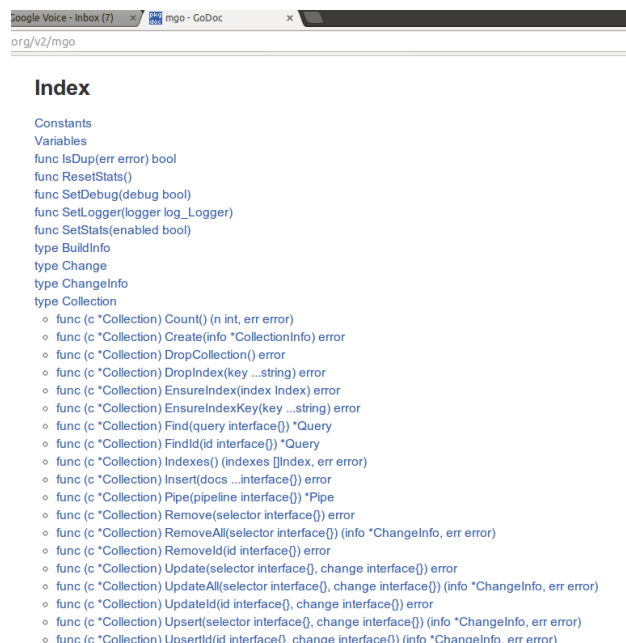
```
c := session.DB(database).C(collection)
err := c.Find(query).One(&result)
```

New sessions may be created by calling New, Copy, or Clone on an initial session. These spawned sessions will share the same cluster information and connection cache, and may be easily handed into other methods and functions for organizing logic. Every session created must have its Close method called at the end of its use.

For more details, see the documentation for the types and methods.

Below the main content, there is an "Index" section with a link to "Constants".

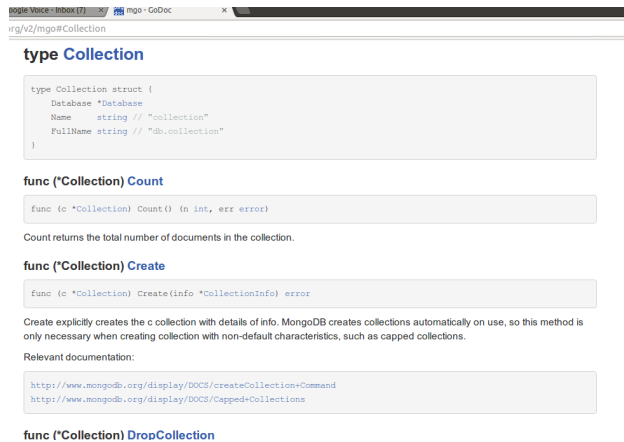
Up next is an alphabetical index of all the exported entities in the package - constants, variables, types, and functions. Methods are displayed beneath the type to which they are bound.



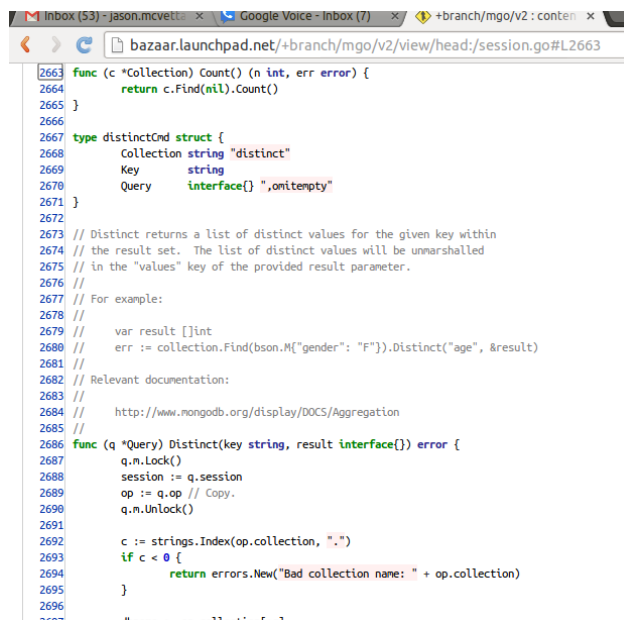
The screenshot shows the GoDoc website interface, specifically the "Index" section. The URL "org/v2/mgo" is displayed. The "Index" section lists the following entities:

- Constants
- Variables
- func IsDup(err error) bool
- func ResetStats()
- func SetDebug(debug bool)
- func SetLogger(logger log.Logger)
- func SetStats(enabled bool)
- type BuildInfo
- type Change
- type ChangeInfo
- type Collection
 - func (c *Collection) Count() (n int, err error)
 - func (c *Collection) Create(info *CollectionInfo) error
 - func (c *Collection) DropCollection() error
 - func (c *Collection) DropIndex(key ...string) error
 - func (c *Collection) EnsureIndex(index Index) error
 - func (c *Collection) EnsureIndexKey(key ...string) error
 - func (c *Collection) Find(query interface{}) *Query
 - func (c *Collection) FindId(id interface{}) *Query
 - func (c *Collection) Indexes() (indexes []Index, err error)
 - func (c *Collection) Insert(docs ...interface{}) error
 - func (c *Collection) Pipe(pipeline interface{}) *Pipe
 - func (c *Collection) Remove(selector interface{}) error
 - func (c *Collection) RemoveAll(selector interface{}) (info *ChangeInfo, err error)
 - func (c *Collection) RemoveId(id interface{}) error
 - func (c *Collection) Update(selector interface{}, change interface{}) error
 - func (c *Collection) UpdateAll(selector interface{}, change interface{}) (info *ChangeInfo, err error)
 - func (c *Collection) UpdateId(id interface{}, change interface{}) error
 - func (c *Collection) Upsert(selector interface{}, change interface{}) (info *ChangeInfo, err error)
 - func (c *Collection) UpsertId(id interface{}, change interface{}) (info *ChangeInfo, err error)

Click on the index entry for `Collection` to be taken to its detailed documentation. The exported fields and methods of structs are displayed, as well as the type signature of functions and methods. The entity’s doc comment is displayed, if it has one.



Sometimes the documentation is not enough, and we want to look at the source code. Click on the function name `Count` to see its source.



5.3.2 Open Source

GoDoc.org is open source! If you want to see how it works, check out <https://github.com/garyburd/gddo>!

5.3.3 Go Walker

Also check out [Go Walker](#), a new enhanced version of GoDoc that displays code snippets alongside the documentation.

Citations

The Go Toolchain

Go is a compiled language - the source code you write must be converted to a binary format understandable to a computer before it is executed. The compiler is accessed through the `go` command. In addition, the `go` command provides several other tools

The sections below describe several of the more popular `go` commands, but do not cover the complete set. For full information run:

```
$ go help
```

6.1 go build

Build compiles the packages named by the import paths, along with their dependencies, but it does not install the results.

If the arguments are a list of `.go` files, build treats them as a list of source files specifying a single package.

When the command line specifies a single main package, build writes the resulting executable to output. Otherwise build compiles the packages but discards the results, serving only as a check that the packages can be built.

```
$ go build somefile.go  # Build just this one file
$ go build               # Build package in current folder
```

6.1.1 godoc.org

The website godoc.org extracts and displays automatic documentation, much like the `godoc` command, for any open source Go module specified by its import URL.

See `GoDoc.org` for more detail.

Project Layout

Just as `$PYTHONPATH` controls where the Python interpreter will look for source files, `$GOPATH` controls where the Go compiler and tools will look for source code.

7.1 Hello World

Let us consider a modified version of the standard “Hello World” program. Our example will consist of a library package and an application package. The library exports one function, which returns the string “Hello world, the time is: ” plus the current time. The application package calls the library function, and prints the hello message to the console.

7.1.1 Python

In Python we might have a file layout like this:

```
$PYTHONPATH/  
    hello.py  
    greeter.py
```

The library package is contained in `greeter.py`:

The application is contained in `hello.py`:

```
1  #!/usr/bin/env python  
2  
3  import time  
4  
5  
6  def greeting():  
7      '''Returns a pleasant, semi-useful greeting.'''  
8      return "Hello world, the time is: " + time.ctime()  
9  
10  
11 def main():  
12     print greeting()  
13  
14  
15 if __name__ == '__main__':  
16     main()
```

We run the Python application like this:

```
$ python hello.py
Hello world, the time is: Mon Jul  8 19:16:40 2013
```

7.1.2 Go

The “standard” layout for Go code is more complex than that for Python code. The disadvantage is that setup is slightly more work. The upside is a well-structured codebase that encourages modular code and keeps things orderly as a project grows in size.

Typically Go code is developed using [distributed version control systems](#) like [Git](#) or [Mercurial](#). Therefore import paths are conventionally named based on the Github etc URL. The Go toolchain is aware of this, and can gracefully handle automatic dependency installation if your project conforms to the convention.

Imagine for a moment we have created a Github repository named `hello` for our Hello World example. We would create a file layout like this:

```
$GOPATH/
  src/
    github.com/
      jmcvetta/
        hello/
          hello.go
          greeter/
            greeter.go
```

The library is located in `greeter.go`. Note the package name, `greeter`, is the same as the name of the containing folder. This is mandatory. The package imports `time` from the standard library.

The application is in `hello.go`. This file declares its package as `main`, and defines a `main()` function. This tells the compiler to generate an executable from the file. The package imports `fmt` from the standard library, and our `greeter` package specified by its path.

Functions imported from another package are *always* namespaced with the package name. In this case, we call `greeter.Greeting()`. Go intentionally has no equivalent of Python’s `from some_package import *`.

We can build the application with the `go build` command:

```
$ go build -v
github.com/jmcvetta/hello/greeter
github.com/jmcvetta/hello

$ ls
greeter/  hello*  hello.go

$ ./hello
Hello world, the time is: 2013-07-08 19:49:39.946836748 -0700 PDT
```

7.2 Dependencies

The `go` tool can automatically install dependencies. They are installed in the same URL-derived folder heirarchy alongside your code.

Let’s embellish our `Greeting()` function by making it return the current `PATH` as well. Although this could be done using nothing but the standard library, for purpose of instruction we will use the popular [env](#) package.


```
1 package greeter
2
3 import (
4     "github.com/darkhelmet/env"
5     "time"
6 )
7
8 // Greeting returns a pleasant, semi-useful greeting.
9 func Greeting() string {
10     msg := "Hello world, the time is "
11     msg += time.Now().String()
12     msg += " and your PATH is "
13     msg += env.String("PATH")
14     return msg
15 }
```

We can automatically install our new dependency with the `go get` command:

```
$ go get -v .
github.com/darkhelmet/env (download)
github.com/darkhelmet/env
github.com/jmcvetta/hello/greeter
github.com/jmcvetta/hello
```

Now we the `env` package installed, and our file system looks like:

```
$GOPATH/
src/
  github.com/
    darkhelmet/
      env/
        env.go
        env_test.go
        format.bash
        LICENSE.md
        README.md
    jmcvetta/
      hello/
        hello.go
        greeter/
          greeter.go
```

We can try out our new Hello World with the `go run` command, which build the application then runs it.

```
$ go run hello.go
Hello world, the time is 2013-07-08 20:36:31.496236239 -0700 PDT and your PATH is /home/jason/.rvm/g
```

- *Search this Book*