1) Formatting: No need to worry of different people working with separate styles.
`gofmt <file_name>.go` => formats the source file
`go fmt <path_to_package>` => formats all files in package

=> easy to write, easy to read, easy to maintain
=> golint: Golint differs from gofmt. Gofmt reformats Go source code, whereas golint prints out style mistakes. The output of this tool is a list of suggestions in Vim quickfix format, which is accepted by lots of different editors.
reference
=> govet: Govet does simple checking of Go source code e.g errors in Println, etc.
usage:
govet [flag] [file.go ...]
govet [flag] [directory ...] # Scan all .go files under directory, recursively
reference

2) Commentary:
/* <comment> */ => for block comments
// <comment> => for single line comments

a) Every package should have '*package comment'* preceding the package clause
e.g
```
/*
  this is a documentary package
*/
package documentary

 or
  // this is a documentary package (if brief)
```

b) For multi-file package, info in one is sufficient
c) Every exported(Capitalized) name should have doc comment, starting with name being declared.
e.g
```
// Parser enhances the query from input url after
// filtering the facets
func Parser(){}
```
d) doc.go : For large packages comment is placed in its own file, e.g doc.go, which contains only those comments and a package clause.

3) Naming:
a) Visibility of a name outside package is determined by whether its first character is upper case.

b) package name: (short, lower-case, single word) => easier for importers and less confusion

c) inferface name: if only single method, then similar to it
<method-name> + ‘er’
d) MixedCaps: preferred UseIt or useIt, rather than undersocre usages.
e) Source file name: lower case with underscores(if necessary)e.g driver.go,
driver_query.go

4) Semicolons:
Go's formal grammar uses semicolons to terminate statements, but those semicolons do not appear in the source
Rule: “if the newline comes after a token that could end a statement, insert one”

=> One consequence of the semicolon insertion rules is that you cannot put the opening brace of a control structure (if, for, switch, or select) on the next line. So use like this:

```
if i < f() {
  g()
}
```

not like this:

```
if i < f()  // wrong!
{           // wrong!
  g()
}
```

5) Multiple return values:
=> Functions and methods can return multiple values.
e.g

```
func readData() (int, string){}
num, name := readData()
```

6) Defer: schedules a function call to be run immediately before current call return.
e.g

```
// Contents returns the file's contents as a string.
func Contents(filename string) (string, error) {
    f, err := os.Open(filename)
    if err != nil {
        return "", err
    }
    defer f.Close()  // f.Close will run when we're finished.
    var result []byte
    buf := make([]byte, 100)
    for {
        n, err := f.Read(buf[0:])
        result = append(result, buf[0:n]...) // append is discussed later.
        if err != nil {
            if err == io.EOF {
                break
            }
            return "", err  // f will be closed if we return here.
        }
    }
    return string(result), nil // f will be closed if we return here.
}
```

=> will never forget to close the file
=> clearer as it sits near open rather than at end
Note: Defer is for your help, not necessary

7) Data Allocation:

a) new:
- allocates zeored storage
- returns pointer

func new(Type) *Type
e.g

```
type Car struct{
     field1 bool
}
var myCar = new(Car)
```

b) make:
- used for initializes slice, map and channel types
- returns type

e.g

```
func make(Type, size IntegerType) Type
v := make([]int, 100)
```

Note: No type hierarchy

8) Unused import and variables, underscore usage:

-> any unused local variable or unused imported package will give compile time errors.
But even though sometimes we require some of them for using it later, so there is a cleaner way.

```
package main

import (
    "fmt"
    "io"
    "log"
    "os"
)
```

```
var _ = fmt.Printf // For debugging; delete when done.
var _ io.Reader    // For debugging; delete when done.

func main() {
    fd, err := os.Open("test.go")
    if err != nil {
        log.Fatal(err)
    }
    // TODO: use fd.
    _ = fd
}
```

9) Error Strings: What is the best practices for error handling when to use our own custom error rather than builtin error? http://golang.org/doc/faq#nil_error http://blog.golang.org/error-handling-and-go

Error strings should not be capitalized (unless beginning with proper nouns or acronyms) or end with punctuation, since they are usually printed following other context. That is, use fmt.Errorf("something bad") not fmt.Errorf("Something bad")

10) Handle Errors, Fatal, Panic:

It is expected in Go to check the error at first place itself rather than throw up and wait. Also to avoid repetitive error handling code, an error struct should be there which should be called
from different packages.

Do not discard errors using _ , rather check whether run was successful or not.
=> By default there is in-built error interface, use it.

```
type error interface {
       Error() string
}

type PathError struct {
      Op string    // "open", "unlink", etc.
      Path string  // The associated file.
      Err error    // Returned by the system call.
 }

func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

=> Panic: Stops current execution, unwind stacks, run deferred calls if any and if reaches top stack, program dies. Use only and only if no other way and program has to shutdown.
e.g initialization call

```
var user = os.Getenv("USER")

func init() {
    if user == "" {
        panic("no value for $USER")
    }
}
```

=> Fatal: logs the message and terminates the program instantly.
e.g log.Fatal(err)

11) Pointers can be used to change the field values in methods. Using values will not reflect the change in caller objects.
e.g

```
type Car struct {
    field1 string
}
type Vehicle interface{
  setfield1()
}
// this will not retail changes and it will be local only
func (c Car) setfield1{
  c.field1 = "xyz"
}
// this will retain the changes in caller's car object
func(c *Car)setfield1{
  c.field1 = "xyz"
}
var myCar A
myCar = new(Car) // gives a pointer
```

For more details:

1) http://golang.org/doc/faq
2) http://golang.org/doc/effective_go.html