

01	☐ A	☐ B	☐ C	☐ D	☐ E
02	☐ A	☐ B	☐ C	☐ D	☐ E
03	☐ A	☐ B	☐ C	☐ D	☐ E
04	☐ A	☐ B	☐ C	☐ D	☐ E
05	☐ A	☐ B	☐ C	☐ D	☐ E
06	☐ A	☐ B	☐ C	☐ D	☐ E
07	☐ A	☐ B	☐ C	☐ D	☐ E
08	☐ A	☐ B	☐ C	☐ D	☐ E
09	☐ A	☐ B	☐ C	☐ D	☐ E
10	☐ A	☐ B	☐ C	☐ D	☐ E
11	☐ A	☐ B	☐ C	☐ D	☐ E
12	☐ A	☐ B	☐ C	☐ D	☐ E
13	☐ A	☐ B	☐ C	☐ D	☐ E

14	☐ A	☐ B	☐ C	☐ D	☐ E
15	☐ A	☐ B	☐ C	☐ D	☐ E
16	☐ A	☐ B	☐ C	☐ D	☐ E
17	☐ A	☐ B	☐ C	☐ D	☐ E
18	☐ A	☐ B	☐ C	☐ D	☐ E
19	☐ A	☐ B	☐ C	☐ D	☐ E
20	☐ A	☐ B	☐ C	☐ D	☐ E
21	☐ A	☐ B	☐ C	☐ D	☐ E
22	☐ A	☐ B	☐ C	☐ D	☐ E
23	☐ A	☐ B	☐ C	☐ D	☐ E
24	☐ A	☐ B	☐ C	☐ D	☐ E
25	☐ A	☐ B	☐ C	☐ D	☐ E
26	☐ A	☐ B	☐ C	☐ D	☐ E

27	☐ A	☐ B	☐ C	☐ D	☐ E
28	☐ A	☐ B	☐ C	☐ D	☐ E
29	☐ A	☐ B	☐ C	☐ D	☐ E
30	☐ A	☐ B	☐ C	☐ D	☐ E
31	☐ A	☐ B	☐ C	☐ D	☐ E
32	☐ A	☐ B	☐ C	☐ D	☐ E
33	☐ A	☐ B	☐ C	☐ D	☐ E
34	☐ A	☐ B	☐ C	☐ D	☐ E
35	☐ A	☐ B	☐ C	☐ D	☐ E
36	☐ A	☐ B	☐ C	☐ D	☐ E
37	☐ A	☐ B	☐ C	☐ D	☐ E
38	☐ A	☐ B	☐ C	☐ D	☐ E
39	☐ A	☐ B	☐ C	☐ D	☐ E

PRACTICE QUESTIONS

University of Panda GO Class Final Assessment

G400

GO 400 Exam

EDITION

Demo

QUESTIONS

9

FORMAT

Limited Edition PDF

Before you begin

What this is

9 practice questions for University of Panda GO Class Final Assessment **G400**, each with its answer key and an explanation. Answer a question before you read its key — the explanation is worth more once you have committed.

If something looks wrong

Send us three things and we will check it:

- the exam code, **G400**
- the question number
- the email address on your order

Write to **support@mometrix.net**. We reply within one business day.

QUESTION 1

SINGLE CHOICE

Which design principle is most central to Go's approach to software construction?

- ☐ A Maximizing runtime flexibility through dynamic typing and late binding
- ☒ B Combining statically typed compilation with lightweight concurrency primitives and a comprehensive standard library
- ☐ C Embedding virtual machine technologies to enable cross-platform bytecode execution
- ☐ D Relying on extensive third-party framework configuration to deliver production services

ANSWER B

WHY Go was designed as a compiled, statically typed language that emphasizes simplicity, fast builds, built-in concurrency (goroutines and channels), and a batteries-included standard library. Options A and C describe languages like Python or Java, and option D contradicts Go's minimal-dependency philosophy.

QUESTION 2

SINGLE CHOICE

Given the following snippet, what is the value and type of x? `x := 3.14 y := x y = 7.5 fmt.Println(x)`

- ☐ A 7.5 (float64)
- ☒ B 3.14 (float64)
- ☐ C 7.5 (float32)
- ☐ D A compilation error because x is never used after assignment to y

ANSWER B

WHY x is a float64 inferred from the literal 3.14. Assigning x to y copies the value because float64 is a value type. Changing y afterwards does not affect x, so the program prints 3.14. Go also does not error on unused variables that are used, only on declared-and-never-used variables; x is referenced inside Println.

QUESTION 3

SINGLE CHOICE

What does the following Go program print? `func main() { defer fmt.Println("A") fmt.Println("B") defer fmt.Println("C") fmt.Println("D") }`

- ☐ A A B C D
- ☒ B B D C A
- ☐ C B D A C
- ☐ D C A B D

ANSWER B

WHY Deferred calls run in LIFO order when the surrounding function returns. The synchronous output prints B then D, and then the deferred calls run from most recent to oldest: C then A. This yields B D C A.

QUESTION 4

SINGLE CHOICE

A method is declared on a struct type using a pointer receiver. Which statement about pointer receivers is correct?

- ☐ A Pointer receivers receive a copy of the original value, so mutations do not affect the caller
- ☐ B Pointer receivers are required only when the method needs to satisfy a generic type constraint
- ☒ C Pointer receivers allow the method to mutate the original value and avoid copying large structs on each call
- ☐ D Pointer receivers can only be used on types defined in the same package as the calling code

ANSWER C

WHY A pointer receiver (e.g., `func (p *T) M()`) gives the method access to the original value through the pointer, enabling mutation and avoiding copies of potentially large structs. Go will automatically take the address of an addressable value when calling a pointer-receiver method, so usage is ergonomic.

QUESTION 5

SINGLE CHOICE

Which statement best describes the behavior of an unbuffered channel in Go?

- ☒ **A** Sends and receives block until both a sender and receiver are ready on the same channel
- ☐ **B** Sends succeed immediately even when no receiver is waiting, and values are dropped if unconsumed
- ☐ **C** Receivers always return the zero value immediately if no sender is currently blocked
- ☐ **D** Sends and receives are non-blocking but values may be reordered across goroutines

ANSWER A

WHY An unbuffered channel synchronizes sender and receiver: a send blocks until a receiver is ready, and a receive blocks until a sender is ready. This is the textbook definition used in Go to model synchronization and hand-off of ownership.

QUESTION 6

SINGLE CHOICE

Which code correctly registers an HTTP handler for the path '/health' that responds with status 200 and the body 'ok' using the net/http package?

- ☐ **A** `http.RegisterHandler('/health', func(w http.ResponseWriter, r *http.Request) { w.Write([]byte('ok')) })`
- ☒ **B** `http.HandleFunc("/health", func(w http.ResponseWriter, r *http.Request) { w.Write([]byte("ok")) })`
- ☐ **C** `http.Route("/health", "ok")`
- ☐ **D** `http.ServeFile("/health", "ok.txt")`

ANSWER B

WHY `http.HandleFunc` registers a handler function for a pattern on the default `ServeMux`. The handler signature is `func(http.ResponseWriter, *http.Request)`. The body is written with `w.Write`, and the implicit status is 200 OK. The other options reference functions that do not exist or misuse APIs.

QUESTION 7

SINGLE CHOICE

Which is the idiomatic way to define a function in Go that may fail, and to surface that failure to its caller?

- ☐ A Throw an exception using panic and expect callers to catch it with a try/catch block
- ☒ B Return a tuple where the last value is an error of interface type, and the caller checks if it is non-nil
- ☐ C Print the failure to stderr and terminate the process from inside the callee
- ☐ D Return a boolean named 'ok' and store the failure message in a package-level variable

ANSWER B

WHY Go uses explicit, multiple return values, with the conventional last return being an error. Callers check `if err != nil`. Go does not have exceptions in the try/catch sense; panic/recover is reserved for genuinely unrecoverable conditions. Returning a boolean or printing to stderr is not idiomatic.

QUESTION 8

SINGLE CHOICE

Which line correctly declares a string variable named greeting and assigns it the value "hello"?

- ☒ A `var greeting string = "hello"`
- ☐ B `greeting := hello`
- ☐ C `string greeting = "hello"`
- ☐ D `var greeting = hello`

ANSWER A

WHY Option A uses the standard var declaration with the explicit type string and a string literal in quotes. Option B is missing quotes around hello and would be parsed as an identifier. Option C uses syntax from C/Java, not Go. Option D is missing quotes and would treat hello as an undeclared identifier.

QUESTION 9

SINGLE CHOICE

Which statement about Go slices is correct?

- ☐ A A slice's length is fixed at compile time and cannot change.
- ☐ B A slice is a value type that has no relationship to any underlying array.
- ☒ C The built-in `append` function may return a new slice value when the underlying array has insufficient capacity.
- ☐ D Two slices that share the same length always share the same backing array.

ANSWER C

WHY `append` may grow the slice by allocating a new underlying array and returning a slice that references it; this is the standard, documented behavior. Option A describes an array, not a slice. Option B is wrong because a slice header references an underlying array. Option D is wrong because two slices can have the same length but point to different backing arrays.