

A function can take zero or more arguments.
In this example, `add` takes two parameters of type `int`.
Notice that the type comes _after_ the variable name.
(For more about why types look the way they do, see the [[/blog/gos-declaration-syntax][article on Go's declaration syntax]].)
* Functions continued
When two or more consecutive named function parameters share a type, you can omit the type from all but the last.
In this example, we shortened
x int, y int

* Functions

to

*	M	ul	lti	a	le	re	sι	ults
---	---	----	-----	---	----	----	----	------

A function can return any number of results.

The `swap` function returns two strings.

* Named return values

Go's return values may be named. If so, they are treated as variables defined at the top of the function.

These names should be used to document the meaning of the return values.

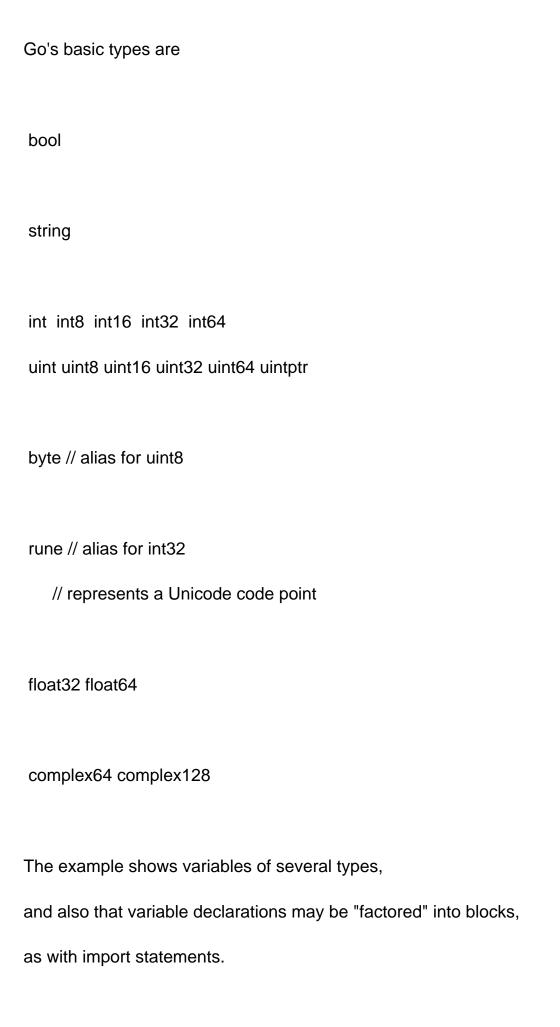
A `return` statement without arguments returns the named return values. This is known as a "naked" return.

Naked return statements should be used only in short functions, as with the example shown here. They can harm readability in longer functions.

* Variables

The `var` statement declares a list of variables; as in function argument lists, the type is last.
A `var` statement can be at package or function level. We see both in this example.
* Variables with initializers
A var declaration can include initializers, one per variable.
If an initializer is present, the type can be omitted; the variable will take the type of the initializer.
* Short variable declarations
Inside a function, the `:=` short assignment statement can be used in place of a `var` declaration with implicit type.
Outside a function, every statement begins with a keyword (`var`, `func`, and so on) and so the `:=` construct is not available.

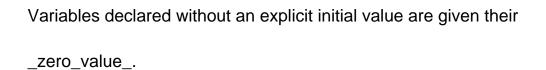
* Basic types



The `int`, `uint`, and `uintptr` types are usually 32 bits wide on 32-bit systems and 64 bits wide on 64-bit systems.

When you need an integer value you should use `int` unless you have a specific reason to use a sized or unsigned integer type.





The zero value is:

- `0` for numeric types,
- `false` for the boolean type, and
- `""` (the empty string) for strings.

* Type conversions

The expression `T(v)` converts the value `v` to the type `T`.

Some numeric conversions:

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)

Or, put more simply:

i := 42
f := float64(i)
u := uint(f)

Unlike in C, in Go assignment between items of different type requires an
```

Unlike in C, in Go assignment between items of different type requires an explicit conversion.

Try removing the `float64` or `uint` conversions in the example and see what happens.

* Type inference

When declaring a variable without specifying an explicit type (either by using the `:=` syntax or `var`=` expression syntax), the variable's type is inferred from the value on the right hand side.

When the right hand side of the declaration is typed, the new variable is of that same type:

var i int

```
j := i // j is an int
```

But when the right hand side contains an untyped numeric constant, the new variable may be an `int`, `float64`, or `complex128` depending on the precision of the constant:

Try changing the initial value of `v` in the example code and observe how its type is affected.

* Constants

Constants are declared like variables, but with the `const` keyword.

Constants can be character, string, boolean, or numeric values.

Constants cannot be declared using the `:=` syntax.

* Numeric Constants

Numeric constants are high-precision _values
An untyped constant takes the type needed by its context.
Try printing `needInt(Big)` too.
(An `int` can store at maximum a 64-bit integer, and sometimes less.)
* Congratulations!
You finished this lesson!
You can go back to the list of [[/tour/list][modules]] to find what to learn next, or continue with the [[javascript:click('.next-page')][next lesson]].