

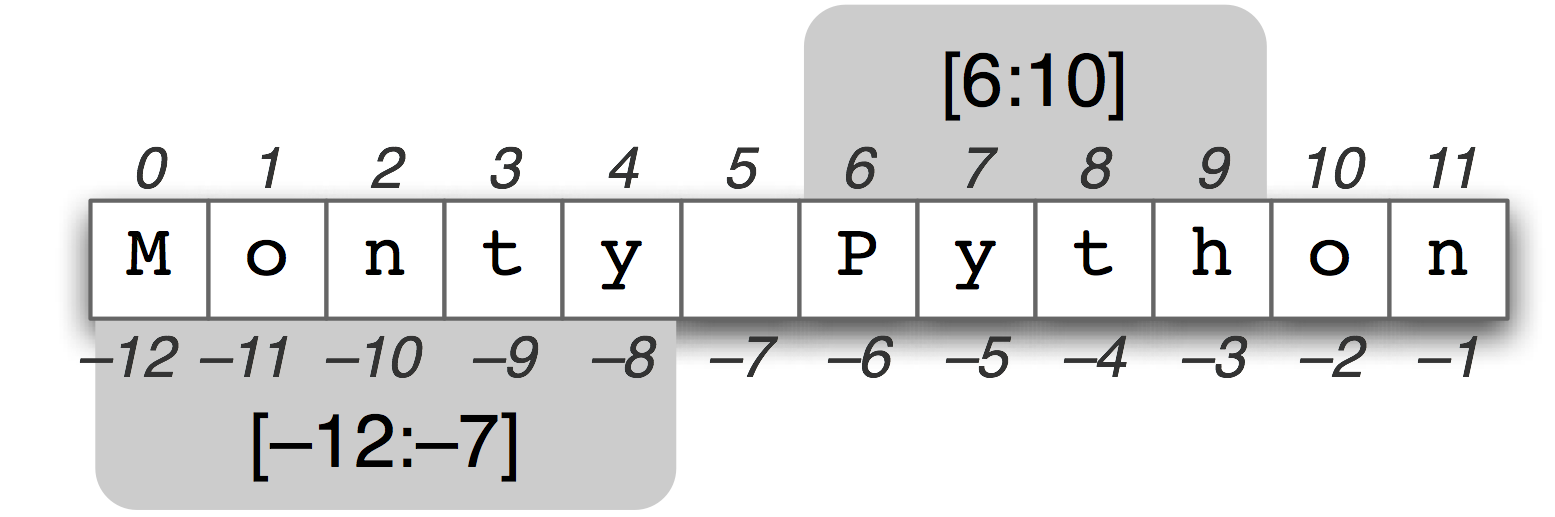
History and Overview

* Interactive, Interpreted, Object Oriented scripting language
  + Interacted with Python shell
* Python 2 vs Python 3
  + can only be used inside func
* Syntax and executing cmds, and scripts using shebang #!

*False class finally is return  
# None continue for lambda try  
# True def from nonlocal while  
# and del global not with  
# as elif if or yield  
# assert else import pass  
# break except in raise*

Variables

* Indentifiers
  + A-Z , a-z, or \_ or anything except @,$, %
  + Naming Conventions – Class Names
* Lines and Indentions
  + No {} for class, function, blocks, flow controls
  + Should same indentations for all statements in a block
* Variables reserved memory locations to store values
  + No explicit declaration, allocates the sufficient memory based on the type
  + = used to assign values to variables
  + multiple assignments
* Data types
* Numerics
* String types



Data type conversions

* Operators
  + Arithmetic Operators
  + Comparison (Relational) Operators
  + Assignment Operators
  + Logical Operators
  + Bitwise Operators
  + Membership Operators
  + Identity Operators

Function

## **Syntax of Function**

def function\_name(parameters):

"""docstring"""

statement(s)

Above shown is a function definition which consists of following components.

1. Keyword **def** marks the start of function header.
2. A function name to uniquely identify it. Function naming follows the same [rules of writing identifiers in Python](http://www.programiz.com/python-programming/keywords-identifier#rules).
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional **return** statement to return a value from the function.

Factorial Recursion

calc\_factorial(4) # 1st call with 4

4 \* calc\_factorial(3) # 2nd call with 3

4 \* 3 \* calc\_factorial(2) # 3rd call with 2

4 \* 3 \* 2 \* calc\_factorial(1) # 4th call with 1

4 \* 3 \* 2 \* 1 # return from 4th call as number=1

4 \* 3 \* 2 # return from 3rd call

4 \* 6 # return from 2nd call

24 # return from 1st call

|  |  |
| --- | --- |
| Python File Modes | |
| Mode | Description |
| 'r' | Open a file for reading. (default) |
| 'w' | Open a file for writing. Creates a new file if it does not exist or truncates the file if it exists. |
| 'x' | Open a file for exclusive creation. If the file already exists, the operation fails. |
| 'a' | Open for appending at the end of the file without truncating it. Creates a new file if it does not exist. |
| 't' | Open in text mode. (default) |
| 'b' | Open in binary mode. |
| '+' | Open a file for updating (reading and writing) |

* + const typedHello string = "Hello, World"

**public** **static** **final** **int** MAX\_VAL = 10;

**public** **static** **final** **boolean** **isOK** = **true**;

* UNTYPED constant
  + a constant value that does not yet have a fixed type
    - a “kind”
    - not yet forced to obey the strict rules that prevent combining differently typed values
* It is this notion of an *untyped* constant that makes it possible for us to use constants in Go with great freedom.
* This is useful, for instance
  + what is the type of 42?
    - int?
    - uint?
    - float64?
  + if we didn’t have UNTYPED constants (constants of a kind), then we would have to do conversion on every literal value we used
    - and that would suck

Day#4

Scope Recap

1. **NO** ***public/private***keywords for visibility of an identifier. If the initial character is an upper case letter, the identifier **is *exported*(public); otherwise it is not**:

* upper case initial letter: Name is visible to clients of package
* otherwise: name (or \_Name) is not visible to clients of package

This rule applies to variables, types, functions, methods, constants, fields.

2. Go has **scope hierarchy**

* **universe** (predeclared identifiers such as int and string)
* **package** (all the **source** files of a package live at the same scope)
* **file** (for package import renames only; not very important in practice)
* function (the usual)
* **block** (the usual)

There is **no scope for name space or class or other wrapping construct**.

The only exception is **statement labels**, the targets of **break** statements and the like; they always have function scope.

4. There is **no implicit this**. Methods in GO declare an **explicit** **receiver** ( eg reciever.Method()). The receiver must be used to access fields and methods of the type.

5. These rules provide an important property **for scaling** because they guarantee that **adding an exported name to a package can never break a client** of that package. The naming rules **decouple** packages, providing scaling, clarity, and robustness.

6. Method lookup is always **by name** only, not **by signature** (**type**) of the method.

* A single type can never have two methods with the same name. Given a method x.M, there's only ever one M associated with x.

Control Flow

For Loop

* resources
  + <https://forum.golangbridge.org/>
  + Dave Cheney’s Blog
* for loop
  + documentation
    - language spec
    - effective go
  + initialization, condition, post

Nested Loops

* for loop
  + nested loops

Conditions, Break, & Continue

* for loop
  + condition
  + no condition
  + break
  + continue

**Day#5**

Switch Statements

* switch / case / default statements
  + no default fall-through
  + creating fall-through
  + multiple cases
  + cases can be expressions
    - evaluate to true, they run
  + type

The first one I was pretty used to, [**type conversion**](http://golang.org/ref/spec#Conversions)**:**

|  |  |
| --- | --- |
| **1**  **2** | **var x int = 5**  **var x64 int64 = int64(x)** |

The one I was unaware of is [**type assertion**](http://golang.org/ref/spec#Type_assertions)**:**

|  |  |
| --- | --- |
| **1**  **2** | **var x interface{} = 5**  **var x64 int64 = x.(int64)** |

So what’s the difference? Well **conversion** should be used when you are dealing with a **type**, whether that be a constant or struct or whatever. **Assertion** is used when when you’re dealing with an **interface & only Interface**. Say for example a method returns an **interface** instead of a **struct**, there’s still a value associated with the return but it has a generic type. You can use assertion to convert the return value to any type that implements that interface.

If Statements

* the not operator
  + !
* initialization statement
* if / else
* if / else if / else
* if / else if / else if / … / else

Intro To Functions

* func main
* func syntax
  + func, receiver, identifier, params, returns
* purpose of functions
  + abstract code
  + code reusability
* parameters vs. arguments
* declaring a func with multiple params

Func Returns

* function syntax when declaring a func
* a single return
* a named return
* multiple returns

Variadic Functions

* a func that accepts an unlimited number of parameters

The behavior of defer statements is straightforward and predictable. There are three simple rules:

1. *A deferred function's arguments are evaluated when the defer statement is evaluated.*

In this example, the expression "i" is evaluated when the Println call is deferred. The deferred call will print "0" after the function returns.

func a() {

i := 0

defer fmt.Println(i)

i++

return

}

2. *Deferred function calls are executed in Last In First Out order after the surrounding function returns.*

This function prints "3210":

func b() {

for i := 0; i < 4; i++ {

defer fmt.Print(i)

}

}

3. *Deferred functions may read and assign to the returning function's named return values.*

In this example, a deferred function increments the return value i *after* the surrounding function returns. Thus, this function returns 2:

func c() (i int) {

defer func() { i++ }()

return 1

}

This is convenient for modifying the error return value of a function; we will see an example of this shortly.

# DATA STRUCTURES in GO lang

* array
  + a numbered sequence of elements of a single type
  + does not change in size
  + <https://golang.org/ref/spec#Array_types>
* slice
  + a list
  + A slice is a descriptor for a contiguous segment of an underlying array and provides access to a numbered sequence of elements from that array. A slice type denotes the set of all slices of arrays of its element type. **The value of an uninitialized slice is nil.**
  + change in size
  + have a length and a capacity
  + multi-dimensional
  + <https://golang.org/ref/spec#Slice_types>
* map
  + key / value storage
  + a “dictionary”
  + A map is an unordered group of elements of one type, called the element type, indexed by a set of unique keys of another type, called the key type. **The value of an uninitialized map is nil.**
  + <https://golang.org/ref/spec#Map_types>
* struct
  + a data structure
  + a composite type
  + allows us to collect properties together
  + <https://golang.org/ref/spec#Struct_types>

**Array**

* definition
  + An array is a numbered sequence of elements of a single type.
  + The number of elements is called the length and is never negative.
  + The **length is part of the array's type**; it must evaluate to a non-negative constant representable by a value of type int.
    - The length of an array a can be discovered using the built-in function len.
  + The elements can be addressed by integer indices 0 through len(a)-1.
  + Array types are always one-dimensional but may be composed to form multi-dimensional types.
  + not [dynamic](https://en.wikipedia.org/wiki/Dynamic_array)
    - does not change in size
* a basic array
  + len
  + index access
  + assigning a value to an index position in an array

Slice

* definition
  + A slice is a descriptor for a contiguous segment of an underlying array and provides access to a numbered sequence of elements from that array.
  + The value of an uninitialized slice is nil.
    - **it is a reference type**
  + Like arrays, slices are indexable and have a length.
  + The length of a slice s can be discovered by the built-in function len;
    - Unlike arrays, slices are dynamic
      * their length may change during execution.
  + The elements can be addressed by integer indices 0 through len(s)-1.
  + A slice, once initialized, is always associated with an underlying array that holds its elements.
    - **it is a reference type**
  + The array underlying a slice may extend past the end of the slice.
    - Capacity is a measure of that extent:
      * it is the sum of the length of the slice and the length of the array beyond the slice;
    - The capacity of a slice a can be discovered using the built-in function cap(a).
  + make
    - A new, initialized slice value for a given element type T is made using the built-in function make, which takes a slice type and parameters specifying the length and optionally the capacity.
    - A slice created with make always allocates a new, hidden array to which the returned slice value refers.
      * make([]T, length, capacity)
      * make([]int, 50, 100)
        + same as this: new([100]int)[0:50]
  + Like arrays, slices are always one-dimensional but may be composed to construct higher-dimensional objects. (multi-dimensional slices)
* a basic slice

Structs Introduction

* introduction to struct
* user-defined types
  + creating values of a certain type
* an example struct
* OOP
  + in other languages
    - classes - blueprint
      * instantiate objects from a class
  + in Go
    - types - blueprint
      * create values of that type

OOP in Go

* Documentation
  + golang spec
    - A struct is a sequence of named elements, called fields, each of which has a name and a type. Field names may be specified explicitly (IdentifierList) or implicitly (AnonymousField). Within a struct, non-blank field names must be unique.
    - A field declared with a type but no explicit field name is an anonymous field, also called an embedded field or an embedding of the type in the struct. An embedded type must be specified as a type name T or as a pointer to a non-interface type name \*T, and T itself may not be a pointer type. The unqualified type name acts as the field name.
    - Promoted
* Resources
  + <http://www.goinggo.net/2015/09/composition-with-go.html>
* OOP
  + Encapsulation
    - state ("fields")
    - behavior ("methods")
    - exported / un-exported
  + Reusability
    - inheritence ("embedded types")
  + Polymorphism
    - interfaces
  + Overriding
    - embedding types as fields in a struct
      * anonymous types
    - outer type and inner type
      * inner type is promoted to the outer type
    - syntactic sugar
    - overriding fields and methods

User-Defined Types

* user-defined types
  + Go is statically typed
* language for talking about types and structs

Composition

* methods
* embedded types
* struct pointers

# CONCURRENCY

Concurrency & WaitGroup

* go routines
* waitgroups
* A WaitGroup waits for a collection of goroutines to finish. The main goroutine calls Add to set the number of goroutines to wait for. Then each of the goroutines runs and calls Done when finished. At the same time, Wait can be used to block until all goroutines have finished.

Parallelism

* But when people hear the word *concurrency* they often think of *parallelism*, a related but quite distinct concept. In programming, concurrency is the *composition* of independently executing processes, while parallelism is the simultaneous *execution* of (possibly related) computations. Concurrency is about *dealing with* lots of things at once. Parallelism is about *doing* lots of things at once.

Race Condition

* [Race conditions](http://en.wikipedia.org/wiki/Race_condition) are among the most insidious and elusive programming errors. They typically cause erratic and mysterious failures, often long after the code has been deployed to production. While Go's concurrency mechanisms make it easy to write clean concurrent code, they don't prevent race conditions. Care, diligence, and testing are required. And tools can help.

Mutex

* A Mutex is a mutual exclusion lock. Mutexes can be created as part of other structures; the zero value for a Mutex is an unlocked mutex.

Atomicity

* Package atomic provides low-level atomic memory primitives useful for implementing synchronization algorithms.
* These functions require great care to be used correctly. Except for special, low-level applications, synchronization is better done with channels or the facilities of the sync package. Share memory by communicating; don't communicate by sharing memory.

Channels Introduction

* making a channel
* putting values on a channel
* taking values off of a channel
* buffered and unbuffered channels
* unbuffered channels
  + block
  + they are like runners in a relay race
    - they are synchronized
    - they have to pass/receive the value at the same time
      * just like runners in a relay race have to pass / receive the baton to each other at the same time
        + one runner can’t pass the baton at one moment
        + and then, later, have the other runner receive the baton
        + the baton is passed/received by the runners at the same time
      * this is the same with unbuffered channels
      * the value is passed/received synchronously; at the same time
* channels allow us to pass values between goroutines

Range Clause

* putting values onto a channel
* taking values off of a channel
* closing a channel
* range clause
  + the range clause takes values off a channel
  + when the channel is closed,
    - you can no longer put values on a channel
    - you can still pull any values already on the channel off of the channel
  + when a channel is closed and there are no more values on it
    - the range clause completes
    - flow of the program continues sequentially