LEARN PYTH
QUICKLY

CODING FOR BEGINI
WITH HANDS ON PR

WRITTEN BY
JJ TAM

EDITED BY
TAM SEL

LEARN GOLANG QUICKLY

CODING FOR BEGINNERS
WITH HANDS ON PROJECTS

WRITTEN BY
JJ TAM

EDITED BY
TAM SEL

LEARN
GOLANG AND PYTHON
QUICKLY

CODING FOR BEGINNERS
WITH HANDS ON PROJECTS
BY
J J TAM

Introduction Go Language
Go: Hello World Application
Go language: Build executable
Go: Primitive Data Types
Go language: Print value and type of a variable
Go language: Initialize multiple variables in one line
Go Language: Constants
Go language: iota identifier
Go Language: Type Conversion
Go language: Type Inference
Go language: strings
Go language: concatenate strings
Go language: multi line strings
Go Language: Arrays
Go Language: Arrays are passed by value
Go language: Slices
Go language: Slices are passed by reference
Go language: Iterate over a slice using range keyword
Go language: Get slice from a slice
Go language: Append an element to a slice
Go Language: Map data structure
Go language: Define map using literal notation
Go language: len : Get number of items in a map
Go Language: Map: Check whether an item exists in map or not
Go language: Delete item from the map
Go language: Print all the keys of map
Go language: Print key, value from map

Go language: if statement
Go language: if-else statement
Go language: if statement: combine initialisation and condition evaluation in same line
Go Language: switch statement
Go language: switch without an expression
Go Language: for loop
Go Language: Use for loop as while loop
Go language: Print elements of array
Go language: Iterate over a map
Go language: Read data from console or user
Go Language: Create a function
Go Language: Pass by Reference
Go language: Variadic Functions
Go language: Return a value from function
Go language: Anonymous functions
Go Language: Functions continued
Go Language: return Error from a function
Go Language: Structs
Go Language: Construct an object to a structure using &
Go language : struct literal notation
Go language: constructor functions
Go Language: Adding Methods to a struct
Go Language: Interfaces
Go Language: Importing multiple packages
Go language: defer the function execution
Go language: Pointers
Go language: Structure pointers
Go Language: Read content type from a url

Go language: Exit the program with given status code
Go Language: Goroutines
Go language: Convert string to byte array
Go language: Generate random uuid or string
Go language: Write a program to print lowercase alphabets
Go language: Sleep goroutine for some time
Go language: Repeat a string n times
Go Language: Concatenate strings
Go Language: Multi line strings
Go Language: Convert byte array to string
For-each loop in Go language
Go language: Convert integer to string
Go language: Find type of object
Go language: Read input from console
Go language: Print structure with field names
LEARN PYTHON QUICKLY
Install python on MAC OS
Python: Hello World program
Python interactive command line
Python: Operators
Arithmetic Operators in python
Relational Operators in python
Assignment operators in python
Logical Operators in python
Bitwise Operators in python
Membership operators in python
Identity operators in python
Python: Short circuit operators
Strings in python

Python: if condition
Python: while statement
Python: for statement
Python: break statement
Python: continue statement
Python: functions
Python: functions: return statement
Python: functions: Default Argument Values
Python: functions: Keyword arguments
Python lists
Python: list extend: append elements of given list
Python: insert: Insert an element at given position
Python: list remove: Remove an element
Python: list: pop: remove last element in the list
Python: list: clear all elements in the list
Python: list: index: Get index of element
Python: list: sort elements of list
Python: count: Count number of times element appears
Python: list: Reverse elements of list
Python:list: del: delete elements
Python: Looping over lists
Python: tuples
Python: Sets
Python: Find intersection and union of lists
Python: Dictionaries
Python modules
Python: Executable statements in module
Python: import functions from module directly
Python: import module in other module

Python command line arguments
Python: File handling
Python: Reading files
Python: Write data to file
Python: classes and objects
Python: class: built in class attributes
Python: __init__ method
Python: Overloading __init__ method
Python: Class Vs Instance variables
Python: Inheritance
Python: Multiple inheritance
Python: Exceptions
Python: Handling Exceptions
Python: User defined exceptions
Python global keyword
Python: Get module name
Python: Get type of variable
Python: Generate random numbers

LEARN GOLANG QUICKLY

CODING FOR BEGINNERS WITH HANDS ON PROJECTS

BY

J J TAM

Introduction Go Language

Go is an open source language to build reliable software in efficient way.

a. Go is a compiled language.
b. Go has clean and clear syntax
c. Go has in built language support for concurrency
d. Go is statically typed language
e. Functions are first class citizens in Go
f. Go initialize default values for uninitialized variables. For example, for the string default value is empty string.
g. Go has good features that makes the development fast
h. Go has only few keywords to remember
. Most of the computers now a days has multiple cores, but not all languages have efficient ways to utilize these multi cores. But Go has very good support to utilize multi core system in efficient way.
. Since Go has very good built-in support for concurrency features, you no need to depend on any threading libraries to develop concurrent applications.
k. Go has in-built garbage collector, so you no need to take the overhead of managing application memory.

j. In Go, complex types are composed of smaller types. Go encourages composition.

Install and setup Go

Download latest version of Go from below location.

https://golang.org/

Extract the downloaded zip file, you can see below content structure.

```
$ ls
AUTHORS  CONTRIBUTORS PATENTS  VERSION  bin  favicon.ico misc
robots.txt test
CONTRIBUTING.md LICENSE  README.md api  doc  lib  pkg  src
```

Add bin directory path to your system path.

Open terminal or command prompt and execute the command 'go', you can see below output in console.

```
$ go
Go is a tool for managing Go source code.

Usage:

go <command> [arguments]

The commands are:

bug         start a bug report
build       compile packages and dependencies
clean       remove object files and cached files
doc         show documentation for package or symbol
env         print Go environment information
fix         update packages to use new APIs
fmt         gofmt (reformat) package sources
```

```
generate    generate Go files by processing source
get         download and install packages and dependencies
install     compile and install packages and dependencies
list        list packages or modules
mod         module maintenance
run         compile and run Go program
test        test packages
tool        run specified go tool
version     print Go version
vet         report likely mistakes in packages

Use "go help <command>" for more information about a command.

Additional help topics:

buildmode   build modes
c           calling between Go and C
cache       build and test caching
environment environment variables
filetype    file types
go.mod      the go.mod file
gopath      GOPATH environment variable
gopath-get  legacy GOPATH go get
goproxy     module proxy protocol
importpath  import path syntax
modules     modules, module versions, and more
module-get  module-aware go get
packages    package lists and patterns
testflag    testing flags
testfunc    testing functions

Use "go help <topic>" for more information about that topic.
```

Note

If you do not want to install Go in your system, you can play around Go at ‘https://play.golang.org/’.

Go: Hello World Application

Open any text editor and create HelloWorld.go file with below content.

HelloWorld.go

```
package main

func main() {
 println("Hello, World")
}
```

Execute the command ‘go run HelloWorld.go’.

```
$ go run HelloWorld.go
Hello, World
```

package main
It is used by the Go compiler to determine application entry point.

func main()
Program execution starts from here. ‘func’ keyword is used to define a function. ‘main’ function do not take any arguments.

println("Hello, World")
‘println’ is a built in function in Go, that is used to print given message to console.

Note

ı. Unlike C, C++ and Java, you no need to end a statement by a semi colon.
). String in Go, are placed in between double quotes
:. Strings in Go are Unicode.

Go language: Build executable

Use the command ‘go build goFilePath’, to build the executable from go program.

App.go

```
package main

import "fmt"

func main() {
fmt.Println("Hello World")
}
```

```
$ go build App.go
$
$ ls
App     App.go
```

As you see 'App' file is created after building. If you run build command in windows, it generates App.exe file.

Run the file 'App' to see the output.

```
$ ./App
Hello World
```

Go: Primitive Data Types

Below table summarizes the data types provided by Go Language.

Integer Data Types

Data Type	Description	Minimum Value	Maximum Value
uint8	Unsigned 8-bit integers	0	255
uint16	Unsigned 16-bit integers	0	65535
uint32	Unsigned 32-bit integers	0	4294967295
uint64	Unsigned 64-bit integers	0	18446744073709551615
int8	Signed 8-bit integers	-128	127
int16	Signed 16-bit integers	-32768	32767
int32	Signed 32-bit integers	-2147483648	2147483647
int64	Signed 64-bit integers	-9223372036854775808	9223372036854775807

Below table summarizes floating point numbers.

Data Type	Description
float32	IEEE-754 32-bit floating-point numbers
float64	IEEE-754 64-bit floating-point numbers

Below table summarizes the complex numbers.

Data Type	**Description**
complex64	Complex numbers with float32 real and imaginary parts
complex128	Complex numbers with float64 real and imaginary parts

Apart from the above types, Go language support below types that are implementation specific.

ı. Byte
ɔ. rune (same as int32)
:. uint
ɪ. int
ɛ. uintptr

Syntax to create variable
var variableName dataType
var variableName dataType = value
variableName := value

HelloWorld.go
```
package main

import "fmt"

func main() {
 var a int = 10
 var b uint8 = 11
 var c uint16 = 12
 var d uint32 = 13
 var e uint64 = 14
 var f int8 = 15
 var g int16 = 16
 var h int32 = 17
 var i int64 = 18

 var k float32 = 2
 var l float64 = 2

complex1 := complex(10, 13)

fmt.Println("a : ", a);
fmt.Println("b : ", b);
fmt.Println("c : ", c);
fmt.Println("d : ", d);
fmt.Println("e : ", e);
fmt.Println("f : ", f);
fmt.Println("g : ", g);
fmt.Println("h : ", h);
fmt.Println("i : ", i);

fmt.Println("k : ", k);
fmt.Println("l : ", l);

fmt.Println("complex1 : ", complex1);

}
```

Output

```
a :  10
b :  11
c :  12
d :  13
e :  14
f :  15
g :  16
h :  17
i :  18
k :  2
l :  2
complex1 :  (10+13i)
```

How to access real and imaginary numbers from complex numbers?

You can use ‘real’ and ‘img’ methods to access the real and complex parts of a number.

```
myComplex := complex(10, 13)
real(myComplex)
imag(myComplex)
```

HelloWorld.go

```
package main

import "fmt"

func main() {

complex1 := complex(10, 13)
 var realPart = real(complex1)
 var imgPart = imag(complex1)

fmt.Println("complex1 : ", complex1);
fmt.Println("realPart : ", realPart);
fmt.Println("imgPart : ", imgPart);

}
```

Output

```
complex1 :  (10+13i)
realPart :  10
imgPart :  13
```

Go language: Print value and type of a variable

%v is used to print the value of a variable
%T is used to print the type of a variable.

App.go

```go
package main

import "fmt"

func main() {
 var x int = 10

fmt.Printf("i : %v, type : %T\n", x, x)
}
```

Output

```
i : 10, type : int
```

Go language: Initialize multiple variables in one line

If variables are of same data type, you can initialize them in one line like below.

```
var i, j, k int = 1, 2, 3
```

App.go

```go
package main

import (
 "fmt"
)

func main() {
 var i, j, k int = 1, 2, 3

fmt.Println("i : ", i, ", j : ", j, ", k : ", k)

}
```

Output

```
i :  1 , j :  2 , k :  3
```

Go Language: Constants

‘const’ keyword is used to define constants.

Example

```
const (
	name = "JJTam"
	id = 123
	pin = "523169"
)
```

Above snippet create 3 constants. ‘name’ and ‘pin’ are of type String and ‘id’ is of type integer.

HelloWorld.go

```
package main

import "fmt"

const (
name = "JJTam"
id = 123
pin = "523169"
)

func main() {

fmt.Println("name : ", name)
fmt.Println("id : ", id)
fmt.Println("pin : ", pin)

}
```

Output

```
name :  JJTam
id :  123
```

```
pin :  523169
```

Just like how you define multiple constnats in ‘const’ block, you can define multiple variable in ‘var’ block.

HelloWorld.go

```go
package main

import "fmt"

var (
name = "JJTam"
id = 123
pin = "523169"
)

func main() {

fmt.Println("name : ", name)
fmt.Println("id : ", id)
fmt.Println("pin : ", pin)

}
```

Output

```
name :  JJTam
id :  123
pin :  523169
```

Go language: iota identifier

‘iota’ identifier is used in const declarations to simplify definitions of incrementing numbers.

Example

```
const (
        first = iota
        second
        third
        fourth
)
```

‘iota’ starts with number 0 and assign subsequent constants by incrementing the variable value by 1.

HelloWorld.go

```
package main

import "fmt"

const (
first = iota
second
third
fourth
)

func main() {

fmt.Println("first : ", first)
fmt.Println("second : ", second)
fmt.Println("third : ", third)
fmt.Println("fourth : ", fourth)
}
```

Output

```
first :  0
second :  1
third :  2
fourth :  3
```

Each ‘const’ block has its own iota.

```
const (
	block1First = iota
	block1Second
)

const (
	block2First = iota
	block2Second
	block2Third
)
```

HelloWorld.go

```go
package main

import "fmt"

const (
block1First = iota
block1Second
)

const (
block2First = iota
block2Second
block2Third
)
```

```
func main() {

fmt.Println("block1First : ", block1First)
fmt.Println("block1Second : ", block1Second)
fmt.Println("block2First : ", block2First)
fmt.Println("block2Second : ", block2Second)
fmt.Println("block2Third : ", block2Third)

}
```

Output

```
block1First :  0
block1Second :  1
block2First :  0
block2Second :  1
block2Third :  2
```

Using iota in constant expressions

We can use iota in constant expressions.

```
const (
	first = 1 << iota
	second
	third
	fourth
	five
	six
)
```

Value of
first is 1 << 0
second is 1 << 1
third is 1 << 2
fourth is 1 << 3 etc,

HelloWorld.go

```
package main

import "fmt"

var a int = 10

const (
first = 1 << iota
second
third
fourth
five
six
)

func main() {

fmt.Println("first : ", first)
fmt.Println("second : ", second)
fmt.Println("third : ", third)
fmt.Println("fourth : ", fourth)
fmt.Println("five : ", five)
fmt.Println("six : ", six)

}
```

Output

```
first :  1
second :  2
third :  4
fourth :  8
five :  16
six :  32
```

Go Language: Type Conversion

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

Example

```
var i int
var j float32 = 2.34
i = int(j)
```

App.go

```go
package main

import "fmt"

func main() {
 var i int
 var j float32 = 2.34
i = int(j)

 var k float32
 var l int = 2
k = float32(l)

fmt.Println("i = ", i)
fmt.Println("j = ", j)

fmt.Println("k = ", k)
fmt.Println("l = ", l)

}
```

Output

```
i =  2
j =  2.34
k =  2
l =  2
```

Go language: Type Inference

When you initialize a variable without specifying type, then the type of the variable is inferred from the value on the right hand side.

App.go

```
package main

import "fmt"

func main() {
i := 10
j := 10.23
k := true
l := "Hello World"

fmt.Printf("i is of type %T\n", i)
fmt.Printf("j is of type %T\n", j)
fmt.Printf("k is of type %T\n", k)
fmt.Printf("l is of type %T\n", l)
}
```

Output

```
i is of type int
j is of type float64
k is of type bool
l is of type string
```

Go language: strings

Strings in go are defined in double quotes.

Example

```
var msg string = "Hello World"
```

App.go

```go
package main

import "fmt"

func main() {

 var msg string = "Hello World"

fmt.Printf("msg : %v, type : %T\n", msg, msg)
}
```

Output

```
msg : Hello World, type : string
```

Get number of bytes in a string

Use built-in ‘len’ function to get the number of bytes in a string.

App.go

```go
package main

import "fmt"

func main() {

 var msg string = "Hello World"

fmt.Printf("msg : %v, type : %T\n", msg, msg)
fmt.Printf("Length of msg is %v\n", len(msg))
}
```

Output

```
msg : Hello World, type : string
Length of msg is 11
```

Access individual byte of a string

You can access individual byte of a string using index notation.

For example, msg[0] returns the first byte of the string.

App.go

```
package main

import "fmt"

func main() {

 var msg string = "Hello World"

 for i := 0; i < len(msg); i++ {
  fmt.Printf("msg[%v] = %v\n", i, msg[i])
}
}
```

Output

```
msg[0] = 72
msg[1] = 101
msg[2] = 108
msg[3] = 108
msg[4] = 111
msg[5] = 32
msg[6] = 87
msg[7] = 111
msg[8] = 114
```

```
msg[9] = 108
msg[10] = 100
```

String are immutable in Go
You can't change the string, once it is created.

App.go
```go
package main

import "fmt"

func main() {

 var msg string = "Hello World"

fmt.Printf("msg : %v", msg)

msg[0] = 112
}
```

When you try to ran above program, you will endup in below error.

```
# command-line-arguments
./App.go:11:9: cannot assign to msg[0]
```

Go language: concatenate strings

You can use + operator to concatenate two strings.

Example

```
var msg1 string = "Hello"
var msg2 string = "How Are You!!!!"
var space string = " "

var result string = msg1 + space + msg2
```

App.go

```go
package main

import "fmt"

func main() {

 var msg1 string = "Hello"
 var msg2 string = "How Are You!!!!"
 var space string = " "

 var result string = msg1 + space + msg2

fmt.Printf("msg1 : %v\n", msg1)
fmt.Printf("msg2 : %v\n", msg2)
fmt.Printf("space : %v\n", space)
fmt.Printf("result : %v\n", result)

}
```

Output

```
msg1 : Hello
msg2 : How Are You!!!!
space :
result : Hello How Are You!!!!
```

Go language: multi line strings

`(backtick) sign is used to construct multi-line strings.

App.go

```
package main

import "fmt"

func main() {

 var msg string = `Hello,
How are you
I am fine, thank you
........`

fmt.Println(msg)

}
```

Output

```
Hello,
	How are you
	I am fine, thank you
	........
```

Go Language: Arrays

Array is a collection that stores elements of same type.

Syntax
arrayame := [size]dataType{}

Example
countries := [5]string{}

Above statement defines an array of strings of size 5. ‘countries’ can able to store 5 strings.

HelloWorld.go
```
package main

import "fmt"

func main() {
countries := [5]string{}

countries[0] = "India"
countries[1] = "Bangladesh"

fmt.Println(countries)
}
```

Output
```
[India Bangladesh   ]
```

One problem with mentioning size to the array is, we can’t insert elements > size of the array.

HelloWorld.go

```
package main

import "fmt"

func main() {
countries := [5]string{}

countries[0] = "India"
countries[1] = "Bangladesh"
countries[2] = "Canada"
countries[3] = "Austria"
countries[4] = "Germany"

countries[5] = "Sri lanka"

fmt.Println(countries)
}
```

As you see above program, I mentioned size of array as 5, but I am trying to insert 6th element into the array.

When you run HelloWorld.go program, you will end up in below error.

```
$ go run HelloWorld.go
# command-line-arguments
./HelloWorld.go:14:11: invalid array index 5 (out of bounds for 5-element
array)
```

Length of the array
You can use ‘len’ function to get the length of the array.

HelloWorld.go

```
package main

import "fmt"

func main() {
```

```
countries := [5]string{}

countries[0] = "India"
countries[1] = "Bangladesh"

fmt.Println("Array length : ", len(countries))
}
```

Output

```
Array length :  5
```

Note

Once array is defined, its length is fixed, it can't be resized.

Go Language: Arrays are passed by value

Unlike other languages, Array are passed by value in Go.

For example, if you call updateCountries method by passing an array as argument, the changes done for the array ‘countries’ inside the method ‘updateCountries’ is not reflected outside.

```
func updateCountries(countries [5]string) {
    countries[0] = "Dummy"
    countries[1] = "Dummy"
    countries[2] = "Dummy"
    countries[3] = "Dummy"
    countries[4] = "Dummy"
}
```

App.go

```
package main

import "fmt"

func main() {
    countries := [5]string{}

    countries[0] = "India"
    countries[1] = "Bangladesh"
    countries[2] = "Canada"
    countries[3] = "Austria"
    countries[4] = "Germany"

    fmt.Println(countries)

    fmt.Println("\nUpdating countries with Dummy data")
    updateCountries(countries)

    fmt.Println()
    fmt.Println(countries)
}

func updateCountries(countries [5]string) {
    countries[0] = "Dummy"
    countries[1] = "Dummy"
    countries[2] = "Dummy"
    countries[3] = "Dummy"
    countries[4] = "Dummy"
}
```

Output

```
[India Bangladesh Canada Austria Germany]

Updating countries with Dummy data

[India Bangladesh Canada Austria Germany]
```

Go language: Slices

Arrays are fixed in size. If you create an array of size 6, then it can able to store exactly 6 elements. You can’t store more than 6 elements in the array, but slices can be resized on demand.

Slices are built on top of arrays. If you are changing any value of the slice, then it is going to change the value of underlying array and vice versa.

How to create a slice?
You can create a slice using built-in make function or indexing notation. ‘make’ function takes three arguments type, length and capacity.

Syntax
make([]Type, length, capacity)
make([]Type, length)
[]Type{}
[]Type{value1, value2, ..., valueN}

type: Specified the type of elements in the slice
length: tells you how many items are in the array.
capacity: Tells you the capacity of the underlying array. This the length of the hidden array.

App.go

```
package main

import "fmt"

func main() {
    data := make([]int, 5, 20)

    fmt.Println("Length : ", len(data))
    fmt.Println("Capacity : ", cap(data))
    fmt.Println("Data : ", data)
}
```

Output

```
Length :  5
Capacity :  20
[0 0 0 0 0]
```

Slices are passed by reference in Go.

App.go

```
package main

import "fmt"

func main() {
    countries := []string{}

    countries = append(countries, "India")
    countries = append(countries, "Bangladesh")
    countries = append(countries, "Canada")
    countries = append(countries, "Austria")
    countries = append(countries, "Germany")

    fmt.Println(countries)

    fmt.Println("\nUpdating countries with Dummy data")

    updateCountries(countries)

    fmt.Println()
    fmt.Println(countries)
}

func updateCountries(countries []string) {
    countries[0] = "Dummy"
    countries[1] = "Dummy"
    countries[2] = "Dummy"
    countries[3] = "Dummy"
    countries[4] = "Dummy"
}
```

Output

```
[India Bangladesh Canada Austria Germany]

Updating countries with Dummy data

[Dummy Dummy Dummy Dummy Dummy]
```

Go language: Slices are passed by reference

In Go language, Arrays are passed by value, whereas slices are passed by reference.

```
func updateCountries(countries []string) {
    countries[0] = "Dummy"
    countries[1] = "Dummy"
    countries[2] = "Dummy"
    countries[3] = "Dummy"
    countries[4] = "Dummy"
}
```

For example, if you call updateCountries method by passing a slice as an argument, the changes done for the slice ‘countries’ inside the method ‘updateCountries’ will be reflected outside.

App.go

```
package main

import "fmt"

func main() {
    countries := []string{}

    countries = append(countries, "India")
    countries = append(countries, "Bangladesh")
    countries = append(countries, "Canada")
    countries = append(countries, "Austria")
    countries = append(countries, "Germany")

    fmt.Println(countries)

    fmt.Println("\nUpdating countries with Dummy data")
    updateCountries(countries)

    fmt.Println()
    fmt.Println(countries)
}

func updateCountries(countries []string) {
    countries[0] = "Dummy"
    countries[1] = "Dummy"
    countries[2] = "Dummy"
    countries[3] = "Dummy"
    countries[4] = "Dummy"
}
```

Output

```
[India Bangladesh Canada Austria Germany]

Updating countries with Dummy data

[Dummy Dummy Dummy Dummy Dummy]
```

Go language: Iterate over a slice using range keyword

Example 1: Get the value of slice from index

```
for i := range vowels {
	fmt.Println(vowels[i])
}
```

Example 2: Get the index and value of slice

```
for i, v := range vowels {
	fmt.Printf("index %v, value %v\n", i, v)
}
```

App.go

```
package main

import "fmt"

func main() {

 var vowels = []string{"a", "e", "i", "o", "u"}

 for i := range vowels {
  fmt.Println(vowels[i])
 }

 for i, v := range vowels {
  fmt.Printf("index %v, value %v\n", i, v)
 }

}
```

Output

```
a
e
```

```
i
o
u
index 0, value a
index 1, value e
index 2, value i
index 3, value o
index 4, value u
```

Go language: Get slice from a slice

You can create a slice from a slice using the notation '[startPositon:endPosition]'.

Example

```
subSlice := mySlice[startPosition:endPositon]
```

Above statement creates a slice from ‘mySlice’. Elements start from startPosition (inclusive), and end at endPoistion (exclusive).

App.go

```go
package main

import (
 "fmt"
)

func main() {
slice := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

subSlice := slice[3:6]

fmt.Println("slice : ", slice)
fmt.Println("subSlice : ", subSlice)
}
```

Output

```
slice :  [0 1 2 3 4 5 6 7 8 9]
subSlice :  [3 4 5]
```

Go language: Append an element to a slice

‘append’ method is used to append an element to a slice.

Example

```
var vowels = []string{"a", "e", "i"}
vowels = append(vowels, "o")
```

App.go

```
package main

import "fmt"

func main() {

 var vowels = []string{"a", "e", "i"}

vowels = append(vowels, "o")
vowels = append(vowels, "u")

fmt.Println(vowels)

}
```

Output

```
[a e i o u]
```

Go Language: Map data structure

‘map’ is a data structure used to store key-value pairs.

Syntax

mapName := make(map[keyDataType]valueDataType)

Example

employees := make(map[int]string)

HelloWorld.go

```
package main

import "fmt"

func main() {
employees := make(map[int]string)

fmt.Println("Empty Map : ", employees)

employees[1] = "JJTam"
employees[12] = "Sibi"

fmt.Println("Map with two elements : ", employees)
}
```

Output

```
Empty Map :  map[]
Map with two elements :  map[1:JJTam 12:Sibi]
```

You can access an entry in the map using the key.

For example, ‘employees[12]’ access the entry in map with key 12.

HelloWorld.go

```
package main

import "fmt"

func main() {
employees := make(map[int]string)

fmt.Println("Empty Map : ", employees)

employees[1] = "JJTam"
employees[12] = "Sibi"

fmt.Println("Value of key 12 is ", employees[12])
fmt.Println("Value of key 123 is ", employees[123])
}
```

Output

```
Empty Map :  map[]
Value of key 12 is  Sibi
Value of key 123 is
```

Since there is no entry with key 123, employees[123] returns an empty string.

Go language: Define map using literal notation

You can define a map in literal notation also.

Syntax

```
mapName := map[dataType1]dataType2{
	key1 : value1,
	key2 : value2,
	.....
	.....
	.....
	keyN : valueN
}
```

App.go

```go
package main

import "fmt"

func main() {

employees := map[int]string{
  1: "JJTam",
  2: "Ram", //Must have trailing comma
}

fmt.Println("Map with two elements : ", employees)
}
```

Output

```
Map with two elements :  map[1:JJTam 2:Ram]
```

Go language: len : Get number of items in a map

‘len’ function return number of items in a map.

App.go

```
package main

import "fmt"

func main() {

employees := map[int]string{
  1: "JJTam",
  2: "Ram", //Must have trailing comma
}

fmt.Println("Map with two elements : ", employees)
fmt.Println("Total number of Items : ", len(employees))
}
```

Output

```
Map with two elements :  map[1:JJTam 2:Ram]
Total number of Items :  2
```

Go Language: Map: Check whether an item exists in map or not

By using double value context, we can check whether an item exists in the map or not.

```
value1, ok := employees[1]
if !ok {
	fmt.Println("Value not found for key 1")
} else {
	fmt.Println("Value for key 1 : ", value1)
}
```

ok value will be assigned to true, if the key is inside the map, else false. value1 is assigned with the value of key 1, if key 1 exists in the map else assigned with zero value.

App.go

```go
package main

import "fmt"

func main() {

employees := map[int]string{
  1: "JJTam",
  2: "Ram", //Must have trailing comma
}

value1, ok := employees[1]
 if !ok {
  fmt.Println("Value not found for key 1")
} else {
  fmt.Println("Value for key 1 : ", value1)
}
value2, ok := employees[10]
 if !ok {
  fmt.Println("Value not found for key 10")
} else {
  fmt.Println("Value for key 10 : ", value2)
}
}
```

Output

```
Value for key 1 :  JJTam
Value not found for key 10
```

Go language: Delete item from the map

‘delete’ method is used to delete an item in the map.

Syntax
delete(mapName, key)

App.go
```
package main

import "fmt"

func main() {

employees := map[int]string{
  1: "JJTam",
  2: "Ram",
  3: "Chamu", //Must have trailing comma
}

fmt.Println("employees : ", employees)

 delete(employees, 2)

fmt.Println("\nDeleting the item with key 2")

fmt.Println("\nemployees : ", employees)
}
```

Output
```
employees :  map[1:JJTam 2:Ram 3:Chamu]

Deleting the item with key 2
```

```
employees :  map[3:Chamu 1:JJTam]
```

Go language: Print all the keys of map

You can print all the keys of a map using for loop.

Example

```
for key := range employees {
        fmt.Println(key)
}
```

App.go

```
package main

import "fmt"

func main() {

employees := map[int]string{
  1: "JJTam",
  2: "Ram",
  3: "Chamu", //Must have trailing comma
}

 for key := range employees {
  fmt.Println(key)
}
}
```

Output

```
1
2
3
```

Go language: Print key, value from map

You can print key, values from a map using for loop.

Example

```
for key, value := range employees {
        fmt.Println(key, " : ", value)
}
```

App.java

```
package main

import "fmt"

func main() {

employees := map[int]string{
  1: "JJTam",
  2: "Ram",
  3: "Chamu", //Must have trailing comma
}

 for key, value := range employees {
  fmt.Println(key, " : ", value)
}
}
```

Output

```
1  :  JJTam
2  :  Ram
3  :  Chamu
```

Go language: if statement

Syntax

```
if condition {

}
```

If the condition evaluates to true, then the block of statements followed by if block are executed.

HelloWorld.go

```go
package main

import "fmt"

func main() {

a := 10

 if a == 10 {
  fmt.Println("Value of a is 10")
}

 if a == 11 {
  fmt.Println("Value of a is 11")
}
}
```

Output

```
Value of a is 10
```

Go language: if-else statement

Syntax

```
if condition {

} else {

}
```

If the condition evaluates to true, then the block of statements followed by if block are executed, else the statements followed by else block are executed.

HelloWorld.go

```
package main

import "fmt"

func main() {

a := 10

 if a == 10 {
  fmt.Println("Value of a is 10")
}else {
  fmt.Println("Value of a is 11")
}
}
```

Output

```
Value of a is 10
```

Go language: if statement: combine initialisation and condition evaluation in same line

Syntax

```
if initialization; condition {

}
```

Example

```
a := 10

if a == 10 {
	fmt.Println("Value of a is 10")
}
```

Above statements can be written like below.

```
if a := 10; a == 10 {
	fmt.Println("Value of a is 10")
}
```

HelloWorld.go

```
package main

import "fmt"

func main() {

 if a := 10; a == 10 {
  fmt.Println("Value of a is 10")
 }else {
  fmt.Println("Value of a is 11")
}
}
```

Output

```
Value of a is 10
```

Go Language: switch statement

Syntax

```
switch expression {

        case caseStatement1 :
                statements

        case caseStatement2 :
                statements


        …..
           …..
           default :
                   statements
                   break
}
```

HelloWorld.go

```
package main

import "fmt"

func main() {

a := 10

 switch a {
  case 10 :
   fmt.Println("Value of a is 10")

  case 20 :
   fmt.Println("Value of a is 20")
}
}
```

Output

```
Value of a is 10
```

If the value in switch statement is not matched to anything, then default block gets executed.

HelloWorld.go

```
package main

import "fmt"

func main() {

a := 100

 switch a {
  case 10 :
   fmt.Println("Value of a is 10")

  case 20 :
   fmt.Println("Value of a is 20")

  default :
   fmt.Println("Default block is executed")
}
}
```

Output

```
Default block is executed
```

Just like ‘if’ statement in Go, you can add initialization in switch statement itself.

```
switch a :=10; a {
}
```

HelloWorld.go

```go
package main

import "fmt"

func main() {

 switch a :=10; a {
  case 10 :
   fmt.Println("Value of a is 10")

  case 20 :
   fmt.Println("Value of a is 20")

  default :
   fmt.Println("Default block is executed")
}
}
```

Output

```
Value of a is 10
```

You can even omit the expression after switch statement.

HelloWorld.go

```go
package main

import "fmt"

func main() {

a :=10

 switch {
  case a == 10 :
   fmt.Println("Value of a is 10")

  case a > 5 :
   fmt.Println("Value of a is > 5")

  case a < 5 :
   fmt.Println("Value of a is < 5")

  default :
   fmt.Println("Default block is executed")
}
}
```

Output

```
Value of a is 10
```

As you see the output, even though both the cases, a == 10, a > 5 are evaluating to true, only first matching statement code gets executed.

Go language: switch without an expression

Syntax

```
switch {

	case caseStatement1 :
		statements

	case caseStatement2 :
		statements


	…..
	   …..
	   default :
		   statements
		   break
}
```

HelloWorld.go

```go
package main

import "fmt"

func main() {

a := 10

 switch {
 case a > 10:
  fmt.Println("Value of a is  > 10")

 case a <= 10:
  fmt.Println("Value of a is <= 10")
}
}
```

Output

```
Value of a is <= 10
```

Go Language: for loop

Go has only one looping construct : ‘for’ loop.

Syntax

```
for initialize: condition: updation {
	statements to evaluate
}
```

Example

```
for i := 1; i < 10; i++ {
	fmt.Println("Value of i is : " , i);
}
```

HelloWorld.go

```go
package main

import "fmt"

func main() {

 for i := 1; i < 10; i++ {
  fmt.Println("Value of i is : " , i);
}

}
```

Output

```
Value of i is :  1
Value of i is :  2
Value of i is :  3
Value of i is :  4
Value of i is :  5
Value of i is :  6
Value of i is :  7
Value of i is :  8
Value of i is :  9
```

You can omit initialize, condition and updation sections of for loop.

Below syntax is possible in Go.

```
for {

}
```

HelloWorld.go

```go
package main

import "fmt"

func main() {

i := 1

 for {
  fmt.Println("Value of i is : " , i)

  i++

  if i > 5 {
   break
  }
}

}
```

Output

```
Value of i is :  1
Value of i is :  2
Value of i is :  3
Value of i is :  4
Value of i is :  5
```

Go Language: Use for loop as while loop

Example

```
for i <= 10 {
        fmt.Println("i : ", i)
        i++
}
```

App.go

```
package main

import "fmt"

func main() {
i := 0

 for i <= 10 {
  fmt.Println("i : ", i)
  i++
}
}
```

Output

```
i :  0
i :  1
i :  2
i :  3
i :  4
i :  5
i :  6
i :  7
i :  8
i :  9
```

i : 10

Go language: Print elements of array

Below syntax print index and value of array elements.

```
for index, value := range countries {
        fmt.Println("index : ", index, ", Value : ", value)
}
```

HelloWorld.go

```
package main

import "fmt"

func main() {

countries := [4]string{"India", "Bangladesh", "Sri Lanka", "Japan"}

 for index, value := range countries {
  fmt.Println("index : ", index, ", Value : ", value)
}

}
```

Output

```
index :  0 , Value :  India
index :  1 , Value :  Bangladesh
index :  2 , Value :  Sri Lanka
index :  3 , Value :  Japan
```

Go language: Iterate over a map

Below statements print key and values of map ‘employees’.

```
for key, value := range employees{
        fmt.Println("Key : ", key, ", Value : ", value);
}
```

HelloWorld.go

```
package main

import "fmt"

func main() {

employees := make(map[int]string)

employees[123] = "JJTam"
employees[23] = "Ram"
employees[67] = "Chamu"

 for key, value := range employees{
  fmt.Println("Key : ", key, ", Value : ", value);
}

}
```

Output

```
Key :  123 , Value :  JJTam
Key :  23 , Value :  Ram
Key :  67 , Value :  Chamu
```

Go language: Read data from console or user

‘fmt.Scanln’ function is used to read data from user.

Syntax
fmt.Scanln(&variableName)

HelloWorld.go
```
package main

import "fmt"

func main() {

 var name string
 var id int

fmt.Println("Enter your name")
fmt.Scanln(&name)

fmt.Println("Enter your id")
fmt.Scanln(&id)

fmt.Println("name : ", name, " id ", id)
}
```

Output
```
Enter your name
JJTam
Enter your id
123
name :  JJTam  id  123
```

Go Language: Create a function

Functions are created using func keyword.

A simple function without parameters looks like below.

```
func functionName(){

}
```

Example

```
func sayHello(){
	fmt.Println("Hello!!!!!")
}
```

You can call the function using the staement ‘sayHello()’

HelloWorld.go

```
package main

import "fmt"

func main() {
sayHello()
}

func sayHello(){
fmt.Println("Hello!!!!!")
}
```

Output

```
Hello!!!!!
```

Parameterized functions

A function can take zero or more parameters.

Syntax

```
func functionName(parameter1 dataType, parameter2 dataType ….parameterN dataType){

}
```

Example

```
func sayHello(name string){
	fmt.Println("Hello!!!!! Mr.", name)
}
```

You can call sayHello function like below.

```
sayHello("JJTam")

name := "Ram"
sayHello(name)
```

HelloWorld.go

```
package main

import "fmt"

func main() {

sayHello("JJTam")

name := "Ram"
sayHello(name)

}

func sayHello(name string){
fmt.Println("Hello!!!!! Mr.", name)
}
```

Output

```
Hello!!!!! Mr. JJTam
Hello!!!!! Mr. Ram
```

Go Language: Pass by Reference

HelloWorld.go

```
package main

import "fmt"

func main() {

name := "Ram"

changeMe(&name)

fmt.Println("Hello!!!!! Mr.", name)

}

func changeMe(name *string){
 *name = "JJTam"
}
```

Output

```
Hello!!!!! Mr. JJTam
```

Go language: Variadic Functions

A function that takes variable number of arguments is called variadic functions. You can declare a variadic function using three dots (…).

For example below function takes any number of strings as argument.

```
func sayHello(names ...string){

}
```

For example, below statements are valid

```
sayHello("Sibi")
sayHello("Mavra", "Sunil")
sayHello("Ram", "JJTam", "Chamu", "Sowmya")
```

HelloWorld.go

```
package main

import "fmt"

func main() {

sayHello("Sibi")
sayHello("Mavra", "Sunil")
sayHello("Ram", "JJTam", "Chamu", "Sowmya")

}

func sayHello(names ...string){
 for _, value := range names{
  fmt.Println("Hello, ", value)
}
}
```

Output

```
Hello,  Sibi
Hello,  Mavra
Hello,  Sunil
Hello,  Ram
Hello,  JJTam
Hello,  Chamu
Hello,  Sowmya
```

Note

ı. Variable number of arguments of a function must be defined at last.

Go language: Return a value from function

Functions in Go can be written zero or more values.

Syntax

```
func functionName(parameters) returnType{
        statements

        return value
}
```

You can return a value from a function using ‘return’ keyword.

```
func add(a int, b int) int{
        return a + b
}
```

HelloWorld.go

```
package main

import "fmt"

func main() {

result := add(10, 20)

fmt.Println("Sum of 10 and 20 is : ", result)

}

func add(a int, b int) int{
 return a + b
}
```

Output

```
Sum of 10 and 20 is :  30
```

Returning multiple values from a function

You can return multiple values from a function.

Syntax

```
func functionName(parameters) (returnType1, … returnType N){
	statements

	return value1, … valueN
}
```

Example

```
func arithmetic(a int, b int) (int, int, int, int){
	return a + b, a - b, a *b, a / b
}
```

HelloWorld.go

```go
package main

import "fmt"

func main() {

addition, subtraction, multiplication, division := arithmetic(10, 20)

fmt.Println("Sum of 10 and 20 is : ", addition)
fmt.Println("Subtraction of 10 and 20 is : ", subtraction)
fmt.Println("Multiplication of 10 and 20 is : ", multiplication)
fmt.Println("Division of 10 and 20 is : ", division)

}

func arithmetic(a int, b int) (int, int, int, int){
 return a + b, a - b, a *b, a / b
}
```

Output

```
Sum of 10 and 20 is :  30
Subtraction of 10 and 20 is :  -10
Multiplication of 10 and 20 is :  200
Division of 10 and 20 is :  0
```

Named parameters

Just add the names to the return types in function signature.

Example

```
func arithmetic(a int, b int) (sum int, sub int, mul int, div int){
        sum =  a + b
        sub = a - b
        mul = a *b
        div = a / b

        return
}
```

HelloWorld.go

```
package main

import "fmt"

func main() {

addition, subtraction, multiplication, division := arithmetic(10, 20)

fmt.Println("Sum of 10 and 20 is : ", addition)
fmt.Println("Subtraction of 10 and 20 is : ", subtraction)
fmt.Println("Multiplication of 10 and 20 is : ", multiplication)
fmt.Println("Division of 10 and 20 is : ", division)

}
```

```
func arithmetic(a int, b int) (sum int, sub int, mul int, div int){
sum =  a + b
sub = a - b
mul = a *b
div = a / b

 return
}
```

Output

```
Sum of 10 and 20 is :  30
Subtraction of 10 and 20 is :  -10
Multiplication of 10 and 20 is :  200
Division of 10 and 20 is :  0
```

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

Go language: Anonymous functions

Function without name is called anonymous function.

Example

```
add := func(a int, b int) int{
        return a + b
}
```

HelloWorld.go

```
package main

import "fmt"

func main() {

add := func(a int, b int) int{
  return a + b
}

result := add(10, 20)

fmt.Println("result : ", result)

}
```

Output

```
result :  30
```

Go Language: Functions continued

When two or more consecutive parameters to a function taking same type, you can omit the type from all except the last.

```
func add(x, y, z int) int {
        return x + y + z
}
```

App.go

```
package main

import (
 "fmt"
)

func main() {
result := add(10, 20, 30)

fmt.Println("Sum of 10, 20, 30 is ", result)

}

func add(x, y, z int) int {
 return x + y + z
}
```

Output

```
Sum of 10, 20, 30 is  60
```

Go Language: return Error from a function

Go functions return multiple values. Using this feature, most of the Go functions return error value as the last value returned by the function.

For example, below function calculate sum of two integers a and b and a nil, if the numbers are >= 0, else return 0 and an error message.

```
func add(a int, b int) (int, error) {
	if a < 0 || b < 0 {
		return 0, fmt.Errorf("sum pf negative numbers is not
supported")
	}

	return a + b, nil
}
```

‘nil’ is like null in other programming languages like Java.

App.go

```
package main

import "fmt"

func main() {
result1, err1 := add(10, 20)

 if err1 != nil {
  fmt.Println(err1)
} else {
  fmt.Println("Sum of 10 and 20 is : ", result1)
}
result2, err2 := add(0, -1)

 if err2 != nil {
  fmt.Println(err2)
} else {
  fmt.Println("Sum of 0 and -1 is : ", result2)
}
}

func add(a int, b int) (int, error) {
 if a < 0 || b < 0 {
  return 0, fmt.Errorf("sum pf negative numbers is not supported")
}

 return a + b, nil
}
```

Output

```
Sum of 10 and 20 is :  30
sum pf negative numbers is not supported
```

Go Language: Structs

‘type’ keyword is used to create a new custom type.

Example

```
type Employee struct{
	name string
	id int
}
```

Above statements create a structure of type Employee. Employee structure contains two fields (name is of type string, id is of type int).

How to create an object of type Employee?

```
emp1 := Employee{}
```

Above statement creates an object of type Employee.

How can you access the fields (name, id) of structre Employee?
You can access the fields of Employee using ‘.’ Operator.

```
emp1.name = "JJTam"
emp1.id = 1
```

HelloWorld.go

```
package main

import "fmt"

func main() {
emp1 := Employee{}

emp1.name = "JJTam"
emp1.id = 1
```

```go
fmt.Println("name : ", emp1.name)
fmt.Println("id : ", emp1.id)
}


type Employee struct{
name string
id int
}
```

Output

```
name :  JJTam
id :  1
```

Creating an object using literal notation

You can even create an object to a structure using literal notation like below.

```go
emp1 := Employee{"JJTam", 1}
```

HelloWorld.go

```go
package main

import "fmt"

func main() {
emp1 := Employee{"JJTam", 1}

fmt.Println("name : ", emp1.name)
fmt.Println("id : ", emp1.id)
}


type Employee struct{
name string
id int
}
```

One thing to notify here is that, this way of creating object, creates an object in local execution stack. But in most of the cases, creation of large object in heap is effective.

How can you create an object in heap?
You can create an object in heap using ‘new’ operator.

Syntax
objectName := new(StructureName)

Example
emp1 := new(Employee)

HelloWorld.go

```
package main

import "fmt"

func main() {
emp1 := new(Employee)

emp1.name = "JJTam"
emp1.id = 1

fmt.Println("name : ", emp1.name)
fmt.Println("id : ", emp1.id)
}

type Employee struct{
name string
id int
}
```

Go Language: Construct an object to a structure using &

You can even create an object to a structure using & operator.

Example

```
emp1 := &Employee{}
```

HelloWorld.go

```go
package main

import "fmt"

func main() {
emp1 := &Employee{}

emp1.name = "JJTam"
emp1.id = 1

fmt.Println("name : ", emp1.name)
fmt.Println("id : ", emp1.id)
}

type Employee struct{
name string
id int
}
```

Output

```
name :  JJTam
id :  1
```

You can use literal notation while constructing the object using &.

Example

```
emp1 := &Employee{"JJTam", 1}
```

HelloWorld.go

```
package main

import "fmt"

func main() {
emp1 := &Employee{"JJTam", 1}

fmt.Println("name : ", emp1.name)
fmt.Println("id : ", emp1.id)
}

type Employee struct{
name string
id int
}
```

Output

```
name :  JJTam

id :  1
```

Go language : struct literal notation

You can create an object to a structure using literal notation like below.

```
emp1 := Employee{"JJTam", 1}
```

HelloWorld.go

```
package main

import "fmt"

func main() {
emp1 := Employee{"JJTam", 1}

fmt.Println("name : ", emp1.name)
fmt.Println("id : ", emp1.id)
}

type Employee struct{
name string
id int
}
```

Output

```
name :  JJTam
id :  1
```

You can list subset of the fields using name of that field.

```
emp1 := Employee{id: 123}
```

App.java

```
package main

import "fmt"
```

```
func main() {
emp1 := Employee{id: 123}

fmt.Println("name : ", emp1.name)
fmt.Println("id : ", emp1.id)
}

type Employee struct {
name string
id   int
}
```

Output

```
name :

id :  123
```

Go language: constructor functions

Unlike other languages Java, Go do not have any constructor functions to initialize an object. But you can write custom functions to initialize the object.

Example

```
func getEmployee(name string, id int, hobbies []string) Employee{
	emp := Employee{}

	emp.name = name
	emp.id = id
	emp.hobbies = hobbies

	return emp
}
```

HelloWorld.go

```
package main

import "fmt"

func main() {
emp1 := getEmployee("JJTam", 1, []string{"Cricket", "Football"})

fmt.Println("name : ", emp1.name)
fmt.Println("id : ", emp1.id)
fmt.Println("hobbies : ", emp1.hobbies)
}

type Employee struct{
name string
id int

hobbies []string
}
```

```
func getEmployee(name string, id int, hobbies []string) Employee{
emp := Employee{}

emp.name = name
emp.id = id
emp.hobbies = hobbies

 return emp
}
```

Output

```
name :  JJTam
id :  1
hobbies :  [Cricket Football]
```

You can enhance the constructor function by adding some validation checks.

```
func getEmployee(name string, id int, hobbies []string) (*Employee,
error) {
	if name == "" {
		return nil, fmt.Errorf("Name can't be nil")
	}

	if id <= 0 {
		return nil, fmt.Errorf("id can't be zero")
	}

	emp := Employee{}

	emp.name = name
	emp.id = id
	emp.hobbies = hobbies

	return &emp, nil
}
```

HelloWorld.go

```
package main

import "fmt"

func main() {
emp1, err1 := getEmployee("JJTam", 1, []string{"Cricket", "Football"})

 if err1 != nil {
  fmt.Println("Error Occurred while creating employee")
  fmt.Println(err1)
} else {
  emp1.printEmployeeInfo()
}

emp2, err2 := getEmployee("JJTam", -10, []string{"Cricket", "Football"})

 if err2 != nil {
  fmt.Println("Error Occurred while creating employee")
  fmt.Println(err2)
} else {
  emp2.printEmployeeInfo()
}

}

type Employee struct {
name string
id   int

hobbies []string
}

func (emp *Employee) printEmployeeInfo() {
fmt.Println("name : ", emp.name, ", id : ", emp.id, ", hobbies : ", emp.hobbies)
}
```

```
func getEmployee(name string, id int, hobbies []string) (*Employee, error) {
 if name == "" {
  return nil, fmt.Errorf("Name can't be nil")
}

 if id <= 0 {
  return nil, fmt.Errorf("id can't be zero")
}

emp := Employee{}

emp.name = name
emp.id = id
emp.hobbies = hobbies

 return &emp, nil
}
```

Output

```
name :  JJTam , id :  1 , hobbies :  [Cricket Football]
Error Occurred while creating employee
id can't be zero
```

Go Language: Adding Methods to a struct

In my previous post, I explained how to create a struct and add fields to it. In this post, I am going to explain how can you add methods to a structure.

```
type MyType struct { }

func (m MyType) myFunc() int {
   //code
}
```

Above snippet add new function 'myFunc' to the structure MyType.

HelloWorld.go

```
package main

import "fmt"

func main() {
emp1 := new(Employee)
emp1.name = "JJTam"
emp1.id = 123

emp1.printEmployeeInfo()

}

type Employee struct {
name string
id   int
}

func (emp Employee) printEmployeeInfo() {
```

```
fmt.Println("name : ", emp.name, ", id : ", emp.id)
}
```

Output

```
name :  JJTam , id :  123
```

One advantage of adding methods to a structure like this is, There is clear separation between data and the methods.

Let’s see another example, where I am going to add a method ‘capitalizeName’, which capitalizes the employee name.

```
func (emp Employee) capitalizeName() {
	emp.name = strings.ToUpper(emp.name)
}
```

App.go

```
package main

import "fmt"
import "strings"

func main() {
emp1 := new(Employee)
emp1.name = "JJTam"
emp1.id = 123

emp1.printEmployeeInfo()
emp1.capitalizeName()
fmt.Println("After calling capitalizeName method")
emp1.printEmployeeInfo()

}

type Employee struct {
name string
id   int
```

```go
}

func (emp Employee) printEmployeeInfo() {
fmt.Println("name : ", emp.name, ", id : ", emp.id)
}

func (emp Employee) capitalizeName() {
emp.name = strings.ToUpper(emp.name)
}
```

Output

```
name :  JJTam , id :  123
After calling capitalizeName method
name :  JJTam , id :  123
```

As you see the output, name is not getting capitalized after calling ‘capitalizeName()’ method. The reason is that you are not using a pointer receiver. Change ‘capitalizeName’ method signature like below to make the things right.

```go
func (emp *Employee) capitalizeName() {
	emp.name = strings.ToUpper(emp.name)
}
```

App.go

```go
package main

import "fmt"
import "strings"

func main() {
emp1 := new(Employee)
emp1.name = "JJTam"
emp1.id = 123

emp1.printEmployeeInfo()
emp1.capitalizeName()
fmt.Println("After calling capitalizeName method")
emp1.printEmployeeInfo()

}

type Employee struct {
name string
id   int
}

func (emp Employee) printEmployeeInfo() {
fmt.Println("name : ", emp.name, ", id : ", emp.id)
}

func (emp *Employee) capitalizeName() {
emp.name = strings.ToUpper(emp.name)
}
```

Output

```
name :  JJTam , id :  123
After calling capitalizeName method
name :  JJTAM , id :  123
```

Go Language: Interfaces

Interfaces are collection of method signatures.

‘interface’ keyword is used to create interfaces.

Example

```
type Details interface {
	aboutMe()
	id() int
}
```

How can a type implement an interface?
You just need to implement all the methods in the interface (You even no need to explicitly specify that you are implementing an interface, you just write the implementation, this is called duck typing.).

For example, Employee type implements Details interface.

```
type Employee struct {
	name string
	id   int
}

func (emp *Employee) aboutMe() {
	fmt.Println("name : ", emp.name, ", id : ", emp.id)
}

func (emp *Employee) getId() int {
	return emp.id
}
```

If you define ‘printMyDetails’ method by adding Details object as an argument, you can pass any object of type ‘T’ that implements Details interface.

```
func printMyDetails(obj Details) {
        obj.aboutMe()
}
```

In Go, if your type implements all the methods of the interface, then your type can be stored in a value of interface type. For example, Details object can store Employee object.

```
emp1 := &Employee{"JJTam", 123}
printMyDetails(emp1)
```

App.go

```
package main

import (
 "fmt"
)

type Details interface {
aboutMe()
getId() int
}

type Employee struct {
name string
id   int
}

func (emp *Employee) aboutMe() {
fmt.Println("name : ", emp.name, ", id : ", emp.id)
}

func (emp *Employee) getId() int {
 return emp.id
}

func printMyDetails(obj Details) {
obj.aboutMe()
}

func main() {
emp1 := &Employee{"JJTam", 123}

printMyDetails(emp1)
}
```

Output

```
name :  JJTam , id :  123
```

Go Language: Importing multiple packages

You can import multiple package using import block.

```
import (
	"fmt"
	"math/rand"
)
```

HelloWorld.go

```go
package main

import (
 "fmt"
 "math/rand"
)

func main() {
fmt.Println("Random Number : ", rand.Intn(100000))
}
```

Sample Output

```
Random Number :  98081
```

You can even import packages like below.

```
import "fmt"
import "math"
```

Go language: defer the function execution

‘defer’ statement is used to defer the execution of a function until the surrounding function returns.

Syntax
defer function

App.go
```
package main

import "fmt"

func main() {

 defer fmt.Println(", Welcome to Go Language")

 defer fmt.Print("Hello ")
}
```

Output
```
Hello , Welcome to Go Language
```

When a function returns, deferred calls are executed in LIFO (Last-In-First-Out) order.

App.go
```
package main

import "fmt"

func main() {
```

```
fmt.Println("Starting Execution")

 for i := 0; i < 10; i++ {
  defer fmt.Println(i)
}
 defer fmt.Println("Finished Execution")
}
```

Output

```
Starting Execution
Finished Execution
9
8
7
6
5
4
3
2
1
0
```

Note

ı. defer statements are mainly used to clean the resources

ɔ. defer will be called, even if there is an error in the code.

Go language: Pointers

A pointer is used to hold the address of a variable.

Syntax

```
var pointerName *dataType
```

Example

```
var intPtr *int
var a int = 10
intPtr = &a
```

'intPtr' can able to hold the address of an integer variable. The & operator gives the address of a variable.

App.go

```
package main

import "fmt"

func main() {
 var intPtr *int

 var a int = 10

intPtr = &a

fmt.Println("Value of a is : ", a)

fmt.Println("\na is stored at address : ", intPtr)
fmt.Println("Value of a is : ", *intPtr)
}
```

Output

```
Value of a is :  10
```

```
a is stored at address :  0xc000014058
Value of a is :  10
```

Go language: Structure pointers

You can store the address of a structure using structure pointer.

Example

```
emp1 := employee{"JJTam", 1}
var empPtr *employee
empPtr = &emp1
```

How to access the variables of a structure using pointer?
Use '.' operator to acceess the variables of a structure.

Example

```
empPtr.name = "Ram"
empPtr.id = 12345
```

App.java

```
package main

import "fmt"

func main() {
emp1 := employee{"JJTam", 1}

fmt.Println("name : ", emp1.name)
fmt.Println("id : ", emp1.id)

 var empPtr *employee
empPtr = &emp1

empPtr.name = "Ram"
empPtr.id = 12345

fmt.Println("\nname : ", emp1.name)
fmt.Println("id : ", emp1.id)
}
```

```
type employee struct {
name string
id   int
}
```

Output

```
name :  JJTam
id :  1

name :  Ram
id :  12345
```

Go Language: Read content type from a url

App.go

```
package main

import (
 "fmt"
 "net/http"
)

func main() {
url := "http://self-learning-java-tutorial.blogspot.com"

contentType, err := contentType(url)

 if err != nil {
  fmt.Println(err)
  return
}

fmt.Println("Content Type : ", contentType)

}

func contentType(url string) (string, error) {
resp, err := http.Get(url)

 if err != nil {
  return "", err
}

 defer resp.Body.Close()

cType := resp.Header.Get("Content-Type")
 return cType, nil
}
```

Output

```
Content Type :  text/html; charset=UTF-8
```

Go language: Exit the program with given status code

'Exit(code int)' method of 'os' package used to exit the application with given status code.

Example

```
os.Exit(1)
```

App.java

```
package main

import (
 "fmt"
 "os"
)

func main() {

fmt.Println("Hello World")

os.Exit(1)

fmt.Println("I am not going to print")
}
```

Output

```
Hello World
exit status 1
```

Go Language: Goroutines

Goroutine is a lightweight thread maintained by GO runtime.

How can you convert a function to goroutine?
You can convert a function to Goroutine by prefixing it with the keyword 'go' while calling.

Example
go printUpperAlphabets()

In the above example, printUpperAlphabets() method is executed as goroutine, that means it will execute parallelly.

Let's see it with an example.

App.go
```
package main

import (
    "fmt"
)

func main() {
    printUpperAlphabets()
    printLowerAlphabets()
}

func printUpperAlphabets() {
    for alphabet := byte('A'); alphabet <= byte('Z'); alphabet++ {
        fmt.Print(string(alphabet), " ")
    }

    fmt.Println()
}
```

```
func printLowerAlphabets() {
    for alphabet := byte('a'); alphabet <= byte('z'); alphabet++ {
        fmt.Print(string(alphabet), " ")
    }

    fmt.Println()
}
```

As you see main method, I am calling two functions 'printUpperAlphabets' followed by 'printLowerAlphabets'. When I ran App.go application, I can see uppercase alphabets followed by lowercase alphabets printed to the console.

App.go

```
package main

import (
    "fmt"
)

func main() {
    printUpperAlphabets()
    printLowerAlphabets()
}

func printUpperAlphabets() {
    for alphabet := byte('A'); alphabet <= byte('Z'); alphabet++ {
        fmt.Print(string(alphabet), " ")
    }

    fmt.Println()
}

func printLowerAlphabets() {
    for alphabet := byte('a'); alphabet <= byte('z'); alphabet++ {
        fmt.Print(string(alphabet), " ")
    }

    fmt.Println()
}
```

Output

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Let’s convert the functions printUpperAlphabets, printLowerAlphabets as goroutines and re run App.go

Prefix method calls with go to run them as goroutines.

```
go printUpperAlphabets()
go printLowerAlphabets()
```

App.go

```go
package main

import (
	"fmt"
)

func main() {
	go printUpperAlphabets()
	go printLowerAlphabets()
}

func printUpperAlphabets() {
	for alphabet := byte('A'); alphabet <= byte('Z'); alphabet++ {
		fmt.Print(string(alphabet), " ")
	}

	fmt.Println()
}

func printLowerAlphabets() {
	for alphabet := byte('a'); alphabet <= byte('z'); alphabet++ {
		fmt.Print(string(alphabet), " ")
```

```
    }

    fmt.Println()
}
ab
```

When you ran App.go, you will not see anything. It is because, once the main function finishes its execution, it terminates the application. Since goroutines run parallelly, we need to give some time to the main() function to terminate.

Update App.go by adding below statement at the end of main function.
time.Sleep(2 * time.Second)

Sleep method makes sure that the main() pauses for 2 seconds.

App.go

```
package main

import (
    "fmt"
    "time"
)

func main() {
    go printUpperAlphabets()
    go printLowerAlphabets()

    time.Sleep(2 * time.Second)
}

func printUpperAlphabets() {
    for alphabet := byte('A'); alphabet <= byte('Z'); alphabet++ {
        fmt.Print(string(alphabet), " ")
    }

    fmt.Println()
}
```

```
func printLowerAlphabets() {
   for alphabet := byte('a'); alphabet <= byte('z'); alphabet++ {
      fmt.Print(string(alphabet), " ")
   }

   fmt.Println()
}
```

abc
Run App.go for 3 or 4 times, you can see different results every time.

Run 1

```
a b c d e f g h i A B C D E F G H I J K L M N O P Q R S T U V W X Y j
k l m n o p q r s t u v w Z
x y z
```

Run 2

```
A a b c d e f g h i j k l m B C D E F G H I J K L M N O P Q n o p q R S T
U V W X Y Z
r s t u v w x y z
```

Run 3

```
a b c d e f A g h i j k l m n o p B C D q r s t u v w x y z
E F G H I J K L M N O P Q R S T U V W X Y Z
```

Since goroutines run parallelly, you are seeing different results every time.

As I said, you can make any function as goroutine by prefixing it with go keyword, you can run 'fmt.Print' also as a goroutine.

```
func printUpperAlphabets() {
   for alphabet := byte('A'); alphabet <= byte('Z'); alphabet++ {
      go fmt.Print(string(alphabet), " ")
   }
```

```
    fmt.Println()
}
```

How to use multi processors?
By default all the Go applications run on single processor, if you want to utilize all the processors, you can do it by using 'runtime' package.

runtime.GOMAXPROCS(8)
Above statement sets the maximum number of CPUs to 8, that can be executing simultaneously

App.go

```
package main

import (
	"fmt"
	"runtime"
	"time"
)

func main() {
	runtime.GOMAXPROCS(8)

	go printUpperAlphabets()

	time.Sleep(2 * time.Second)
}

func printUpperAlphabets() {
	for alphabet := byte('A'); alphabet <= byte('Z'); alphabet++ {
		go fmt.Print(string(alphabet), " ")
	}

	fmt.Println()
}
```

Go language: Convert string to byte array

Below statement converts string to byte array.
[]byte("Hello World

Test.go

```
package main

import (
 "fmt"
)

func main() {
msg := "Hello World"

msgBytes := []byte(msg)

fmt.Println("msg : ", msg)
fmt.Println("msgBytes : ", msgBytes)
}
```

Output

```
msg :  Hello World
msgBytes :  [72 101 108 108 111 32 87 111 114 108 100]
```

Go language: Generate random uuid or string

Below snippet generate random string.

```
func GenerateRandomString() *string {
	tempBytes := make([]byte, 16)
	_, err := rand.Read(tempBytes)

	if err != nil {
		fmt.Println("Error: ", err)
		return nil
	}

	uuid := fmt.Sprintf("%X-%X-%X-%X-%X", tempBytes[0:4],
tempBytes[4:6], tempBytes[6:8], tempBytes[8:10], tempBytes[10:])

	return &uuid
}
```

Test.go

```
package main

import (
 "crypto/rand"
 "fmt"
)

func GenerateRandomString() *string {
tempBytes := make([]byte, 16)
_, err := rand.Read(tempBytes)

 if err != nil {
  fmt.Println("Error: ", err)
  return nil
}

uuid := fmt.Sprintf("%X-%X-%X-%X-%X", tempBytes[0:4], tempBytes[4:6],
tempBytes[6:8], tempBytes[8:10], tempBytes[10:])

 return &uuid
}


func main(){
randomString := *GenerateRandomString()

fmt.Println(randomString)
}
```

Output

```
D968C2BE-F2E0-9610-9EAE-991E808165E7
```

Go language: Write a program to print lowercase alphabets

App.go

```
package main

import (
 "fmt"
)

func main() {

 for start := byte('a'); start <= byte('z'); start++ {
  fmt.Print(string(start) + " ")
}

fmt.Println()
}
```

Output

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

Go language: Sleep goroutine for some time

“time” package provides sleep method to sleep/pause the goroutine for some time.

Example

```
time.Sleep(5 * time.Second)
```

Above statement make the main goroutine to sleep for 5 seconds. If you pass a negative or zero duration as an argument to Sleep method, then it causes Sleep to return immediately

App.go

```
package main

import (
 "fmt"
 "time"
)

func main() {

fmt.Println("Going to sleep for 5 seconds")

time.Sleep(5 * time.Second)

fmt.Println("I woke up after 5 seconds")
}
```

Output

```
Going to sleep for 5 seconds
I woke up after 5 seconds
```

Go language: Repeat a string n times

‘strings.Repeat’ function is used to repeat a string given number of times.

Example

```
msg1 := strings.Repeat("*", 50)
```

Above statement stores 50 *’s in msg1.

App.go

```go
package main

import (
 "fmt"
 "strings"
)

func main() {
msg1 := strings.Repeat("*", 50)
msg2 := strings.Repeat("@", 50)

fmt.Println(msg1)
fmt.Println(msg2)
}
```

Output

```
**************************************************
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@@@@@@@
```

Go Language: Concatenate strings

In this post, I am going to explain different ways to concatenate strings.

a. Using concatenate operator

```
result := ""
for _, str := range data {
    result = result + str
}
```

App.go

```go
package main

import "fmt"

func main() {
    data := []string{"Hello", "world", "How", "Are", "You"}

    result := ""
    for _, str := range data {
        result = result + str
    }

    fmt.Println(result)
}
```

Output

```
HelloworldHowAreYou
```

Since we are appending string to previous concatenated string, this approach is not efficient.

Approach 2: Using strings.Builder

```
var result strings.Builder
```

```
for _, str := range data {
    result.WriteString(str)
}

fmt.Println(result.String())
```

App.go

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    data := []string{"Hello", "world", "How", "Are", "You"}

    var result strings.Builder

    for _, str := range data {
        result.WriteString(str)
    }

    fmt.Println(result.String())
}
```

Output

```
HelloworldHowAreYou
```

I will prefer this approach.

Approach 3: Using bytes.Buffer

```
var buffer bytes.Buffer

for _, str := range data {
    buffer.WriteString(str)
}

fmt.Println(buffer.String())
```

App.go

```
package main

import (
    "bytes"
    "fmt"
)

func main() {
    data := []string{"Hello", "world", "How", "Are", "You"}

    var buffer bytes.Buffer

    for _, str := range data {
        buffer.WriteString(str)
  abc }

    fmt.Println(buffer.String())
}
```

Approach 4: Using built-in copy() function

func copy(dst, src []Type) int

The copy built-in function copies elements from a source slice into a destination slice.

```
result := make([]byte, 100)

currentLength := 0

for _, str := range data {
    currentLength += copy(result[currentLength:], []byte(str))
```

```go
}

fmt.Println(string(result))
```

App.go

```go
package main

import (
    "fmt"
)

func main() {
    data := []string{"Hello", "world", "How", "Are", "You"}

    result := make([]byte, 100)

    currentLength := 0

    for _, str := range data {
        currentLength += copy(result[currentLength:], []byte(str))
    }

    fmt.Println(string(result))
}
```

Approach 5: Using strings.join method

```go
result := strings.Join(data, "")
```

App.go

```go
package main

import (
    "fmt"
    "strings"
)

func main() {
    data := []string{"Hello", "world", "How", "Are", "You"}

    result := strings.Join(data, "")

    fmt.Printlnab(result)
}
```

Approach 6: Using fmt.SPrintf

App.go

```
package main

import (
    "fmt"
)

func main() {
    msg1 := "Hello"
    msg2 := "World"

    result := fmt.Sprintf("%s %s", msg1, msg2)

    fmt.Println(result)

}
```

ab

Approach 7: Using fmt.Sprint

App.go

```
package main

import (
    "fmt"
)

func main() {
    msg1 := "Hello"
    msg2 := "World"
    result := fmt.Sprint(msg1, msg2)

    fmt.Println(result)

}
```

Go Language: Multi line strings

You can write multi line strings by placing the string in backticks operator.

Example

```
    msg := `Hello
    World,
    How
    Are
    You`

    fmt.Println(msg)
```

App.go

```go
package main

import "fmt"

func main() {
    msg := `Hello
    World,
    How
    Are
    You`

    fmt.Println(msg)
}
```

Output

```
Hello
        World,
        How
        Are
        You
```

Backticks will not interpret escape characters

App.go
```
package main

import "fmt"

func main() {
    msg := `Hello
    \n World,
    \t How
    \a Are
    You`

    fmt.Println(msg)
}
```

a **Output**
```
Hello
        \n World,
        \t How
        \a Are
        You
```

If you want to preserve escape sequences, you can use ‘+’ operator.

```
    msg := "Hello " +
        "\n World" +
        "\t How" +
        "\a Are" +
        "You"
```

App.go
```
package main
```

```
import "fmt"

func main() {
    msg := "Hello " +
        "\n World" +
        "\t How" +
        "\a Are" +
        "You"

    fmt.Println(msg)
}
```

Output

```
Hello
 World   How AreYou
```

Go Language: Convert byte array to string

Below statement is used to convert byte array to string.

Approach 1: Using string()
Syntax

```
msg := string(byteArray[:])
```

App.go

```
package main

import "fmt"

func main() {
	arr := []byte{97, 98, 99, 100, 101, 102, 103}

	msg := string(arr)

	fmt.Println("msg : "
, msg) }
```

Output

```
msg :  abcdefg
```

Approach 2: Using fmt.Sprintf
Syntax

```
msg := fmt.Sprintf("%s", arr)
```

App.go

```
package main

import "fmt"

func main() {
	arr := []byte{97, 98, 99, 100, 101, 102, 103}

	msg := fmt.Sprintf("%s", arr)

	fmt.Println("msg : ", msg)

}
```

Approach 3: Using bytes.NewBuffer
Syntax
msg := bytes.NewBuffer(byteArray).String()

App.go

```
package main

import (
	"bytes"
	"fmt"
)

func main() {
	byteArray := []byte{97, 98, 99, 100, 101, 102, 103}

	msg := bytes.NewBuffer(byteArray).String()

	fmt.Println("msg : ", msg)

}
```

For-each loop in Go language

for loop with range keyword used to iterate over the elements of string, array, map and slice.

Iterate over an array

```
byteArray := []byte{97, 98, 99, 100, 101, 102, 103}

for index, value := range byteArray {
	fmt.Println(index, ":", value)
}
```

Iterate over a map

```
employees := map[int]string{}

employees[0] = "JJTam"
employees[1] = "Chamu"
employees[2] = "Ram"

for key, value := range employees {
	fmt.Println(key, " : ", value)
}
```

App.go

```go
package main

import (
	"fmt"
)

func main() {
	byteArray := []byte{97, 98, 99, 100, 101, 102, 103}

	fmt.Println("Elements of byte array")
	for index, value := range byteArray {
		fmt.Println(index, ":", value)
	}

	employees := map[int]string{}

	employees[0] = "JJTam"
	employees[1] = "Chamu"
	employees[2] = "Ram"

	fmt.Println("\nValues in map are")
	for key, value := range employees {
		fmt.Println(key, " : ", value)
	}
}
```

Output

```
Elements of byte array
0 : 97
1 : 98
2 : 99
3 : 100
4 : 101
5 : 102
6 : 103

Values in map are
0  :  JJTam
1  :  Chamu
2  :  Ram
```

Go language: Convert integer to string

Approach 1: str := strconv.Itoa(123)
'strconv.Itoa' method converts an integer to string.

App.go

```
package main

import (
	"fmt"
	"reflect"
	"strconv"
)

func main() {
	str := strconv.Itoa(123)

	fmt.Println(str, reflect.TypeOf(str))

}
a
```

Output

```
123 string
```

Approach 2: Using fmt.Sprintf("%v",value)

App.go

```
package main

import (
	"fmt"
	"reflect"
)

func main() {
	str := fmt.Sprintf("%v", 123)

	fmt.Println(str, reflect.TypeOf(str))

}
```

Output

```
123 string
```

Approach 3: Using FormatInt method

str := strconv.FormatInt(int64(123), 10)

App.go

```go
package main

import (
	"fmt"
	"reflect"
	"strconv"
)

func main() {
	str := strconv.FormatInt(int64(123), 10)

	fmt.Println(str, reflect.TypeOf(str))

}
```

Output

```
123 string
```

Go language: Find type of object

Approach 1: Using 'reflect.TypeOf' method

App.go

```
package main

import (
	"fmt"
	"reflect"
)

type Employee struct {
	Name string
	ID   int
}

func main() {

	i := 10
	j := 10.11
	k := "JJTam"
	emp := Employee{}

	fmt.Println("Type of i : ", reflect.TypeOf(i))
	fmt.Println("Type of j : ", reflect.TypeOf(j))
	fmt.Println("Type of k : ", reflect.TypeOf(k))
	fmt.Println("Type of emp : ", reflect.TypeOf(emp))
}
```

Output

```
Type of i :  int
Type of j :  float64
Type of k :  string
Type of emp :  main.Employee
```

Approach 2: Using fmt.Sprintf("%T", v)

App.go

```go
package main

import (
	"fmt"
)

type Employee struct {
	Name string
	ID   int
}

func main() {

	i := 10
	j := 10.11
	k := "JJTam"
	emp := Employee{}

	fmt.Println("Type of i : ", fmt.Sprintf("%T", i))
	fmt.Println("Type of j : ", fmt.Sprintf("%T", j))
	fmt.Println("Type of k : ", fmt.Sprintf("%T", k))
	fmt.Println("Type of emp : ", fmt.Sprintf("%T", emp))
}
```

Output

```
Type of i :  int
Type of j :  float64
Type of k :  string
Type of emp :  main.Employee
```

Approach 3: Using reflect.ValueOf(variable).Kind()

App.go

```go
package main
```

```
import (
	"fmt"
	"reflect"
)

type Employee struct {
	Name string
	ID   int
}

func main() {

	i := 10
	j := 10.11
	k := "JJTam"
	emp := Employee{}

	fmt.Println("Type of i : ", reflect.ValueOf(i).Kind())
	fmt.Println("Type of j : ", reflect.ValueOf(j).Kind())
	fmt.Println("Type of k : ", reflect.ValueOf(k).Kind())
	fmt.Println("Type of emp : ", reflect.ValueOf(emp).Kind())
}
```

Output

```
Type of i :  int
Type of j :  float64
Type of k :  string
Type of emp :  struct
```

As you see the output, kind() method returns ‘struct’ for a struct of type Employee, whereas previous two approaches return main.Employee.

reflect.TypeOf, fmt.Sprintf gives type along with the package name.

reflect.TypeOf().Kind() gives underlining type.

Go language: Read input from console

Approach 1: Using bufio.NewScanner

```
func readData(message string) string {
    fmt.Println(message)
    scanner := bufio.NewScanner(os.Stdin)
    scanner.Scan()

    if scanner.Err() != nil {
        return "Can't read input"
    }

    return scanner.Text()
}
```

App.go

```
package main

import (
    "bufio"
    "fmt"
    "os"
)

func readData(message string) string {
    fmt.Println(message)
    scanner := bufio.NewScanner(os.Stdin)
    scanner.Scan()

    if scanner.Err() != nil {
        return "Can't read input"
    }

    return scanner.Text()
}
```

```go
func main() {
	userName := readData("Enter your name")
	city := readData("Enter your city")
	country := readData("Enter your country")

	fmt.Println("\nUser Entered\n")
	fmt.Println("User Name : ", userName)
	fmt.Println("City : ", city)
	fmt.Println("Country : ", country)
}
```

Output

```
Enter your name
JJTam
Enter your city
Bangalore
Enter your country
India

User Entered

User Name :  JJTam
City :  Bangalore
Country :  India
```

Approach 2: Using fmt.Scanln

```go
var data string
fmt.Scanln(&data)
```

App.go

```go
package main

import (
	"fmt"
)
```

```
func readData(message string) string {
    fmt.Println(message)
    var data string
    fmt.Scanln(&data)

    return data
}

func main() {
    userName := readData("Enter your name")
    city := readData("Enter your city")
    country := readData("Enter your country")

    fmt.Println("\nUser Entered\n")
    fmt.Println("User Name : ", userName)
    fmt.Println("City : ", city)
    fmt.Println("Country : ", country)
}
```

Output

```
Enter your name
JJTam
Enter your city
Bangalore
Enter your country
India

User Entered

User Name :  JJTam
City :  Bangalore
Country :  India
```

Approach 3: Using fmt.Scanf

```
var data string
fmt.Scanf("%s", &data)
```

App.go

```go
package main

import (
    "fmt"
)

func readData(message string) string {
    fmt.Println(message)
    var data string
    fmt.Scanf("%s", &data)

    return data
}

func main() {
    userName := readData("Enter your name")
    city := readData("Enter your city")
    country := readData("Enter your country")

    fmt.Println("\nUser Entered\n")
    fmt.Println("User Name : ", userName)
    fmt.Println("City : ", city)
    fmt.Println("Country : ", country)
}
```

Go language: Print structure with field names

Format specifier ‘%+v’ is used to print the structure with field names.

Example

```
fmt.Printf("%+v\n", emp)
```

App.go

```go
package main

import "fmt"

type Employee struct {
	FirstName string
	LastName  string
	id        int
}

func main() {
	emp := Employee{"JJTam", "Delhi", 123}

	fmt.Printf("%+v\n", emp)
}
```

Output

```
{FirstName:JJTam LastName:Delhi id:123}
```

LEARN PYTHON QUICKLY

CODING FOR BEGINNERS WITH HANDS ON PROJECTS

BY

J J TAM

LEARN PYTHON QUICKLY

a. Python is open source and is available on Windows, Mac OS X, and Unix operating systems. You can download and work with python for free.

b. Python is fun to experiment and easy to learn. Python provides interpreter that can be used interactively, which makes it easy to experiment with features of the language.

c. You can extend the python interpreter with new functions and data types implemented in C or C++

d. Python improves developer productivity many times as compared to C, C++ and Java.

e. No need to compile python program, it runs immediately after development.

f. We can easily port python project from one platform to another (Of course Java also provides this feature)

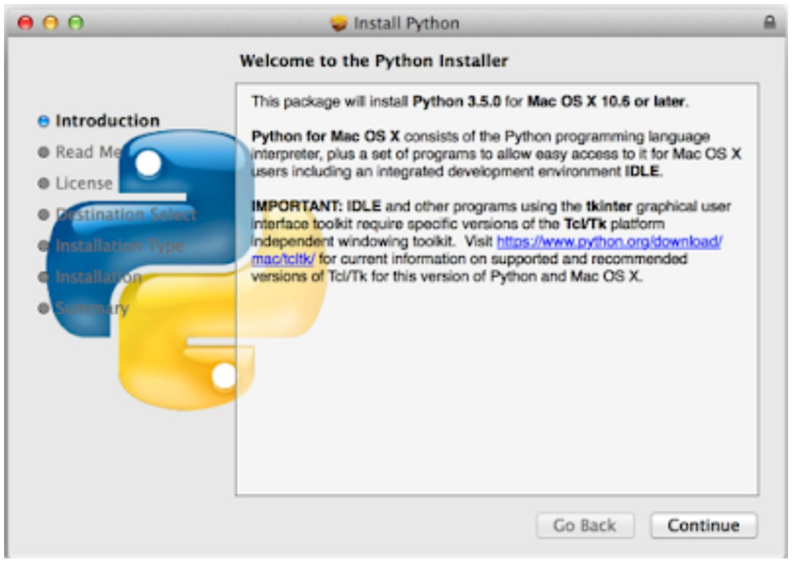
g. Rich in built library. Many third party open source libraries are available for logging, web development (Django — the popular open source web application framework written in Python), networking, database access etc.

h. Python has very great community, whatever the problem you faced in python, you will get quick help.

Install python on MAC OS

Step 1: Download python software from following location. I downloaded pkg file to install on mac os.
https://www.python.org/downloads/

Step 2: Click the pkg file.



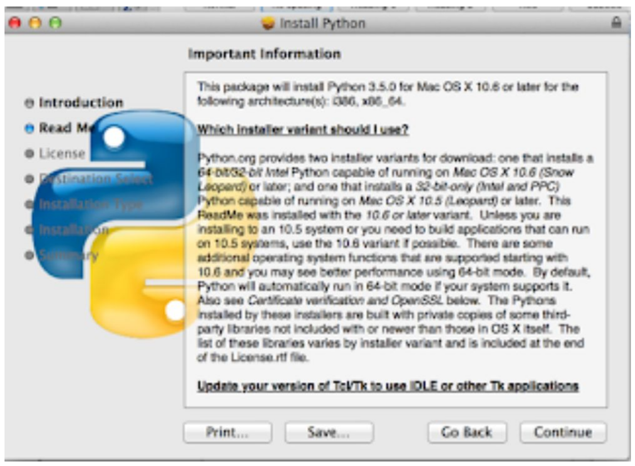
Welcome to the Python Installer

This package will install **Python 3.5.0** for **Mac OS X 10.6 or later**.

Python for Mac OS X consists of the Python programming language interpreter, plus a set of programs to allow easy access to it for Mac OS X users including an integrated development environment **IDLE**.

IMPORTANT: IDLE and other programs using the **tkinter** graphical user interface toolkit require specific versions of the **Tcl/Tk** platform independent windowing toolkit. Visit https://www.python.org/download/mac/tcltk/ for current information on supported and recommended versions of Tcl/Tk for this version of Python and Mac OS X.

Press Continue.



Important Information

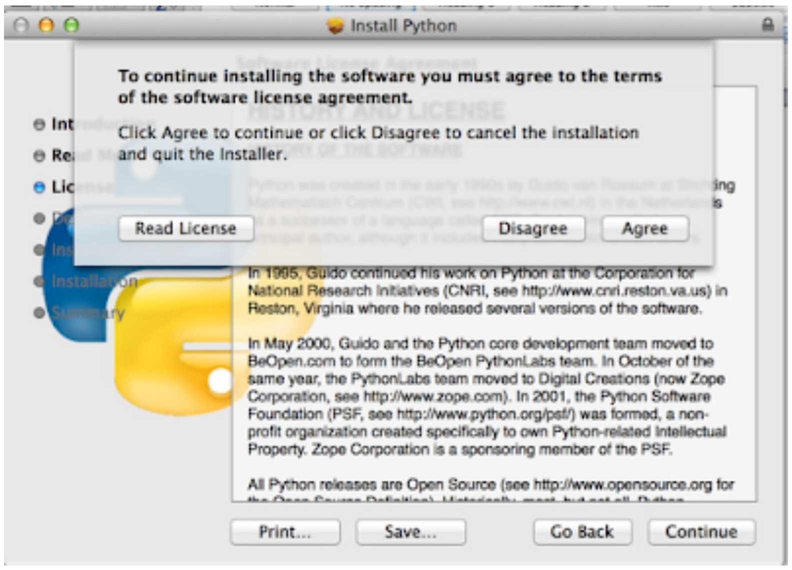
This package will install Python 3.5.0 for Mac OS X 10.6 or later for the following architecture(s): i386, x86_64.

Which installer variant should I use?

Python.org provides two installer variants for download: one that installs a *64-bit/32-bit Intel* Python capable of running on *Mac OS X 10.6 (Snow Leopard)* or later; and one that installs a *32-bit-only (Intel and PPC)* Python capable of running on *Mac OS X 10.5 (Leopard)* or later. This ReadMe was installed with the *10.6 or later* variant. Unless you are installing to an 10.5 system or you need to build applications that can run on 10.5 systems, use the 10.6 variant if possible. There are some additional operating system functions that are supported starting with 10.6 and you may see better performance using 64-bit mode. By default, Python will automatically run in 64-bit mode if your system supports it. Also see *Certificate verification and OpenSSL* below. The Pythons installed by these installers are built with private copies of some third-party libraries not included with or newer than those in OS X itself. The list of these libraries varies by installer variant and is included at the end of the License.rtf file.

Update your version of Tcl/Tk to use IDLE or other Tk applications

Press Continue.



To continue installing the software you must agree to the terms of the software license agreement.

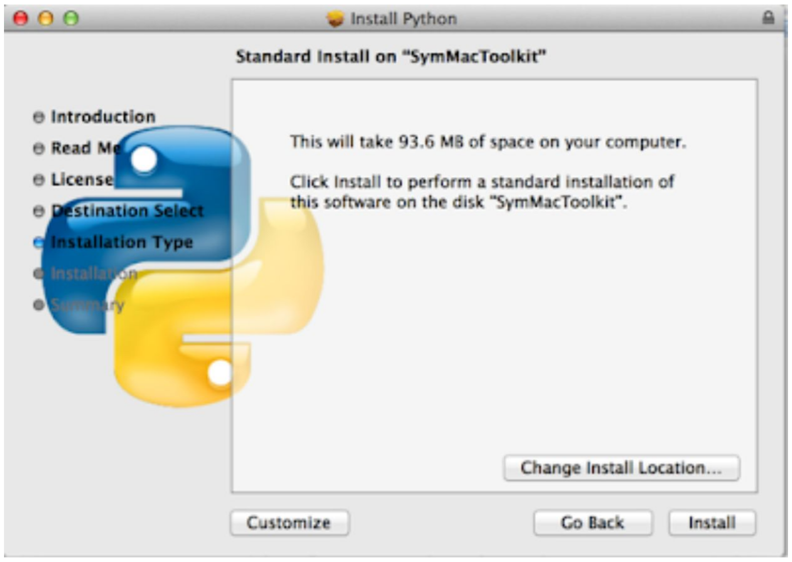
Click Agree to continue or click Disagree to cancel the installation and quit the Installer.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation, see http://www.zope.com). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org for

Accept license agreement and press Continue.

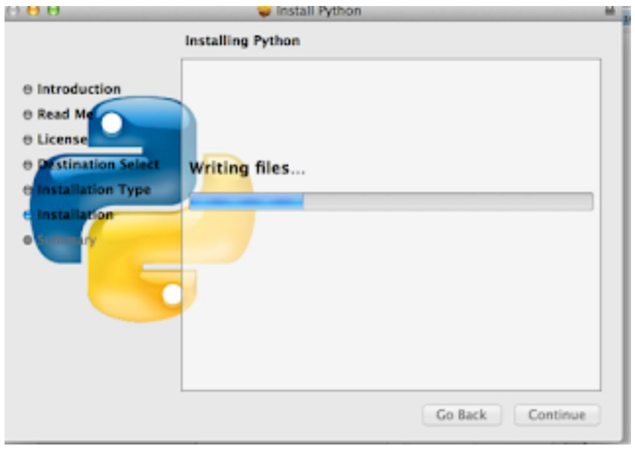


Standard Install on "SymMacToolkit"

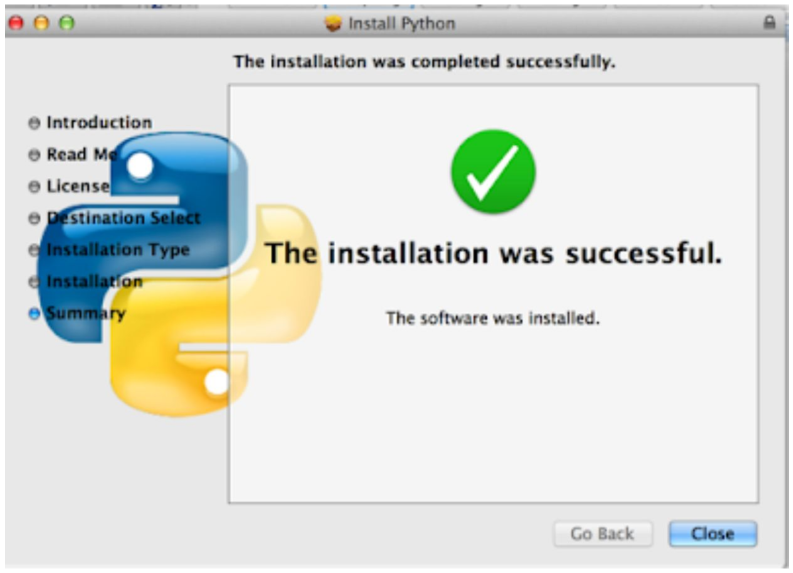
This will take 93.6 MB of space on your computer.

Click Install to perform a standard installation of this software on the disk "SymMacToolkit".

You can change the installation location use the button ‘Change Install Location’, and press the button Install.



Once installation is successful, you will get following screen.



Once installation is successful, open terminal and type python3 (Since I installed python 3.5).

```
$ python3
Python 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015, 11:00:19)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>  quit()
```

Use ‘quit()’ to exit from python prompt.

Location of python3

```
$ which python3
/Library/Frameworks/Python.framework/Versions/3.5/bin/python3

$ which python3.5
/Library/Frameworks/Python.framework/Versions/3.5/bin/python3.5
```

On windows machines python usually placed at ‘C:\Python35’.

Python: Hello World program

Open any text editor and copy the statement “print ('hello world')” and save the file name as hello.py.

hello.py

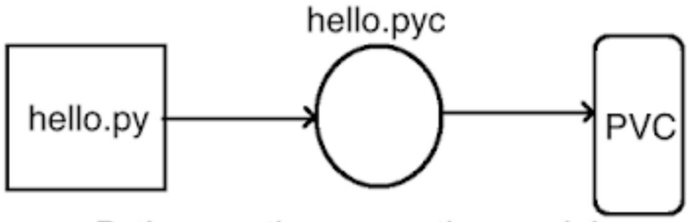
```
print ('hello world')
```

Open terminal (or) command prompt, use the command ‘python3 hello.py’ to run the file hello.py. You will get output like below

```
$ python3 hello.py
hello world
```

What happens when you instruct python to run your script?

Python first compiles your source code to byte code and sends it to python virtual machine (PVM). Byte code is the platform independent representation of your source code. PVM reads the byte code one by one and execute them.



Python runtime execution model

Note:

Byte code is not machine understandable code, it is python specific representation.

Python interactive command line

Open command prompt (or) terminal and type ‘python3’ command. It opens python interactive session.

```
$ python3
Python 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015, 11:00:19)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

How to exit from python interpreter

On Windows, a Ctrl-Z gets you out of this session; on Unix, try Ctrl-D instead. Another way is simply call the quit() function to quit from python.

```
$ python3
Python 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015, 11:00:19)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
$
```

You can type any statement (or) expressions in python interpreter, interpreter execute those statements and gives you the result.

```
>>> 10+20
30
>>> 20*20
400
```

```
>>> print('Hello World')
Hello World
```

Note:

On Unix, the Python 3.x interpreter is not installed by default with name python, so that it does not conflict with other installed Python 2.x executable.

Python: Operators

An operator is a symbol, which perform an operation. Following are the operators that python supports.

Arithmetic Operators

Relational Operators

Assignment Operators

Logical Operators

Bitwise Operators

Membership Operators

Identity Operators

Arithmetic Operators in python

Following are the arithmetic operators supported by python.

Operator	**Description**	**Example**
+	Addition	2+3
-	Subtraction	2-3
*	Multiplication	2*3
/	Division always returns a float value. To get only integer result use //.	2/3 returns 0.6666666666666666
//	Floor division discards the fractional part	2//3 returns 0
%	Returns the remainder of the division.	2%3
**	Power	2**3 returns 2 power 3 = 8

```
>>> 4+2
6
>>> 4-2
2
>>> 4/2
2.0
>>> 4*2
8
>>> 4//2
2
```

```
>>> 4**2
16
```

Relational Operators in python

Relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand.

Operator	**Description**	**Example**
==	Equal to	a==b returns true if a is equal to b, else false
!=	Not equal to	a!=b returns true if a is not equal to b, else false.
>	Greater than	a>b returns true, if a is > b, else false.
>=	Greater than or equal to	a>=b returns true, if a is >= b, else false.
<	Less than	a<b returns true, if a is < b, else false.
<=	Less than or equal to	a<=b returns true, if a is <= b, else false.

```
>>>  a =10
>>>  b =12
>>>  a == b
False

>>>  a != b
True

>>>  a > b
False

>>>  a >= b
False

>>>  a < b
```

```
True
>>>  a <= b
True
>>>
```

Assignment operators in python

Assignment operators are used to assign value to a variable. Following are the assignment operators provided by python.

Operator	Description
=	a=10 assigns 10 to variable a
+=	a+=10 is same as a=a+10
-=	a-=10 is same as a=a-10
*=	a*=10 is same as a=a*10
/=	a/=10 is same as a=a/10
//=	a//=10 is same as a=a//10
%=	a%=10 is same as a=a%10
**=	a**=10 is same as a=a**10

```
>>> a =10
>>> a
10

>>>
>>> a +=10
>>> a
20

>>>
>>> a -=10
>>> a
10

>>>
>>> a *=10
>>> a
100

>>>
>>> a /=10
>>> a
10.0
>>>
>>> a //=10
>>> a
1.0
>>>
>>> a **=10
>>> a
1.0
```

Multiple Assignments
You can assign values to multiple variables simultaneously.

```
>>> a, b, c  =  10 ,  'hello' ,  12.345
>>> a
10
>>> b
'hello'
>>> c
12.345
```

Logical Operators in python

Following are the logical operators supported by python.

Operator	Description
and	‘a and b’ returns true if both a, b are true. Else false.
or	‘a or b’ return false, if both a and b are false, else true.
not	‘not a’ Returns True if a is false, true otherwise

```
>>>  a = bool( 0 )
>>>  b = bool( 1 )
>>>
>>>  a
False

>>>  b
True

>>>
>>>  a  and  b
False

>>>
>>>  a  or  b
True

>>>
>>> not  a
True

>>>
>>> not  (a  and  b)
True

>>>
```

```
>>> not (a or b)
False
```

Bitwise Operators in python

Python supports following bitwise operators, to perform bit wise operations on integers.

Operator	Description
>>	bitwise right shift
<<	bitwise left shift
&	bitwise and
^	Bitwise Ex-OR
|	Bitwise or
~	Bitwise not

Following post, explains bitwise operators clearly.
http://self-learning-java-tutorial.blogspot.in/2014/02/bit-wise-operators.html

```
>>> a =2
>>> a >>1
1

>>>
>>> a <<1
4

>>>
>>> a &3
2

>>> a |3
3

>>>
>>> ~ a
-3
```

Membership operators in python

Membership operators are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

Operator	Description
in	Return true if value/variable is found in the sequence, else false.
not in	Return True if value/variable is not found in the sequence, else false

```
>>> primeNumbers  = [ 2 ,  3 ,  5 ,  7 ,  11 ]
>>> 2  in  primeNumbers
True

>>> 4  in  primeNumbers
False

>>> 4  not  in  primeNumbers
True
```

Identity operators in python

Identity operators are used to compare the memory locations of two objects.

Operator	Description
is	a is b returns true, if both a and b point to the same object, else false.
is not	a is not b returns true, if both a and b not point to the same object, else false.

```
>>> name1 ="Hari JJTam"
>>> name2 = name1
>>> name3 ="Hari JJTam"
>>> name4 ="abc"
>>>
>>> name1  is  name2
True

>>> name1  is  name3
False

>>> name1  is  name4
False

>>> name1  is  not  name3
True
```

Python: Short circuit operators

Boolean operators and, or are called short circuit operators, it Is because evaluation of expression stops, whenever the outcome determined.

Why Boolean and is called short circuit operator?
Since if the first statement in the expression evaluates to false, then python won't evaluates the entire expression. So boolean and is called short circuit and.

Why Boolean OR is called short circuit operator?
Since if the first statement evaluates to true, then Python won't evaluates the entire expression.

Strings in python

String is a sequence of character specified in single quotes('…'), double quotes("…"), (or) triple quotes ("""…""" or '''… '''). Strings in python are immutable, i.e., an object with a fixed value.

```
>>> str1 ='Hello World'
>>> str2 ="Hello World"
>>> str3 ="""Hello
... World
... """
>>> str1
'Hello World'
>>> str2
'Hello World'
>>> str3
'Hello\nWorld\n'
```

Special characters are escaped with backslash.

```
>>> message ='He don\'t know about this'
>>> message
"He don't know about this"
```

'print' method treat characters preceded by \ (backslash) as special characters.

```
>>> print ( 'firstline\nsecondline' )
firstline
secondline
```

As you observe output, \n prints new line. If you don't want characters prefaced by \ to be interpreted as special characters, you can use raw strings by adding an r before the first quote

```
>>> print ( r'firstline\nsecondline' )
firstline\nsecondline
```

Concatenate strings
'+' operator is used to concatenate strings.

```
>>>  hello ="Hello,"
>>>  message ="How are you"
>>>  info = hello + message
>>>  info
'Hello,How are you'
```

Two or more string literals next to each other are automatically concatenated.

```
>>> 'Hello'  "How"  'are'  'you'
'HelloHowareyou'
```

Repeat strings
By using '*', we can repeat the strings.

```
>>> 'hi'*2
'hihi'
>>> 'hi'*2+'hello'*3
'hihihellohellohello'
```

Access specific character from strings
You can access, specific character of string using index position.

```
>>>  name ="Hello World"
>>>  name[ 0 ]
'H'
>>>  name[ 6 ]
```

```
'W'
```

Index 0 represents 1st character, 1 represents 2nd character etc.,

You can also use negative numbers for indexing.

-1 represents last character; -2 represents second-last character etc.,

```
>>> name
'Hello World'
>>> name[ -1 ]
'd'
>>> name[ -2 ]
'l'
>>> name[ -7 ]
'o'
```

Slicing
Slicing is used to get sub string of given string.

Example	**Description**
string[start:end]	Returns sub string from index start (included) to end index (excluded).
string[:end]	Returns sub string from index 0(included) to end index (excluded).
string[start:]	Return sub string from index start to till end.
string[-2:]	Return characters from 2nd last to end.

```
>>> message ="Hello World"
>>>
>>> message[ 0 : 5 ]
```

```
'Hello'
>>> message[ 5 :]
' World'
>>> message[:]
'Hello World'
>>> message[ -2 :]
'ld'
>>> message[ -5 : -2 ]
'Wor'
```

Get length of the string

Function ‘len(str)’ is used to get the length of the string.

```
>>> message
'Hello World'
>>> len(message)
11
>>> len( "How are you" )
11
```

Python: if condition

“if” statement
"if" tell the program execute the section of code when the condition evaluates to true.

Syntax
if_stmt ::= "if" expression ":" suite

test.py

```
a =10

if (a < 10) :

 print("a is less than 10")

if (a == 10) :

 print("a is equal to 10")

if( a > 10) :

 print("a is greater than 10")
```

```
$ python3 test.py
a is equal to 10
```

if-else statement
If the condition true, then if block code executed. other wise else block code executed.

Syntax

```
if_stmt ::=  "if" expression ":" suite
             ["else" ":" suite]
```

test.py

```
a =10
```

```
if (a != 10) :
 print("a is not equal to 10")
else :
 print("a is equal to 10")
```

if-elif-else statement
By using if-elif-else construct, you can choose number of alternatives. An if statement can be followed by an optional elif...else statement.

Syntax

```
if_stmt ::=  "if" expression ":" suite
             ( "elif" expression ":" suite )*
             ["else" ":" suite]
```

test.py

```
a =10

if (a > 10) :
 print("a is greater than 10")
elif (a < 10) :
 print("a is less than 10")
else :
 print("a is equal to 10")
```

```
$ python3 test.py
a is equal to 10
```

Note:
a. In python, any non-zero integer value is treated as true; zero is false.

test.py

```
if (100) :
 print("Hello")
else :
 print("Bye")
```

```
$ python3 test.py
Hello
```

b. Any sequence (string, list etc.,) with a non-zero length is true, empty sequences are false.

test.py

```
list=[]
data="abc"

if (list) :
 print("list is not empty")
else :
 print("list is empty")

if (data) :
 print("data is not empty")
else :
 print("data is empty")
```

```
$ python3 test.py
list is empty
data is not empty
```

Python: while statement

‘while’ statement executes a block of statements until a particular condition is true

Syntax

```
while_stmt ::=  "while" expression ":" suite
                ["else" ":" suite]
```

‘else’ clause is optional. If the expression evaluates to false, then else clause will execute (if else present), and terminates.

test.py

```
a =2

print ( "Even numbers are" )
while(a < 10) :
 print (a, end=' ')
a +=2
else:
 print("\nExit from loop")

print ( "Done" )
```

```
$ python3 test.py
Even numbers are
2 4 6 8
Exit from loop
Done
```

Python: for statement

Python’s for statement is used to iterate over the items of any sequence like a list, string.

Syntax

```
for_stmt ::=  "for" target_list "in" expression_list ":" suite
              ["else" ":" suite]
```

‘else’ clause is optional, it is executes, once the loop terminates.

test.py

```
names = [ "Phalgun" ,  "Sambith" ,  "Mahesh" ,  "swapna" ]

for name in names:
 print(name)
else:
 print("Exiting from loop")

print ( "Finsihed Execution" )
```

```
$ python3 test.py
Phalgun
Sambith
Mahesh
swapna
Exiting from loop
Finsihed Execution
```

Python: break statement

'break' statement is used to come out of loop like for, while.

test.py

```
i = 0

while (1):
i+=2
 print(i)
 if(i==10):
  break
else:
 print("Exiting loop")

print( "Finished Execution" )
```

```
$ python3 test.py
2
4
6
8
10
Finished Execution
```

As you observe the output, print statement in else clause is not printed. It is because, else clause will not execute, when a break statement terminates the loop.

Python: continue statement

‘continue’ statement skips the current iteration of loops like for, while.

test.py

```
i  =  2

while (i < 20):

i+=2

 if(i%2 != 0):

  continue

 print(i)

else:

 print("Exiting loop")

print( "Finished Execution" )
```

Above program prints all the even numbers up to 20 (exclusive).

```
$ python3 test.py
4
6
8
10
12
14
16
18
20
Exiting loop
Finished Execution
```

Python: functions

A function is a block of statements identified by a name. The keyword 'def' is used to define a function. Functions are mainly used for two reasons.

ı. To make code easier to build and understand
›. To reuse the code

Syntax

def functionName(argument1, argument2 …. argumentN):

test.py

```
def factorial(n):
 if(n<=1):
  return 1

result =1

 for i in range(2, n+1):
  result *= i
 return result

print(factorial( 1 ))
print(factorial( 2 ))
print(factorial( 3 ))
print(factorial( 4 ))
print(factorial( 5 ))
```

```
$ python3 test.py
1
2
6
24
120
```

Variables created before function definition may be read inside of the function only if the function does not change the value.
test.py

```
# Create the x variable and set to 44

x  =  44

# Define a simple function that prints x

def f():

x += 1

 print(x)


# Call the function

f()
```

Run above program, you will get following error.

```
$ python3 test.py
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    f()
  File "test.py", line 7, in f
    x += 1
UnboundLocalError: local variable 'x' referenced before assignment
```

Remove ‘x+=1’ statement and re run the above program, value of x is printed to console.

Python: functions: return statement

‘return’ statement is used to return a value from function to the caller.

test.py

```
def factorial(n):
 if(n<=1):
  return 1

result =1

 for i in range(2, n+1):
  result *= i
 return result

print(factorial( 1 ))
print(factorial( 2 ))
print(factorial( 3 ))
print(factorial( 4 ))
print(factorial( 5 ))
```

```
$ python3 test.py
1
2
6
24
120
```

‘return’ without an expression argument returns None.

```
def hello():
	print("Hello")
print(hello())

$ python3 test.py
Hello
None
```

Python: functions: Default Argument Values

In Python, you can pass default values to function arguments.

```
def welcomeMessage(name='user', message='Hello'):
        print(message, name)
```

You can call above function in following ways.

welcomeMessage() : prints ‘Hello user‘
welcomeMessage('JJTam') : prints ‘Hello JJTam‘
welcomeMessage('JJTam', "Welcome") : prints ‘Welcome JJTam‘.

test.py

```
def welcomeMessage(name='user', message='Hello'):
 print(message, name)



welcomeMessage()
welcomeMessage('JJTam')
welcomeMessage('JJTam', "Welcome")
```

```
$ python3 test.py
Hello user
Hello JJTam
Welcome JJTam
```

Note:
Default value is evaluated only once. So be cautious, while using mutable objects like lists, dictionary etc., For example, following function accumulates the result in subsequent calls.

test.py

```
def getEvenNumbers(limit, result=[]):
```

```
i=2
 while(i < limit):
  result.append(i)
  i+=2

  return  result

print(getEvenNumbers( 4 ))
print(getEvenNumbers( 8 ))
print(getEvenNumbers( 12 ))
```

```
$ python3 test.py
[2]
[2, 2, 4, 6]
[2, 2, 4, 6, 2, 4, 6, 8, 10]
```

If you don’t want the default to be shared between subsequent calls, you can write the function like below instead:

```
def getEvenNumbers(limit, result=None):
 if result is None:
  result=[]

i=2
 while(i < limit):
  result.append(i)
  i+=2

  return  result

print(getEvenNumbers( 4 ))
print(getEvenNumbers( 8 ))
print(getEvenNumbers( 12 ))
```

```
$ python3 test.py
[2]
[2, 4, 6]
[2, 4, 6, 8, 10]
```

Python: functions: Keyword arguments

Functions can also be called using keyword arguments, i.e, by using argument names.

```
def printEmployee(id, firstName, lastName, designation='Engineer'):
	print(id, firstName, lastName, designation)
```

Above function can be called in following ways.

printEmployee(id=1, lastName='Delhi', firstName='Hari')
printEmployee(lastName='JJTam', firstName='Rama', id=2)
printEmployee(designation='Tech Lead', lastName='Battu', firstName='Gopi', id=3)

test.py

```
def printEmployee(id, firstName, lastName, designation='Engineer'):
 print(id, firstName, lastName, designation)

printEmployee(id=1, lastName='Delhi', firstName='Hari')
printEmployee(lastName='JJTam', firstName='Rama', id=2)
printEmployee(designation='Tech Lead', lastName='Battu',
firstName='Gopi', id=3)
```

```
$ python3 test.py
1 Hari Delhi Engineer
2 Rama JJTam Engineer
3 Gopi Battu Tech Lead
```

Note:

1. Keyword arguments must follow positional arguments.

```
def printEmployee(id, firstName, lastName, designation='Engineer'):
	print(id, firstName, lastName, designation)
```

```
printEmployee(1, lastName='Delhi', firstName='Hari', "engineer")
```

When you tries to compile above file, you will get following error.

```
$ python3 test.py
  File "test.py", line 4
    printEmployee(1, lastName='Delhi', firstName='Hari', "engineer")
                                                        ^
SyntaxError: positional argument follows keyword argument
```

Python lists

List is group of values in between square brackets separated by commas.

```
>>> primes = [ 2 , 3 , 5 , 7 , 11 , 13 , 17 , 19 , 23 ]
>>> primes
[2, 3, 5, 7, 11, 13, 17, 19, 23]
```

‘primes’ is a list that contain prime numbers.

```
>>> students = [ "Hari" , "JJTam" , "Kiran" , "Ravi" ]
>>> students
['Hari', 'JJTam', 'Kiran', 'Ravi']
```

‘students’ is a list that contain all student names.

List can contain any type of data.

```
>>> objects = [ 1 , 3 , "Hello" , 10.23 ]
>>> objects
[1, 3, 'Hello', 10.23]
```

You can access elements of list by using index.

```
>>> objects
[1, 3, 'Hello', 10.23]
>>> objects[ 0 ]
1
>>> objects[ 2 ]
'Hello'
```

objects[0] return the first element of list objects.
objects[1] return the second element of list objects.

-ve indexes also used to access elements of a list.

```
>>> objects
[1, 3, 'Hello', 10.23]
>>> objects[ -1 ]
10.23
>>> objects[ -3 ]
3
```

Slicing
Slicing is used to get sub list.

Example	Description
list[start:end]	Returns sub list from index start (included) to end index (excluded).
list[:end]	Returns sub list from index 0(included) to end index (excluded).
list[start:]	Return sub list from index start to till end.
list[-2:]	Return list from 2nd last to end.

```
>>> objects
[1, 3, 'Hello', 10.23]
>>> objects[ -1 ]
10.23
>>> objects[ -3 ]
3
>>> objects
[1, 3, 'Hello', 10.23]
>>>
>>> objects[:]
[1, 3, 'Hello', 10.23]
>>>
```

```
>>> objects[ 2 :]
['Hello', 10.23]
>>>
>>> objects[: 3 ]
[1, 3, 'Hello']
>>>
>>> objects[ -2 :]
['Hello', 10.23]
```

Concatenate two lists

'+' Operator is used to concatenate two lists.

```
>>> even = [ 2 , 4 , 6 , 8 , 10 ]
>>> odd = [ 1 , 3 , 5 , 7 , 9 ]
>>> numbers = even + odd
>>> numbers
[2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
```

Lists are mutable; you can change the values of list.

```
>>> numbers
[2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
>>> numbers[ 0 ] =12
>>> numbers[ 1 ] =14
>>> numbers
[12, 14, 6, 8, 10, 1, 3, 5, 7, 9]
```

Add elements to end of list

List provides ‘append’ method to add new elements to the end of a list.

```
>>> numbers
[12, 14, 6, 8, 10, 1, 3, 5, 7, 9]
>>>
>>> numbers . append( 11 )
>>> numbers . append( 13 )
```

```
>>>
>>> numbers
[12, 14, 6, 8, 10, 1, 3, 5, 7, 9, 11, 13]
```

Assignment to slices
If you want to replace sequence of elements in a list, you can use slice notation.

numbers[2:5] = [21, 22, 23]
Above statement replace elements at index 2, 3, 4 with 21, 22, 23 respectively.

numbers[:] = []
Above statement clear the list by replacing all the elements with an empty list.

```
>>> numbers
[12, 14, 6, 8, 10, 1, 3, 5, 7, 9, 11, 13]
>>>
>>> numbers[ 2 : 5 ] = [ 21 ,  22 ,  23 ]
>>> numbers
[12, 14, 21, 22, 23, 1, 3, 5, 7, 9, 11, 13]
>>>
>>> numbers[:]  =  []
>>> numbers
[]
```

Get length of the list
By using 'len' function, you can get the length of the list.

```
>>> vowels = [ 'a' ,  'e' ,  'i' ,  'o' ,  'u' ]
>>> len(vowels)
5
```

Nested lists
A list can be nested in other list. For example, in below example, numbers contains 3 lists, first list represent odd numbers, second list represent even numbers and third list represent prime numbers.

```
>>> numbers = [[ 1 ,  3 ,  5 ,  7 ],[ 2 ,  4 ,  6 ,  8 ],[ 2 ,  3 ,  5 ,  7 ,  11 ]]
>>> numbers
[[1, 3, 5, 7], [2, 4, 6, 8], [2, 3, 5, 7, 11]]
>>>
>>> numbers[ 0 ]
[1, 3, 5, 7]
>>>
>>> numbers[ 1 ]
[2, 4, 6, 8]
>>>
>>> numbers[ 2 ]
[2, 3, 5, 7, 11]
>>>

>>>

>>> len(numbers)
3
>>> len(numbers[ 0 ])
4
>>> len(numbers[ 1 ])
4
>>> len(numbers[ 2 ])
5
>>>
>>> numbers[ 0 ][ 1 ] =9
>>> numbers[ 1 ][ 1 : 4 ]  = [ 10 ,  12 ,  14 ]
>>> numbers
[[1, 9, 5, 7], [2, 10, 12, 14], [2, 3, 5, 7, 11]]
```

Python: list extend: append elements of given list

Python provides ‘extend’ method to append all the items in the given list.

test.py

```
evenNumbers=[2, 4, 6]

oddNumbers=[1, 3, 5]

numbers=[]


numbers . extend(evenNumbers)
print(numbers)

numbers . extend(oddNumbers)
print(numbers)
```

```
$ python3 test.py
[2, 4, 6]
[2, 4, 6, 1, 3, 5]
```

Python: insert: Insert an element at given position

List provides ‘insert’ method to insert an element at given position.

Syntax

list.insert(i, x)

list.insert(0, x) : Insert element x at front of list.
list.insert(len(list), x) : Insert element x at the end of the list.
list.insert(2, x) : Insert element x at 2nd position.

test.py

```
list = [ 2 ,  4 ,  6 ]
print(list)

list . insert( 0 ,  -2 )
print(list)

list . insert(len(list),  8 )
print(list)

list . insert( 2 ,  3 )
print(list)
```

```
$ python3 test.py
[2, 4, 6]
[-2, 2, 4, 6]
[-2, 2, 4, 6, 8]
[-2, 2, 3, 4, 6, 8]
```

Python: list remove: Remove an element

List provides 'remove' method to remove an element 'x' from it.

Syntax

list.remove(x)

'remove' method removes the first item from the list, whose value is x.

test.py

```
list = [ 2 , 4 , 6 , 6 , 4 , 2 ]
print(list)

list . remove( 6 )
print(list)
```

```
$ python3 test.py
[2, 4, 6, 6, 4, 2]
[2, 4, 6, 4, 2]
```

'remove' method throws an error, if element is not in the list.

test.py

```
list = [ 2 , 4 , 6 , 6 , 4 , 2 ]
print(list)

list . remove( 60 )
print(list)
```

```
$ python3 test.py
[2, 4, 6, 6, 4, 2]
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    list.remove(60)
```

ValueError: list.remove(x): x not in list

Python: list: pop: remove last element in the list

list.pop([i])

‘pop’ method without argument removes last element in the list. If you specify the position ‘i’, it removes the element at position ‘i’ and return it.

test.py

```
list = [ 2 ,  4 ,  6 ,  6 ,  4 ,  2 ]
print(list)

print( "Removing last element " , list . pop())
print(list)

print( "Removing element at index 2 " , list . pop( 2 ))
print(list)
```

```
$ python3 test.py
[2, 4, 6, 6, 4, 2]
Removing last element  2
[2, 4, 6, 6, 4]
Removing element at index 2  6
[2, 4, 6, 4]
```

Python: list: clear all elements in the list

List provides 'clear' method to clear all elements in the list.

test.py

```
list = [ 2 ,  4 ,  6 ,  6 ,  4 ,  2 ]
print(list)

list . clear()
print(list)
```

```
$ python3 test.py
[2, 4, 6, 6, 4, 2]
[]
```

Python: list: index: Get index of element

list.index(x)
‘list’ provides index method, which returns the index of the first item whose value is x.

test.py

```
list = [ 2 ,  4 ,  6 ,  6 ,  4 ,  2 ]
print(list)

print( "index of 2 is " , list . index( 2 ))
print( "index of 4 is " , list . index( 4 ))
print( "index of 6 is " , list . index( 6 ))
```

```
$ python3 test.py
[2, 4, 6, 6, 4, 2]
index of 2 is  0
index of 4 is  1
index of 6 is  2
```

Throws an error, if element is not in the list.
test.py

```
list = [ 2 ,  4 ,  6 ,  6 ,  4 ,  2 ]
print(list)

print( "index of 20 is " , list . index( 20 ))
```

```
$ python3 test.py
[2, 4, 6, 6, 4, 2]
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    print("index of 20 is ", list.index(20))
```

ValueError: 20 is not in list

Python: list: sort elements of list

‘list’ provides sort method to sort elements of list. By default, sort method sort elements in ascending order. To sort elements in descending order, pass ‘reverse=True’ argument.

test.py

```
list = [ 12 ,  5 ,  32 ,  43 ,  21 ,  356 ]
print(list)

list . sort()
print(list)

list . sort(reverse =True )
print(list)
```

```
$ python3 test.py
[12, 5, 32, 43, 21, 356]
[5, 12, 21, 32, 43, 356]
[356, 43, 32, 21, 12, 5]
```

Python: count: Count number of times element appears

List provides count method, to count number of times element appear in the result.

test.py

```
list = [ 12 ,  5 ,  32 ,  43 ,  21 ,  356 ,  5 ,  21 ]
print(list)

print( "5 appears " ,list . count( 5 ), " times" )
print( "12 appears " ,list . count( 12 ), " times" )
```

```
$ python3 test.py
[12, 5, 32, 43, 21, 356, 5, 21]
5 appears  2  times
12 appears  1  times
```

Python: list: Reverse elements of list

List provides reverse method to reverse elements of list.

test.py

```
list = [ 12 ,  5 ,  32 ,  43 ,  21 ,  356 ,  5 ,  21 ]
print(list)

list . reverse()
print(list)
```

```
$ python3 test.py
[12, 5, 32, 43, 21, 356, 5, 21]
[21, 5, 356, 21, 43, 32, 5, 12]
```

Python: list: Copy elements of list

List provides copy method, to get a shallow copy of the list.

test.py

```
list=[12, 5, 32, 43, 21, 356, 5, 21]
list1 = list.copy()


print(list)
print(list1)

list1 . append( 12345 )

print(list)
print(list1)
```

```
$ python3 test.py
[12, 5, 32, 43, 21, 356, 5, 21]
[12, 5, 32, 43, 21, 356, 5, 21]
```

```
[12, 5, 32, 43, 21, 356, 5, 21]
[12, 5, 32, 43, 21, 356, 5, 21, 12345]
```

Python:list: del: delete elements

By using del statement, you can remove an element from a list using index.

test.py

```
list = [ 12 ,  5 ,  32 ,  43 ,  21 ,  356 ,  5 ,  21 ]

print(list)

del  list[ 0 ]
del  list[ 5 ]

print(list)
```

```
$ python3 test.py
[12, 5, 32, 43, 21, 356, 5, 21]
[5, 32, 43, 21, 356, 21]
```

By using del statement, you can remove slices from list.

test.py

```
list = [ 12 ,  5 ,  32 ,  43 ,  21 ,  356 ,  5 ,  21 ]
print(list)

del  list[ 2 : 6 ]
print(list)
```

```
$ python3 test.py
[12, 5, 32, 43, 21, 356, 5, 21]
[12, 5, 5, 21]
```

By using del statement, you can clear list. 'del list[:]' removes all elements from list list.

test.py

```
list = [ 12 ,  5 ,  32 ,  43 ,  21 ,  356 ,  5 ,  21 ]
print(list)

del  list[:]
print(list)
```

```
$ python3 test.py
[12, 5, 32, 43, 21, 356, 5, 21]
[]
```

By using del statement, you can remove variable itself. Once you remove variable, you can’t use it. If you try to use the variable, after it removed, you will get error.

test.py

```
list = [ 12 ,  5 ,  32 ,  43 ,  21 ,  356 ,  5 ,  21 ]
print(list)

del  list
list[ 0 ]  =  1
```

```
$ python3 test.py
[12, 5, 32, 43, 21, 356, 5, 21]
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    list[0] = 1
TypeError: 'type' object does not support item assignment
```

Python: Looping over lists

Using for loop

for var in list:
    # Do processing here

You can use for loop like above to loop over a list. Following example explains this.

```
>>>  countries = [ 'India' ,  'China' ,  'Pakisthan' ,  'Sri Lanka' ,  'Bangladesh' ]
>>>  for  i  in  countries:
...      print(i)
...
India
China
Pakisthan
Sri Lanka
Bangladesh
```

By using enumerate method, we can iterate over a list.

By using enumerate function

'enumerate' function return the position, corresponding value in the list.

```
>>> countries
['India', 'China', 'Pakisthan', 'Sri Lanka', 'Bangladesh']
>>>  for  i,v  in  enumerate(countries):
...      print(i, v)
...
0  India
1  China
2  Pakisthan
3  Sri Lanka
```

```
4 Bangladesh
```

To loop over a sequence reversely
'reversed' function is used to loop over a sequence reversely.

```
>>> countries
['India', 'China', 'Pakisthan', 'Sri Lanka', 'Bangladesh']
>>>
>>> for i in reversed(countries):
...     print(i)
...
Bangladesh
Sri Lanka
Pakisthan
China
India
```

To loop over a sequence in sorted order
Use 'sorted' function to loop over a sequence in sorted order.

```
>>> for i in sorted(countries):
...     print(i)
...
Bangladesh
China
India
Pakisthan
Sri Lanka
```

Loop over more than one list at a time
'zip' function is used to loop over more than one list at a time.

```
>>> for country, capital in zip(countries, capitals):
...     print(country, capital)
...
India Delhi
```

```
China Beijing
Pakisthan Islamabad
Sri Lanka Sri Jayawardenepura - Kotte
Bangladesh Dhaka
```

Loop using index

```
>>> countries
['India', 'China', 'Pakisthan', 'Sri Lanka', 'Bangladesh']
>>>
>>>  for  i  in  range(len(countries)):
...      print(countries[i])
...
India
China
Pakisthan
Sri Lanka
Bangladesh
```

Python: tuples

A tuple is just like a list, consist of number of values separated by commas.

Differences between tuple and list

a. List is mutable, where as tuple is immutable
b. Tuple can contain heterogeneous data, where as list usually contains homogeneous data.

test.py

```
employee = ( 1 ,  "Hari Krihsna" ,  "Delhi" ,  12345.678 )

print(employee)
print(employee[ 0 ])
print(employee[ 1 ])
print(employee[ 2 ])
print(employee[ 3 ])
```

```
$ python3 test.py
(1, 'Hari Krihsna', 'Delhi', 12345.678)
1
Hari Krihsna
Delhi
12345.678
```

As you observe above example, elements in tuple are enclosed in parenthesis. Eventhough tuples are immutable, you can create tuples which contain mutable objects, such as lists.

test.py

```
employee = ( 1 , [])

print(employee)
```

```
employee[1].append(2)
employee[1].append(4)
employee[1].append(6)

print(employee)
```

```
$ python3 test.py
(1, [])
(1, [2, 4, 6])
```

Packing and unpacking
You can define tuples, without using parenthesis.
For example,
employee=1, "Hari Krihsna", "Delhi", 12345.678

Above one is the example of tuple packing.

id, firstName, lastName, salary = employee
Above one is an example of tuple unpacking. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence.

test.py
```
employee =1 ,  "Hari Krihsna" ,  "Delhi" ,  12345.678

id, firstName, lastName, salary  =  employee

print(id)
print(firstName)
print(lastName)
print(salary)
```

```
$ python3 test.py
1
Hari Krihsna
Delhi
```

12345.678

Concatenate tuples
'+' operator is used to concatenate tuples.

```
>>>  tuple1 = ( 1 ,  "HI" ,  2 ,  45.65 )
>>>  tuple2 = ( "abcdef" ,  54 ,  67 )
>>>  tuple3 = tuple1 + tuple2
>>> tuple3
(1, 'HI', 2, 45.65, 'abcdef', 54, 67)
```

Slicing
Just like lists, you can access tuples using slice notation.

Example	Description
tuple[start:end]	Returns tuple from index start (included) to end index (excluded).
tuple[:end]	Returns tuple from index 0(included) to end index (excluded).
tuple[start:]	Return tuple from index start to till end.
tuple[-2:]	Return elements from 2nd last to end.

```
>>>  tuple1
(1, 'HI', 2, 45.65)
>>>  tuple1[ 0 :]
(1, 'HI', 2, 45.65)
>>>  tuple1[:]
(1, 'HI', 2, 45.65)
>>>  tuple1[: 3 ]
(1, 'HI', 2)
>>>  tuple1[ 2 : 5 ]
(2, 45.65)
```

‘*’: Repeat tuple elements
‘*’ is the repetition operator, used to repeat the elements of tuple.

```
>>> tuple1
(1, 'HI', 2, 45.65)
>>>
>>> tuple1 *3
(1, 'HI', 2, 45.65, 1, 'HI', 2, 45.65, 1, 'HI', 2, 45.65)
>>>
```

Remove tuple elements
As I said, tuples are immutable, so it is not possible to remove elements from tuple. But you can remove the entire tuple using del statement.

```
>>> tuple1 = ( 1 ,  "HI" ,  2 ,  45.65 )
>>>
>>> del  tuple1
>>> tuple1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError : name 'tuple1' is not defined
```

Observe the output, an exception raised; this is because after deletion, tuple does not exist any more.

Python: Sets

Set is an unordered collection of elements with no duplicates. We can perform union, intersection, difference, and symmetric difference operations on sets.

How to create set

You can create set using {} (or) set() function. 'set()' creates empty set.
>>> evenNumbers={2, 4, 6, 8, 8, 4, 10} >>> evenNumbers {8, 10, 2, 4, 6}

Observe above snippet, evenNumbers is a set that contains even numbers. Observe the output, set doesn't contain duplicate elements.

Following operations are supported by set.

len(s) : cardinality of set

Returns cardinality(Number of distinct elements) of the set.

```
>>> evenNumbers = { 2 ,  4 ,  6 ,  8 ,  8 ,  4 ,  10 }
>>> len(evenNumbers)
5
```

x in s : Check whether element is in set or not

'in' operator is used to check whether element is in set or not, return true if the element is set, else false.

```
>>> evenNumbers
{8, 10, 2, 4, 6}
>>>
>>> 100  in  evenNumbers
False
>>>
>>> 2  in  evenNumbers
True
```

x not in s : Check whether element is in set or not

‘not in’ operator is opposite of ‘in’ operator, return true if the element is not in set, else false.

```
>>>  evenNumbers
{8, 10, 2, 4, 6}
>>>
>>> 10  not  in  evenNumbers
False
>>>
>>> 100  not  in  evenNumbers
True
```

isdisjoint(other)
Return true if two sets are disjoint, else false. Two sets are said to be disjoint if they have no element in common.

```
>>>  evenNumbers
{8, 10, 2, 4, 6}
>>>
>>>  evenNumbers . isdisjoint({ 1 ,  3 ,  5 ,  7 })
True
>>>
>>>  evenNumbers . isdisjoint({ 1 ,  3 ,  5 ,  7 ,  8 })
False
```

issubset(other)
Return true, if this set is subset of other, else false.

```
>>>  evenNumbers
{8, 10, 2, 4, 6}
>>>
>>>  evenNumbers . issubset({ 2 ,  4 })
False
>>>
```

```
>>> evenNumbers . issubset({ 2 , 4 , 6 , 8 , 10 , 12 })
True
```

set <= other
Return true if every element in the set is in other.

set < other
Return true, if the set is proper subset of other, that is, set >= other and set != other.

```
>>> evenNumbers
{8, 10, 2, 4, 6}
>>>
>>> evenNumbers <= { 2 , 4 , 6 , 8 , 10 }
True
>>> evenNumbers <= { 2 , 4 , 6 , 8 , 10 , 12 }
True
>>>
>>> evenNumbers < { 2 , 4 , 6 , 8 , 10 }
False
>>> evenNumbers < { 2 , 4 , 6 , 8 , 10 , 12 }
True
```

Union of two sets
union(other, ...)
'set | other | ...'

```
>>> evenNumbers = { 2 , 4 , 6 , 8 , 10 }
>>> oddNumbers = { 1 , 3 , 5 , 7 , 9 }
>>> result = evenNumbers | oddNumbers
>>> result
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Intersection of two sets

intersection(other, ...)
'set & other & ...'

```
>>> evenNumbers = { 2 , 4 , 6 , 8 , 10 }
>>> powersOf2 = { 1 , 2 , 4 , 8 , 16 }
>>> result = evenNumbers & powersOf2
>>> result
{8, 2, 4}
```

Difference between two sets
difference(other, ...)
'set - other - ...'
Return a new set with elements in the set that are not in the others.

```
>>> evenNumbers
{8, 10, 2, 4, 6}
>>> powersOf2
{16, 8, 2, 4, 1}
>>> evenNumbers - powersOf2
{10, 6}
>>> powersOf2 - evenNumbers
{16, 1}
```

Symmetric difference between two sets
symmetric_difference(other)
set ^ other
If A and B are two sets, then Simmetric difference between A and B is A^B = (A-B) union (B-A)

```
>>> evenNumbers
{8, 10, 2, 4, 6}
>>> powersOf2
{16, 8, 2, 4, 1}
>>> evenNumbers - powersOf2
{10, 6}
```

```
>>> powersOf2 - evenNumbers
{16, 1}
>>>
>>> evenNumbers ^ powersOf2
{1, 6, 10, 16}
```

Copy elements of set
‘copy’ function return a new set with a shallow copy of s.

```
>>> evenNumbers
{8, 10, 2, 4, 6}
>>> temp = evenNumbers . copy()
>>> temp
{8, 10, 2, 4, 6}
```

Update the set
update(other, ...)
set |= other | ...

Update the set by adding elements from other sets.

```
>>> evenNumbers
{8, 10, 2, 4, 6}
>>> oddNumbers
{9, 3, 5, 1, 7}
>>> powersOf2
{16, 8, 2, 4, 1}
>>>
>>> evenNumbers . update(oddNumbers, powersOf2)
>>> evenNumbers
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16}
```

Intersection of all sets
intersection_update(other, ...)
set &= other & ...

Update the set, keeping only elements found in it and all others.

```
>>> numbers
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16}
>>> oddNumbers
{9, 3, 5, 1, 7}
>>> powersOf2
{16, 8, 2, 4, 1}
>>> numbers . intersection_update(oddNumbers, powersOf2)
>>> numbers
{1}
```

Difference update
difference_update(other, ...)
set -= other | ...
Update the set, removing elements found in others.

```
>>> oddNumbers
{9, 3, 5, 1, 7}
>>> powersOf2
{16, 8, 2, 4, 1}
>>> oddNumbers . difference_update(powersOf2)
>>> oddNumbers
{9, 3, 5, 7}
```

Symmetric difference update
symmetric_difference_update
set ^= other
Update the set, keeping only elements found in either set, but not in both.

```
>>> oddNumbers
```

```
{9, 3, 5, 7}
>>> powersOf2
{16, 8, 2, 4, 1}
>>> oddNumbers . symmetric_difference_update(powersOf2)
>>> oddNumbers
{1, 2, 3, 4, 5, 7, 8, 9, 16}
```

Add element to the set
‘add’ method is used to add element to set.

```
>>> temp
{2, 3, 5, 7}
>>>
>>> temp . add( 11 )
>>> temp . add( 13 )
>>>
>>> temp
{2, 3, 5, 7, 11, 13}
```

Remove an element from set
‘remove’ method is used to remove element from set.

```
>>> temp
{2, 3, 5, 7, 11, 13}
>>>
>>> temp . remove( 2 )
>>> temp
{3, 5, 7, 11, 13}
>>>
>>> temp . remove( 11 )
>>> temp
{3, 5, 7, 13}
```

Throws KeyError, if element is not in the set.

```
>>> temp.remove(100)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 100
```

Remove arbitrary element from set
'pop()' is used to remove and return an arbitrary element from the set. Throws KeyError, if the set is empty.

```
>>>  temp
{5, 7, 13}
>>>  temp . pop()
5
>>>
>>>  temp . pop()
7
>>>  temp . pop()
13
>>>  temp . pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError : 'pop from an empty set'
```

Remove all elements from set
'clear' method is used to remove all elements from set.

```
>>>  powersOf2
{16, 8, 2, 4, 1}
>>>
>>>  powersOf2 . clear()
>>>  powersOf2
```

set()

Python: Find intersection and union of lists

Following program finds

1. Unique elements in list
2. Intersection of two lists
3. Union of two lists.

test.py

```
def unique(a):
    return list(set(a))

def intersect(a, b):
    return list(set(a) & set(b))

def union(a, b):
    return list(set(a) | set(b))

if __name__ == "__main__":
    a = [2, 4, 6, 8, 10, 10, 8, 6]
    b = [2, 4, 8, 16, 1, 3, 5, 2, 4]
    print(unique(a))
    print(intersect(a, b))
    print(union(a, b))
```

```
$ python3 test.py
[8, 2, 10, 4, 6]
[8, 2, 4]
[1, 2, 3, 4, 5, 6, 8, 10, 16]
```

Python: Dictionaries

Dictionaries are used to store key-value pairs, these are similar to associative arrays, map in other languages.

```
>>> employees = { 1 :{ "Hari JJTam" ,  "Delhi" },  2 :{ "Prithi" ,  "Nair" },  3 :
{ "Mohan" ,  "Shekkappa" }}
>>> employees[ 1 ]
{'Hari JJTam', 'Delhi'}
>>>
>>> employees[ 2 ]
{'Prithi', 'Nair'}
>>>
>>> employees[ 3 ]
{'Shekkappa', 'Mohan'}
```

As shown in above example, employees is a dictionary, which maps employee id to their names (first and last names).

Id	Name
1	{"Hari JJTam", "Delhi"}
2	{"Prithi", "Nair"}
3	{"Mohan", "Shekkappa"}

Dictionaries are indexed by keys, keys are immutable types. You can use strings, numbers, any other immutable type as keys. Tuples can be used as keys if they contain only strings, numbers, or tuples.

Building dictionary using dict constructor

*class dict(**kwarg)*
*class dict(mapping, **kwarg)*
*class dict(iterable, **kwarg)*

Above constructors are used to define a dictionary.

```
>>> emps = dict([( 1 ,  "Pradeep" ), ( 2 ,  "Srinath" ), ( 3 ,  "Thushar" )])
>>> emps
{1: 'Pradeep', 2: 'Srinath', 3: 'Thushar'}
>>> a = dict(one =1 , two =2 , three =3 )
>>> a
{'two': 2, 'three': 3, 'one': 1}
>>> b = dict({ 'three' : 3 , 'one' : 1 , 'two' : 2 })
>>> b
{'two': 2, 'three': 3, 'one': 1}
```

Get number of items in the dictionary
'len' method is used to get number of items in the dictionary.

```
>>> employees
{1: {'Hari JJTam', 'Delhi'}, 2: {'Prithi', 'Nair'}, 3: {'Shekkappa', 'Mohan'}}
>>>
>>> len(employees)
3
```

Get the value associated with key
D[key] is used to get the value associated with key.

```
>>> employees[ 1 ]
{'Hari JJTam', 'Delhi'}
>>> employees[ 2 ]
{'Prithi', 'Nair'}
>>> employees[ 3 ]
{'Shekkappa', 'Mohan'}
```

Throws KeyError, if key is not in dictionary.

```
>>> employees[ 5 ]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError : 5
```

Override the value associated with key
d[key] = value
Above statement overrides the value associated with key, if key not exist, it creates new item.

```
>>> employees
{1: {'Hari JJTam', 'Delhi'}, 2: {'Prithi', 'Nair'}, 3: {'Shekkappa', 'Mohan'}}
>>> employees[ 4 ] = { "Ranganath" , "Thippisetty" }
>>> employees[ 1 ] = { "Phalgun" , "Garimella" }
>>>
>>> employees
{1: {'Garimella', 'Phalgun'}, 2: {'Prithi', 'Nair'}, 3: {'Shekkappa',
'Mohan'}, 4: {'Ranganath', 'Thippisetty'}}
```

Remove an item associated with key
del d[key]
‘del’ statement is used to delete an item associate with key (item means <key, value> pair).

```
>>> employees
{1: {'Garimella', 'Phalgun'}, 2: {'Prithi', 'Nair'}, 3: {'Shekkappa',
'Mohan'}, 4: {'Ranganath', 'Thippisetty'}}
>>> del employees[ 2 ]
>>>
>>> employees
{1: {'Garimella', 'Phalgun'}, 3: {'Shekkappa', 'Mohan'}, 4: {'Ranganath',
'Thippisetty'}}
```

Throws KeyError, if key is not in dictionary.

```
>>> del employees[ 2 ]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError : 2
```

Check for key existence
key in d
'in' operator return true, if key is in dictionary, else false.

```
>>> employees
{1: {'Garimella', 'Phalgun'}, 3: {'Shekkappa', 'Mohan'}, 4: {'Ranganath', 'Thippisetty'}}
>>>
>>> 2 in employees
False
>>>
>>> 3 in employees
True
```

Check for key non-existence
key not in d
'not in' operator return true, if key not in dictionary, else false.

```
>>> employees
{1: {'Garimella', 'Phalgun'}, 3: {'Shekkappa', 'Mohan'}, 4: {'Ranganath', 'Thippisetty'}}
>>>
>>> 2 not in employees
True
>>>
>>> 3 not in employees
False
```

Get an iterator over dictionary
iter(d)

‘iter’ function returns an iterator over keys of the dictionary.

```
>>> for  key  in  iter(employees):
...      print (key, employees[key])
...
1 {'Garimella', 'Phalgun'}
3 {'Shekkappa', 'Mohan'}
4 {'Ranganath', 'Thippisetty'}
```

Remove all items from dictionary

‘clear’ function is used to remove all items from dictionary.

```
>>>  employees
{1: {'Garimella', 'Phalgun'}, 3: {'Shekkappa', 'Mohan'}, 4: {'Ranganath',
'Thippisetty'}}
>>>
>>>  employees . clear()
>>>  employees
{}
```

Copy dictionary

‘copy()’ method is used to get shallow copy of this dictionary.

```
>>>  employees  = { 1 : { 'Garimella' ,  'Phalgun' },  3 : { 'Shekkappa' ,
'Mohan' },  4 : { 'Ranganath' ,  'Thippisetty' }}
>>>  employees
{1: {'Garimella', 'Phalgun'}, 3: {'Shekkappa', 'Mohan'}, 4: {'Ranganath',
'Thippisetty'}}
>>>
>>>  emps  =  employees . copy()
>>>  emps
{1: {'Garimella', 'Phalgun'}, 3: {'Shekkappa', 'Mohan'}, 4: {'Ranganath',
'Thippisetty'}}
```

Create new dictionary from keys of given dictionary
fromkeys(seq[, value])

Create a new dictionary with keys from seq and values set to value.

```
>>> employees
{1: {'Garimella', 'Phalgun'}, 3: {'Shekkappa', 'Mohan'}, 4: {'Ranganath',
'Thippisetty'}}
>>> emps = employees . fromkeys([ 1 , 2 , 3 ],  "default" )
>>> emps
{1: 'default', 2: 'default', 3: 'default'}
```

Get the value associated with this key
get(key[, default])
'get' method is used to get the value associated with given key. If key don't exist, then it return the default value. If default is not given, it defaults to None, so that this method never raises a KeyError.

```
>>> employees
{1: {'Garimella', 'Phalgun'}, 3: {'Shekkappa', 'Mohan'}, 4: {'Ranganath',
'Thippisetty'}}
>>> employees . get( 1 )
{'Garimella', 'Phalgun'}
>>>
>>> employees . get( 5 , { "Kiran Kumar" ,  "Darsi" })
{'Kiran Kumar', 'Darsi'}
```

Get all the items in dictionary
'items' method is used to get all the items in the dictionary.

```
>>> employees . items()
dict_items([(1, {'Garimella', 'Phalgun'}), (3, {'Shekkappa', 'Mohan'}), (4,
{'Ranganath', 'Thippisetty'})])
```

Get all the keys from dictionary

'keys()' method is used to get all the keys from dictionary.

```
>>> employees . keys()
dict_keys([1, 3, 4])
```

pop(key[, default])

If the key is in dictionary, pop method removes and return the value associated with this key. If key don't present, it returns the default value. If the default value not specified, it throws KeyError.

```
>>> employees
{1: {'Garimella', 'Phalgun'}, 3: {'Shekkappa', 'Mohan'}, 4: {'Ranganath',
'Thippisetty'}}
>>>
>>> employees . pop( 1 )
{'Garimella', 'Phalgun'}
>>>
>>> employees . pop( 1 , { "dummy" })
{'dummy'}
>>>
>>> employees . pop( 1 )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError : 1
```

Remove arbitrary item from dictionary

'popitem()' method is used to remove a random item from dictionary. If dictionary is empty, it throws KeyError.

```
>>> employees
{3: {'Shekkappa', 'Mohan'}, 4: {'Ranganath', 'Thippisetty'}}
>>>
>>> employees . popitem()
(3, {'Shekkappa', 'Mohan'})
```

```
>>>
>>> employees . popitem()
(4, {'Ranganath', 'Thippisetty'})
>>>
>>> employees . popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError : 'popitem(): dictionary is empty'
```

setdefault(key[, default])
If the key is in dictionary, setdefault method returns the value associated with it, else set the key to default and return default.

```
>>> employees
{}
>>> employees . setdefault( 1 , { 'Shekkappa' ,  'Mohan' })
{'Shekkappa', 'Mohan'}
>>>
>>> employees
{1: {'Shekkappa', 'Mohan'}}
>>>
>>> employees . setdefault( 1 , { 'Garimella' ,  'Phalgun' })
{'Shekkappa', 'Mohan'}
>>>
>>> employees
{1: {'Shekkappa', 'Mohan'}}
```

Update dictionary from other dictionary
update([other])
‘update’ method is used to update the dictionary with the key/value pairs from other, overwriting existing keys.

```
>>> emps1 = { 1 : { 'Garimella' ,  'Phalgun' },  3 : { 'Shekkappa' ,  'Mohan' },
4 : { 'Ranganath' ,  'Thippisetty' }}
>>> emps2 = { 1 :{ 'Hari JJTam' ,  "Delhi" },  5 : { 'Sudheer' ,  'Ganji' }}
>>>
>>> emps1
{1: {'Garimella', 'Phalgun'}, 3: {'Shekkappa', 'Mohan'}, 4: {'Ranganath',
'Thippisetty'}}
>>>
>>> emps2
{1: {'Hari JJTam', 'Delhi'}, 5: {'Ganji', 'Sudheer'}}
>>>
>>> emps1 . update(emps2)
>>>
>>> emps1
{1: {'Hari JJTam', 'Delhi'}, 3: {'Shekkappa', 'Mohan'}, 4: {'Ranganath',
'Thippisetty'}, 5: {'Ganji', 'Sudheer'}}
>>>
>>> emps2
{1: {'Hari JJTam', 'Delhi'}, 5: {'Ganji', 'Sudheer'}}
```

Get all values from dictionary

‘values’ method is used to get all the values from dictionary.

```
>>> emps1
{1: {'Hari JJTam', 'Delhi'}, 3: {'Shekkappa', 'Mohan'}, 4: {'Ranganath',
'Thippisetty'}, 5: {'Ganji', 'Sudheer'}}
>>>
>>> emps1 . values()
dict_values([{'Hari JJTam', 'Delhi'}, {'Shekkappa', 'Mohan'},
{'Ranganath', 'Thippisetty'}, {'Ganji', 'Sudheer'}])
```

Python modules

Module is a file, which contains definitions. We can define classes, modules, variables etc., in module file and reuse them in other python scripts.

For example, create a file ‘arithmetic.py’ and copy following code.

arithmetic.py

```
def sum(a, b):
        return a+b

def subtract(a, b):
        return a-b
def mul(a,b):
        return a*b
def div(a, b):
        return a/b
```

Open python interpreter, import the module using ‘import’ key word and call the functions defined in module.

```
>>> import  arithmetic
>>>
>>>  arithmetic . sum( 10 ,  20 )
30
>>>  arithmetic . subtract( 10 ,  20 )
-10
>>>  arithmetic . mul( 10 ,  20 )
200
>>>  arithmetic . div( 10 ,  20 )
```

```
0.5
```

How to get the module name

By using the property '__name__', you can get the module name.

```
>>> arithmetic.__name__
'arithmetic
```

Use functions of module as local functions

```
>>> import  arithmetic
>>>  sum = arithmetic . sum
>>>  sub = arithmetic . subtract
>>>  mul = arithmetic . mul
>>>  div = arithmetic . div
>>>
>>>  sum( 10 ,  20 )
30

>>>  mul( 10 ,  20 )
200

>>>  sub( 10 ,  20 )
-10
```

Python: Executable statements in module

A module can contain executable statements. These statements are executed; only the first time the module name is encountered in an import statement.

arithmetic.py

```
print( "Simple api to perform arithmetic operations" )
def sum(a, b):

        return a+b


def subtract(a, b):

        return a-b

def mul(a,b):

        return a*b

def div(a, b):

        return a/b
```

```
>>> import  arithmetic
Simple api to perform arithmetic operations
>>>
>>>  arithmetic . sum( 10 ,  20 )
30
>>> import  arithmetic
>>>
>>>  arithmetic . div( 10 ,  20 )
0.5
```

Observe above snippet, print statement executed only for first import of the module, not for the second import.

Python: import functions from module directly

By using from, import keywords we can import functions from modules directly.

arithmetic.py

```
print( "Simple api to perform arithmetic operations" )
def sum(a, b):

	return a+b


def subtract(a, b):

	return a-b

def mul(a,b):

	return a*b

def div(a, b):

	return a/b
```

```
>>> from  arithmetic  import  sum, subtract, mul, div
>>>
>>>  sum( 10 ,  20 )
30
>>>  subtract( 20 ,  30 )
-10
```

Use the statement ‘from fibo import *’ to import all the functions. This imports all names except those beginning with an underscore (_)

Python: import module in other module

Modules can import other modules using import statement.

arithmetic.py

```
def sum(a, b):
        return a+b

def subtract(a, b):
        return a-b
def mul(a,b):
        return a*b
def div(a, b):
        return a/b
```

main.py

```
import  arithmetic

print(arithmetic . sum( 10 ,  20 ))
print(arithmetic . subtract( 10 ,  20 ))
print(arithmetic . mul( 10 ,  20 ))
print(arithmetic . div( 10 ,  20 ))
```

```
$ python3 main.py
30
-10
200
0.5
```

Note

It is not necessary to put all import statements in the beginning of module. You can import whenever required.

Python command line arguments

Python scripts accept any number of arguments from command line. This option facilitates us to configure Application at the time of running.

How to pass command line arguments
pyhon scriptname arg1 arg2 …argN

By using ‘sys’ module we can access command line arguments.

sys.argv[0] contains file name
sys.argv[1] contains first command line argument, sys.argv[2] contains second command line argument etc.,

arithmetic.py
```
def sum(a, b):

	return a+b

def subtract(a, b):

	return a-b
def mul(a,b):

	return a*b
def div(a, b):

	return a/b
```

main.py
```
import  arithmetic

if __name__ == "__main__":
```

```
import sys
a = int(sys.argv[1])
b = int(sys.argv[2])

print(sys.argv[0])
print("a = ", a, "b = ", b)
print(arithmetic.sum(a,b))
print(arithmetic.subtract(a,b))
print(arithmetic.mul(a,b))
print(arithmetic.div(a,b))
```

```
$ python3 main.py 10 11
main.py
a =  10 b =  11
21
-1
110
0.9090909090909091

$ python3 main.py 8 13
main.py
a =  8 b =  13
21
-5
104
0.6153846153846154
```

Python: File handling

In this post and subsequent posts, I am going to explain how to open, read and write to file.

How to open file
To read data from a file (or) to write data to a file, we need a reference object that points to file. ‘open’ method is used to get a file object.

open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
'open' function opens file in specific mode and return corresponding file object. Throws OSError, if it is unable to open a file.

Parameter	Description
file	Full path of the file to be opened.
mode	Specifies the mode, in which the file is opened. By default file opens in read mode.
buffering	0: Switch off the buffer (only allowed in binary mode) 1: Line buffering (only usable in text mode) >1: Specify the size of buffer in bytes. If you don't specify any value, by default buffering works like below. a. Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device’s “block size” and falling back on io.DEFAULT_BUFFER_SIZE. b. “Interactive” text files use line buffering.

encoding	Type of encoding used to encode/decode a file. This value should be used in text mode. if encoding is not specified the encoding used is platform dependent.
errors	Specifies how encoding and decoding errors are to be handled. This cannot be used in binary mode
newline	controls how universal newlines mode works. A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention '\n', the Windows convention '\r\n', and the old Macintosh convention '\r'.
closefd	If closefd is False and a file descriptor rather than a filename was given, the underlying file descriptor will be kept open when the file is closed. If a filename is given closefd must be True (the default) otherwise an error will be raised

Following are different modes that you can use while opening a file.

Mode	**Description**
'r'	open for reading
'w'	open for writing, truncating the file first
'x'	open for exclusive creation, failing if the file already exists
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)

For example, data.txt contains following data.

data.txt

```
First line
Second line
Third line

Fourth line
```

```
>>> f = open( "/Users/hariJJTam_Delhi/data.txt" )
>>>
>>> f . read()
'First line\nSecond line\nThird line\nFourth line\n'
```

Python: Reading files

Following methods are used to read file contents.

ı. read
ɔ. readline
:. readlines

read(size)

Reads 'size' bytes of data and returns it as a string or bytes object. If size is negative (or) omitted, all the file contents are returned. If the end of the file has been reached, f.read() will return an empty string ('').

```
>>> f = open( "/Users/hariJJTam_Delhi/data.txt" )
>>> f . read()
'First line\nSecond line\nThird line\nFourth line\n'
```

readline

Read single line from file. A newline character (\n) is left at the end of the string. Return an empty string, if the file reaches to end.

```
>>> f = open( "/Users/hariJJTam_Delhi/data.txt" )
>>> f . readline()
'First line\n'

>>> f . readline()
'Second line\n'

>>> f . readline()
'Third line\n'

>>> f . readline()
'Fourth line\n'

>>> f . readline()
''
```

readlines

'readlines' method is used to read all the lines of a file as list.

```
>>>  data = f . readlines()
>>>  data
['First line\n', 'Second line\n', 'Third line\n', 'Fourth line\n']
```

Reading a file using for loop

For reading lines from a file, you can loop over the file object.

```
>>>  f = open( "/Users/hariJJTam_Delhi/data.txt" )
>>> for  line  in  f:
...       print (line)
...
First line

Second line

Third line

Fourth line
```

As you observe output, new line is printed after every line. You can get rid of this by passing ‘end’ parameter to print method.

```
>>>  f = open( "/Users/hariJJTam_Delhi/data.txt" )
>>> for  line  in  f:
...       print (line, end ='' )
...
First line
Second line
Third line
Fourth line
```

Python: Write data to file

Python provides following methods to write data to a file.

write (): Used to write a fixed sequence of characters to a file. Return number of characters written to file.

writelines(): writelines can write a list of strings.

```
>>> f = open( "/Users/hariJJTam_Delhi/data.txt" , 'w' )
>>>
>>> f . write( "Hello, How are you\n" )
19
>>> f . write( "I am fine, How is u\n" )
20
>>> data = [ "first line\n" , "second line\n" ]
>>> f . writelines(data)
>>> f . close()
>>>
>>> f = open( "/Users/hariJJTam_Delhi/data.txt" )
>>> for line in f:
...     print (line, end ='' )
...
Hello, How are you
I am fine, How is u
first line
second line
```

‘write’ method only accepts string argument. If you try to insert non-string data, you will get TypeError.

```
>>> f = open( "/Users/hariJJTam_Delhi/data.txt" , 'w' )
>>> f . write( 123 )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError : write() argument must be str, not int
```

To write something other than a string, it needs to be converted to a string first. By using ‘str’ function, you can convert non string data to string.

```
>>> a =123.45
>>> temp = str(a)
>>> temp
'123.45'
>>>
>>> f = open( "/Users/hariJJTam_Delhi/data.txt" ,  'w' )
>>> f . write(temp)
6
>>>
>>> f . close()
>>>
>>> f = open( "/Users/hariJJTam_Delhi/data.txt" )
>>> f . read()
'123.45'
```

Python: classes and objects

Class is a blue print to create objects.

Syntax

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

'class' keyword is used to define a class. You can instantiate any number of objects from a class.

Syntax

```
objName = new ClassName(arguments)
```

test.py

```
class Employee:
 """ Employee class """
noOfEmployees=0  # Class level variable

 def __init__(self, id, firstName, lastName):
  self.id = id
  self.firstName = firstName
  self.lastName = lastName
  Employee.noOfEmployees = Employee.noOfEmployees + 1
 def displayEmployee(self):
  print(self.id, self.firstName, self.lastName)

emp1 = Employee( 1 , "Hari JJTam" , "Delhi" )
print( "Total Employees" , Employee . noOfEmployees)

emp2 = Employee( 2 , "PTR" , "Nayan" )
print( "Total Employees" , Employee . noOfEmployees)

emp3 = Employee( 3 , "Sankalp" , "Dubey" )
print( "Total Employees" , Employee . noOfEmployees)

emp1.displayEmployee()
emp2.displayEmployee()
emp3.displayEmployee()
```

```
$ python3 test.py
Total Employees 1
Total Employees 2
Total Employees 3
1 Hari JJTam Delhi
2 PTR Nayan
3 Sankalp Dubey
```

__init__(arguments)
__init__ is a special function called constructor used to initialize objects. In Employee class, __init__ method is used to initialize id, firstName, lastName to an object at the time of creation.

noOfEmployees=0

‘noOfEmployees’ is a class variable, shared by all the objects. Class variable are accessed using Class name like ClassName.variableName, ‘Employee.noOfEmployees’ is used to access the class variable noOfEmployees’.

Instance variables
Instance variables have values unique to an object. Usually these are defined in __init__ method. Employee class has 3 instance variables id, firstName, lastName.

The first parameter of any method in a class must be self. This parameter is required even if the function does not use it. ‘self’ is used to refer current object.

Python: class: built in class attributes

Following are the built in class attributes.

Attribute	Description
__dict__	Dictionary contains class namespace.
__doc__	Points to documentation string of a class
__name__	Class name
__module__	Module name in which the class is defined
__bases__	Tuple representing the base classes of this class

test.py

```
class Employee:
 """ Blue print for all employees """
noOfEmployees=0  # Class level variable

 def __init__(self, id, firstName, lastName):
  self.id = id
  self.firstName = firstName
  self.lastName = lastName
  Employee.noOfEmployees = Employee.noOfEmployees + 1

 def displayEmployee(self):
  print(self.id, self.firstName, self.lastName)

print ( "Employee.__doc__:" , Employee . __doc__)
print ( "Employee.__name__:" , Employee . __name__)
print ( "Employee.__module__:" , Employee . __module__)
print ( "Employee.__bases__:" , Employee . __bases__)
print ( "Employee.__dict__:" , Employee . __dict__)
```

```
$ python3 test.py
Employee.__doc__:  Blue print for all employees
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: (<class 'object'>,)
Employee.__dict__: {'__module__': '__main__', '__dict__': <attribute
'__dict__' of 'Employee' objects>, 'displayEmployee': <function
Employee.displayEmployee at 0x1006dd7b8>, '__init__': <function
Employee.__init__ at 0x1006dd730>, 'noOfEmployees': 0, '__weakref__':
<attribute '__weakref__' of 'Employee' objects>, '__doc__': ' Blue print
for all employees '}
```

Python: __init__ method

'__init__' is a special function called automatically, whenever you are created an object (It is same like constructor in C++, Java).

Employee.py

```
class Employee:
 """ Blue print for all employees """
noOfEmployees=0  # Class level variable

 def __init__(self, id, firstName, lastName):
  print("Inside Employee constructor")
  self.id = id
  self.firstName = firstName
  self.lastName = lastName
  Employee.noOfEmployees = Employee.noOfEmployees + 1

 def displayEmployee(self):
  print(self.id, self.firstName, self.lastName)

emp1  =  Employee( 1 ,  "Hari JJTam" ,  "Delhi" )
print( "Total Employees" , Employee . noOfEmployees)

emp1 . displayEmployee()
```

```
$ python Employee.py
Inside Employee constructor
('Total Employees', 1)
```

```
(1, 'Hari JJTam', 'Delhi')
```

Observe the output, the message ‘Inside Employee constructor
‘ is printed first, even though program don’t call the method __init__ explicitly. It is because, whenever you create an object, python calls its __init__ method internally.

Python: Overloading __init__ method

If you are familiar with Java, you can overload constructors like below.

```
public class Employee {

private int id;

private String firstName, lastName;

Employee() {
  id = -1;

  this.firstName = "Nil";

  this.lastName = "Nil";

}

Employee(int id, String firstName) {

  this(id, firstName, firstName);

}

Employee(int id, String firstName, String lastName) {

  this.id = id;

  this.firstName = firstName;

  this.lastName = lastName;

}

 ....

 ....

}
```

In case of Python you have to write a single constructor that handles all cases, using either default arguments or type or capability tests.

For example,

Overloading __init__ using None
Employee.py

```
class Employee:
 """ Blue print for all employees """
noOfEmployees=0  # Class level variable

 def __init__(self, id=None, firstName=None, lastName=None):
  print("Inside Employee constructor")
  if(id is None and firstName is None and lastName is None):
   self.id = -1
   self.firstName = firstName
   self.lastName = lastName
  else:
   if(firstName is None):
    self.firstName = self.lastName = "Nil"
   elif(lastName is None):
    self.firstName = firstName
    self.lastName = firstName
   else:
    self.firstName = firstName
    self.lastName = lastName
   self.id = id

 def displayEmployee(self):
  print(self.id, self.firstName, self.lastName)
```

```
emp1 = Employee(1, "Hari JJTam", "Delhi")
emp2 = Employee(2)
emp3 = Employee(3, "PTR")
emp4 = Employee()

emp1.displayEmployee()
emp2.displayEmployee()
emp3.displayEmployee()
emp4.displayEmployee()
```

```
$ python Employee.py
Inside Employee constructor
Inside Employee constructor
Inside Employee constructor
Inside Employee constructor
(1, 'Hari JJTam', 'Delhi')
(2, 'Nil', 'Nil')
(3, 'PTR', 'PTR')
(-1, None, None)
```

Overloading __init__ using default arguments
Employee.py

```
class Employee:
 """ Blue print for all employees """
noOfEmployees=0  # Class level variable

 def __init__(self, id=-1, firstName="Nil", lastName="Nil"):
  self.id = -1
  self.firstName = firstName
  self.lastName = lastName

 def displayEmployee(self):
  print(self.id, self.firstName, self.lastName)

emp1 = Employee(id=1, firstName="Hari JJTam", lastName="Delhi")
emp2 = Employee(id=2)
emp3 = Employee(id=3, firstName="PTR")
emp4 = Employee()

emp1.displayEmployee()
emp2.displayEmployee()
emp3.displayEmployee()
emp4.displayEmployee()
```

```
$ python Employee.py
(-1, 'Hari JJTam', 'Delhi')
(-1, 'Nil', 'Nil')
(-1, 'PTR', 'Nil')
(-1, 'Nil', 'Nil')
```

Python: Class Vs Instance variables

Class variables are associated with class and available to all the instances (objects) of the class, where as instance variables are unique to objects, used to uniquely identify the object.

Employee.py

```
class Employee:
 """ Blue print for all employees """
 # Class level variables
noOfEmployees=0
organization="abcde corporation"

 def __init__(self, id=-1, firstName="Nil", lastName="Nil"):
  self.id = -1
  self.firstName = firstName
  self.lastName = lastName
  Employee.noOfEmployees+=1

 def displayEmployee(self):
  print(self.id, self.firstName, self.lastName)

emp1 = Employee(id=1, firstName="Hari JJTam", lastName="Delhi")
emp1.displayEmployee()
print( "Total Employees : " , Employee . noOfEmployees)
print( "Organization : " , Employee . organization)

emp2 = Employee(id=3, firstName="PTR")
```

```
emp2.displayEmployee()
print( "Total Employees : " , Employee . noOfEmployees)
print( "Organization : " , Employee . organization)
```

```
$ python3 Employee.py
-1 Hari JJTam Delhi
Total Employees :  1
Organization :  abcde corporation
-1 PTR Nil
Total Employees :  2
Organization :  abcde corporation
```

As you observe Employee.py noOfEmployees, organization are class variables, are available to all the instances. Class variables are accessed using ClassName followed by dot followed by variable name.

Employee.noOfEmployees: is used to access class variable noOfEmployees.
Employee.organization: is used to access class variable organization.

Python: Inheritance

Inheritance is the concept of re usability. Object of one class can get the properties and methods of object of another class by using inheritance.

Syntax

```
class DerivedClassName(BaseClassName1, BaseClassName2 ... BaseClassNameN):
    <statement-1>
    .
    .
    .
    <statement-N>
```

inheritance.py

```
class Parent:
 def printLastName(self):
  print("Delhi")

 def printPermAddress(self):
  print("State : Andhra Pradesh")
  print("Country : India")

class Child(Parent):
 def printName(self):
  print("Hari JJTam Delhi")
```

```
child1  =  Child()

child1.printName()
child1.printLastName()
child1.printPermAddress()
```

```
$ python3 inheritance.py
Hari JJTam Delhi
Delhi
State : Andhra Pradesh
Country : India
```

Observe above program, two classes parent and child are defined. Parent class defines two methods printLastName, printPermAddress.
Child class defines one method printName. Child class inheriting the methods printLastName, printPermAddress from parent class.

Overriding the methods of Parent class
You can override the properties, methods of parent class in child class. For example, following application overrides ‘printPermAddress’ method of Parent class.

inheritance.py

```
class Parent:
 def printLastName(self):
  print("Delhi")

 def printPermAddress(self):
  print("State : Andhra Pradesh")
  print("Country : India")
```

```
class Child(Parent):
 def printName(self):
  print("Hari JJTam")

 def printPermAddress(self):
  print("City : Bangalore")
  print("State : Karnataka")
  print("Country : India")

child1  =  Child()

child1.printName()
child1.printLastName()
child1.printPermAddress()
```

```
$ python3 inheritance.py
Hari JJTam
Delhi
City : Bangalore
State : Karnataka
Country : India
```

Python: Multiple inheritance

A class can inherit properties, methods from more than one class.

MultipleInheritance.py

```
class A:
 def printA(self):
  print("I am in A")

class B:
 def printB(self):
  print("I am in B")

class C:
 def printC(self):
  print("I am in C")

class D(A, B, C):
 def printD(self):
  print("I am in D")

obj=D()
obj.printA()
obj.printB()
obj.printC()
obj.printD()
```

```
$ python3 MultipleInheritance.py
I am in A
I am in B
I am in C
I am in D
```

Python: Exceptions

Exception is an event that disrupts the normal flow of execution. Even though statements in your program are syntactically correct, they may cause an error.

```
>>> 10/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError : division by zero
>>>
>>>  tempList = []
>>>  tempList[ 20 ]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError : list index out of range
```

Observer above snippet, '10/0' causes ZeroDivisionError. When program tries to access 20th element of tempList it causes IndexError.

Python: Handling Exceptions
Python provide keywords try, except to handle exceptions.

test.py
```
while True:
 try:
  x = int(input("Enter input "))
  print(x)
  break;
 except ValueError:
  print("Please enter valid number")
```

```
$ python3 test.py
Enter input an
Please enter valid number
Enter input ana
Please enter valid number
Enter input ptr
Please enter valid number
Enter input 10
10
```

How try and except block work?
The statements in try block executed first. If no exception occurs, the except clauses are skipped and execution of the try statement is finished. If any exception occurs during the execution of try block, the rest of the try clause is skipped.

‘try’ block followed by number of except clauses, if exception thrown in try cause matches to any exception followed by except clause, then particular except clause is executed.

If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops by throwing exception message.

Try block can be followed by multiple except clauses

test.py

```
while True:
 try:
  x = int(input("Enter divisor "))
  y = int(input("Enter dividend "))
  print(x/y)
  break;
 except ValueError:
  print("Please enter valid number")
 except ZeroDivisionError:
  print("y should be non zero")
```

```
$ python3 test.py
Enter divisor 2
Enter dividend 0
y should be non zero
Enter divisor 4
Enter dividend 0
y should be non zero
Enter divisor 4
Enter dividend 2
2.0
```

Handling multiple exceptions in single except clause
An except clause can catch more than one exception. For example ‘except (ValueError, ZeroDivisionError)’ can handle both ValueError and ZeroDivisionError.

test.py
```
while True:
 try:
  x = int(input("Enter divisor "))
  y = int(input("Enter dividend "))
  print(x/y)
  break;
 except (ValueError, ZeroDivisionError):
  print("Please enter valid number (or) y should be greater than 0")
```

```
$ python3 test.py
Enter divisor 2
Enter dividend 0
Please enter valid number (or) y should be greater than 0
Enter divisor aa
Please enter valid number (or) y should be greater than 0
Enter divisor 2
Enter dividend 4
0.5
```

Last except clause can omit exception name. It is used as global exception handler.

test.py

```
while True:
 try:
  tempList=[]
  print(tempList[10])
  break
 except ValueError:
  print("Please enter valid number")
 except ZeroDivisionError:
  print("y should be non zero")
 except Exception as inst:
  print("Global handler", inst)
  break
```

```
$ python3 test.py
Global handler list index out of range
```

try…except…else clause

'try…except' statement can have optional else clause, it is followed by except clause. If try block doesn't throw any exception, else clause will be executed.

test.py

```
while True:
 try:
  x = int(input("Enter divisor "))
  y = int(input("Enter dividend "))
  print(x/y)
 except ValueError:
  print("Please enter valid number")
 except ZeroDivisionError:
  print("y should be non zero")
 else:
  print("Program executed successfully")
  break
```

```
$ python3 test.py
Enter divisor 4
Enter dividend 2
2.0
Program executed successfully
```

Exception argument
Whenever an exception occurs, it is associated with a variable called exception argument.

test.py

```
while True:
 try:
  x = int(input("Enter divisor "))
  y = int(input("Enter dividend "))
  print(x/y)
 except ValueError as inst:
  print(inst)
 except ZeroDivisionError as inst:
  print(inst)
 else:
  print("Program executed successfully")
  break
```

```
$ python3 test.py
Enter divisor qwerty
invalid literal for int() with base 10: 'qwerty'
Enter divisor 4
Enter dividend 0
division by zero
Enter divisor 2
Enter dividend 4
0.5
Program executed successfully
```

The except clause can specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in instance.args.

test.py

```
while True:
 try:
  x = int(input("Enter divisor "))
  y = int(input("Enter dividend "))

  if y==0:
   raise Exception(x, y)
  print("x/y = ",x/y)
  break
 except Exception as inst:
  arg1, arg2 = inst.args
  print("arg1=", arg1)
  print("arg2=", arg2)
```

```
$ python3 test.py
Enter divisor 2
Enter dividend 0
arg1= 2
arg2= 0
Enter divisor 2
Enter dividend 4
x/y =  0.5
```

If an exception has arguments associated with it, those are printed as last part of the exception message.

test.py

```
while True:
 try:
  x = int(input("Enter divisor "))
  y = int(input("Enter dividend "))

  if y==0:
   raise Exception(x, y)
  print("x/y = ",x/y)
  break
 except Exception as inst:
  print(inst)
```

```
$ python3 test.py
Enter divisor 2
Enter dividend 0
(2, 0)
Enter divisor 2
Enter dividend 4
x/y =  0.5
```

Python: Raising Exceptions

‘raise’ statement is used to raise an exception.

Syntax

raise_stmt ::= "raise" [expression ["from" expression]]

If no expressions are present, raise re-raises the last exception that was active in the current scope. If no exception is active in the current scope, a RuntimeError exception is raised indicating that this is an error.

test.py

```
while True:
 try:
  dividend = int(input("Enter divisor "))
  divisor = int(input("Enter dividend "))
  if divisor==0:
   raise Exception("divisor shouldn't be zero")

  print(dividend/divisor)
  break
 except Exception as inst:
  print(inst)
```

```
$ python3 test.py
Enter divisor 2
Enter dividend 0
divisor shouldn't be zero
Enter divisor 23
Enter dividend 43
0.5348837209302325
```

raise Exception("divisor shouldn't be zero")

‘raise’ statement throws Exception, when divisor is 0.

Python: User defined exceptions

You can create custom exceptions by extending the class Exception.

test.py

```
class PasswordException(Exception):
 def __init__(self, value):
  self.value = value

 def __str__(self):
  return repr(self.value)

password = input( "Enter password\n" )

if(len(password) < 8):
 raise PasswordException("Minimum password lenght should be 8 characters")
```

```
$ python3 test.py
Enter password
abcde
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    raise PasswordException("Minimum password lenght should be 8 characters")
__main__.PasswordException : 'Minimum password lenght should be 8 characters'
```

In this example, the default __init__() of Exception has been overridden. The new behaviour simply creates the value attribute. This replaces the default behaviour of creating the args attribute.

Python global keyword

‘global’ keyword is used to create/change a global variables (You can declare functions, classes etc. also) from local context.

test.py

```
def function1():
 global data
data="Hello World"

def function2():
 print(data)

function1()
function2()
```

```
$ python3 test.py
Hello World
```

Observe ‘test.py’, even though data is declared in function1, it is accessed by function2. It is because, data is declared in global scope.

Python: Get module name

Every module in python has predefined attribute '__name__'. If the module is being run standalone by the user, then the module name is '__main__', else it gives you full module name.

arithmetic.py

```
def sum(a,b):
    return a+b

print( "Module name : " ,__name__)
```

Run above program you will get following output.

```
Module name :  __main__
```

test.py

```
import  arithmetic

print( "Module name : " ,arithmetic . __name__)
```

Run test.py, you will get following output.

```
Module name :  arithmetic
Module name :  arithmetic
```

Observe the output, module name printed twice. A module can contain executable statements. These statements are executed; only the first time the module name is encountered in an import statement. So executable statement in arithmetic.py is also executed.

Update arithmetic.py like below and rerun test.py.

```
def sum(a,b):
    return a+b
```

```
if(__name__=="__main__"):
    print("This program is run by itslef")
else:
    print("This module called by other module")
```

You will get following output.

```
This module called by other module
Module name :  arithmetic
```

Python: Get type of variable

You can get the type of a variable using ‘type()’ function or __class__ property.

```
>>> data = [ 1 , 2 , 3 , 4 ]
>>> type(data)
<class 'list'>

>>> data . __class__
<class 'list'>

>>>
>>> data = { 1 : "hari" , 2 : "JJTam" }
>>> type(data)
<class 'dict'>

>>> data . __class__
<class 'dict'>
```

Checking the type using if statement

```
data = { 1 : "hari" , 2 : "JJTam" }

#Approach 1

if(data.__class__.__name__ == 'dict' ):

    print("data is of type dictionary")

else:

    print("data is not dictionary type")


#Approach 2

if(type(data).__name__ == 'dict'):

    print("data is of type dictionary")

else:

    print("data is not dictionary type")
```

```
#Approach 3
if type(data)==type(dict()):
    print("data is of type dictionary")
else:
    print("data is not dictionary type")
```

Run above program, you will get following output.

```
data is of type dictionary
data is of type dictionary
data is of type dictionary
```

Check whether an object is instance of class or not
'isinstance(object, classinfo)' method is used to check whether an object is instance of given class or not. Return true if the object argument is an instance of the classinfo argument, or of a subclass thereof, else false.

```
data = { 1 : "hari" ,  2 : "JJTam" }

class Employee:
    def __init__(self, id, firstName, lastName):
     self.id = id
     self.firstName = firstName
     self.lastName = lastName

emp = Employee( 1 ,  "Hari" ,  "JJTam" )

print(isinstance(emp, Employee))
print(isinstance(emp, dict))
print(isinstance(data, dict))
```

Run above program, you will get following output.

```
True
```

False
True

Python: Generate random numbers

Python provides random module to generate random numbers, By using random.seed method you can initialize the random generator.

random.seed(a=None, version=2)
Used to initialize random generator. If you don’t specify value for a, then system time is used by default. ‘a’ can be int, string.

If version=1, then hash of ‘a’ is used for initialization, if version=2 a str, bytes, or bytearray object gets converted to an int and all of its bits are used.

Following methods are used to generate random number and shuffle list randomly in python.

Method	**Description**
random.getrandbits(k)	Returns an integer with k random bits.
random. randrange(a, b)	Returns an integer in the range [a, b).
random.randint(a, b)	Return a random integer N such that a <= N <= b. It is equivalent to random. randrange(a, b+1)
random.choice(seq)	Return a random element from the non-empty sequence seq. If seq is empty, raises IndexError.
random.shuffle(list)	Shuffles a list in-place, i.e. permutes it randomly
random.random()	Return the next random floating point number in the range [0.0, 1.0).
random.uniform(a, b)	Return a random floating point number N such that a <= N <= b

	for a <= b and b <= N <= a for b < a.
random.triangular(low, high, mode)	Return a random floating point number N such that low <= N <= high and with the specified mode between those bounds. The low and high bounds default to zero and one. The mode argument defaults to the midpoint between the bounds, giving a symmetric distribution.

Note

os.urandom(n): Return a string of n random bytes suitable for cryptographic use.

```
>>> import  random
>>>
>>>  random . getrandbits( 20 )
700469

>>>  random . getrandbits( 10 )
216

>>>
>>>  random . randrange( -10000 ,  234567 )
160279

>>>  random . randrange( -10000 ,  234567 )
163448

>>>  random . randrange( -10000 ,  234567 )
141854

>>>  random . randrange( -10000 ,  234567 )
113743

>>>
>>>
>>>  random . choice(list(range( 10 ,  1000 )))
```

```
448
>>> random . choice(list(range( 10 ,  1000 )))
459
>>>
>>> random . choice([ 2 ,  4 ,  6 ,  8 ,  10 ,  12 ,  14 ,  16 ,  18 ,  20 ])
12
>>> random . choice([ 2 ,  4 ,  6 ,  8 ,  10 ,  12 ,  14 ,  16 ,  18 ,  20 ])
10
>>> random . choice([ 2 ,  4 ,  6 ,  8 ,  10 ,  12 ,  14 ,  16 ,  18 ,  20 ])
20
>>> random . choice([ 2 ,  4 ,  6 ,  8 ,  10 ,  12 ,  14 ,  16 ,  18 ,  20 ])
8
>>>
>>> list = [ 2 ,  4 ,  6 ,  8 ,  10 ]
>>> random . shuffle(list)
>>> list
[8, 4, 2, 6, 10]
>>>
>>> random . uniform( 0 ,  10 )
4.136505113937443
>>> random . uniform( 0 ,  10 )
7.627400639276029
>>> random . uniform( 0 ,  10 )
8.812919559259498
```