Chapter – 12 Functions

Contents

Chapter – 12 Functions

Function များသည် Python programming language တွင် အရေးပါသည့် အစိတ်အပိုင်းများ ဖြစ်ကြသည်။ ပရိုဂရမ်၏ နေရာ အတော်များများတွင် မကြာခဏ အသုံးပြုရမည့် code တချို့ကို function အဖြစ် ရေးထားပြီး လိုအပ်လျှင် ခေါ်သုံးနိုင်သည်။ တစ်ခါ ရေးထားရုံဖြင့် ထပ်ခါထပ်ခါ သုံးနိုင်သောကြောင့် အလွန် အသုံးဝင်သည်။ Function ကို သင်၏ program ထဲတွင် ရှိနေမည့် mini-program များ အဖြစ် ယူဆနိုင်သည်။

တိတိကျကျ သတ်မှတ်ထားသည့် အလုပ်(specific task) တစ်ခုကို လုပ်ဆောင်ပေးရန် အတွက်သာ function တစ်ခုကို တည်ဆောက်ရေးသားသင့်သည်။ ရေရှည်တွင် လွယ်ကူစွာ debug လုပ်ရန် ရည်ရွယ်၍ code များကို တစ်ခုစီ ဖြစ်အောင်(modular ဖြစ်အောင်) ရေးသားသင့်သည်။

Function အကြောင်းလေ့လာသည့်အခါ Python တွင်ရှိသည် function အမျိုးအစား(function type)များ အကြောင်းကို သိထားရမည်။ Global variable နှင့် local variable ကို ခွဲခြားပြီး သိရပါမည်။

Function များသည် function class ၏ instance များ ဖြစ်ကြသည်။

12. Function ဆိုတာ

Function ဆိုတာ မိမိအလိုရှိသည့် အလုပ်တစ်ခု ပြီးမြောက်ရန်အတွက် ရေးထားသည့် code အစုအဝေး (block of code)ဖြစ်သည်။ အစုအဝေး (block of code)ဆိုသည်မှာ ကုဒ်လိုင်းလေးတွေ (lines of code) ကို အတူတကွ စုစည်းထားသည့် အစုအဝေးဖြစ်သည်။ Function ကို ပရိုဂရမ်အငယ်စားလေး(mini-programs)တွေ အဖြစ်လည်းယူဆလို့ရသည်။

တိကျသည့် အလုပ်တစ်ခုကို လုပ်ရန်အတွက် function တစ်ခု အဖြစ် ခွဲထုတ်၍ ရေးခြင်းသည့် အလေ့ အကျင့်ကောင်း တစ်ခု ဖြစ်သည်။ (it is better to write functions that only perform one specific task.)။ ထိုသို့ အလုပ်တစ်ခုအတွက် function တစ်ခုအဖြစ် ခွဲထုတ်၍ ရေးခြင်း ကြောင့် ရေရှည်တွင် အမှားရှာရန် ပို၍ လွယ်ကူ(easier to debug)သည်။ ပြုပြင်ထိန်းသိမ်းမှုတွေ လုပ်ရန်လည်း အဆင် ပြေသည်။ Input ထည့်ပေးရန် လိုအပ်သည့် function များ ရှိနိုင်သလို input မလိုအပ်သည့် function များလည်း ရှိနိုင်သည်။ return ပြန်ပေးသည့် function များ ရှိနိုင်သလို return ပြန်မပေးသည့် function များလည်း ရှိနိုင်သည်။

12.1 Function Basics

Python မှာ မိမိ အလုပ်ချင်သည့် အလုပ်တွေကို အလွယ်တကူလုပ်လို့ ရအောင် ကိုယ်ပိုင် function လေးတွေ ရေးလို့ ရသည်။ ထပ်ခါထပ်ခါ လုပ်ရမည့် အလုပ်မျိုးတွေအတွက် function တစ်ခု ရေးထားလိုက်ရုံနဲ့ လိုအပ်သည့် အခါတိုင်း အသုံးပြုနိုင်သည်။ တခြား program language တွေမှာ function ကို subroutines သို့မဟုတ် procedures လို့ခေါ်သည်။

Function ဆိုတာ တစ်ခါထက် ပို run ဘို့ လိုတဲ့ statement တွေကို စုပြီး ရေးထားတာပါ။(groups a set of statements so they can be run more than once in a program)။ အဲဒီ statement အစုအဝေးကို နာမည် ပေးလိုက်ပြီး နောင်လိုအပ်လို့ ခေါ်သုံးချင်ရင် နာမည်လေး ခေါ်လိုက်ရုံနဲ့ သူက အလုပ်လုပ်ပေးမှာပါ။

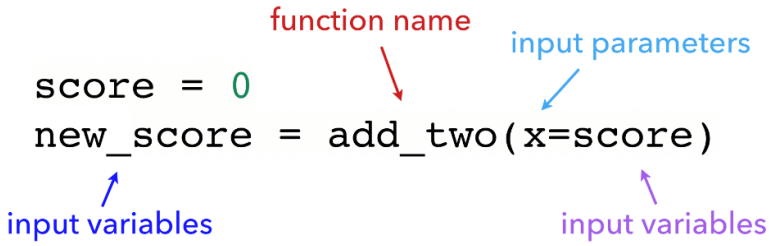
ခဏခဏ လုပ်ရမဲ့ ကိစ္စတွေကို function နဲ့ မရေးထားရင် လုပ်စရာရှိတိုင်း statement တွေ ခဏခဏ ရေးနေရင် ပရိုဂရမ်က မလိုအပ်ဘဲ စာကြောင်းတွေ များနေပါလိမ့်မည်။ ပြင်ဘို့ ၊ ထိန်းသိမ်းဘို့ ၊ အမှား စစ်ဘို့လည်း ခက်ခဲပြီး အချိန်ကုန်သည်။ ကိုယ်ပိုင် function လေးတွေ ရေးထားခြင်းကြောင့် အလုပ်ရှုပ်တော်တော် သက်သာသည်။ Function တစ်နေရာတွင် ပြင်ရုံဖြင့် function ခေါ်သုံးထားသည့်နေရာများ အားလုံး ပြင်ပြီးသား ဖြစ်သည်။ (only one copy to update in the function)

ပရိုဂရမ်တစ်ခုကို သက်ဆိုင်ရာ function လေးတွေ ပိုင်းခြားပြီးတော့ ရေးရတာမို့လည်း ပိုအဆင်ပြေသည်။ (reusable and easier to code)။

Python တွင် **def** (define)နည်း နှင့် **lambda** နည်း ဆိုပြီးတော့ function လုပ်နိုင်တဲ့ နည်း(၂)မျိုး ရှိသည်။ Global နှင့် nonlocal ဆိုပြီးတော့ visibility ကို စီမံနိုင်တဲ့ နည်း(၂)နည်း ရှိပါတယ်။ Function ခေါ်သုံးသူ(caller) ဆီကို ပြန်ပြီးတော့ ပို့ပေးဘို့ return နှင့် yield ဆိုပြီးတော့ ပြန်ပို့ပေးသည့်နည်း (၂)မျိုး ရှိသည်။

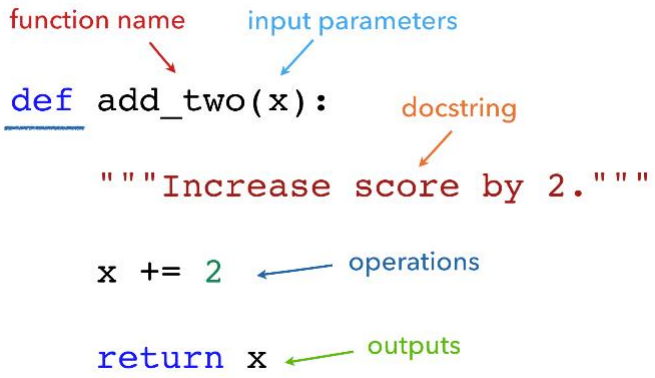
Built-in function နှင့် မိမိဖာသာရေးတဲ့(user define) function ဆိုပြီးတော့ (၂)မျိုး ရှိပါတယ်။ Built-in function အမျိုးမျိုးကို သုံးတတ်ပြီးသားပါ။ ဥပမာ string တစ်ရဲ့ အရှည်(length)ကို ရှာဘို့ len built-in function ကို အသုံးပြုတတ်ပြီးသား ဖြစ်နေပါပြီ။ built-in function နှင့် user define function အလုပ်လုပ်ပုံက အတူတူပါပဲ။ ရေးတဲ့သူပဲ မတူတာပါ။

Function ထဲမှာ ပါတဲ့ statement တွေကို expression လို့ ခေါ်ပါတယ်။ function ထဲကို တန်ဖိုးတွေ ထည့်ပေးမယ်။ အဲလိုတန်ဖိုးတွေ ထည့်ပေးတာကို value pass လုပ်တယ်လို့ ပြောပါတယ်။ Function က ရေးထားတဲ့ expression တွေအတိုင်း အလုပ်လုပ်ပေးပြီးတော့ အဖြေ ပြန်ပေးပါတယ်။ အဲလို function ကနေ ပြန်ပေးတာတွေကို return result ကို return လုပ်တယ်လို့ ပြောပါတယ်။ (return result)



12.2 Function ဘယ်လိုရေးမလဲ။

def သည် define ၏ အတိုခေါက် ဖြစ်သည်။ Define သည် function တစ်ခုတည်ဆောက်ရန်အတွက် define လုပ်ခြင်း ဖြစ်သည်။ Function တစ်ခု တည်ဆောက်တော့မည်ဟု အသိပေးခြင်း ဖြစ်သည်။ တစ်နည်းအားဖြင့် def အတွင်း၌ ရှိသည့် expression များသည် function အတွက် ဖြစ်သည်။ C language က function အလုပ်လုပ်ပုံနှင့် Python function အလုပ်လုပ်ပုံ မတူညီကြပါဘူး။ တော်တော် ကွဲပြားသည်။



12.3 Python Function အလုပ်လုပ်ပုံ

(i) def is executable code

Python function မှာ ပါသည်ပထမဆုံး စာကြောင်း “def” က executable code ဖြစ်သည်။ Python function က def ဟာ executable statement တစ်ခုဖြစ်ပေမဲ့ def စာကြောင်း (statement) မရောက်ခင် အချိန်အထိ ဘာအလုပ်မှ မလုပ်ပါဘူး။ **def** ဆိုသည့်ကုဒ် အထိမရောက်သေးခင် အချိန်တွင် function မတည်ရှိ သေးပါ။(function does not exist until Python reaches and runs the def)

if statement တွေထဲမှာ while loop တွေထဲမှာ ၊ တခြားသော def တွေ ထဲမှာ နောင် Python function ဖြစ်တဲ့ def statement ကို ထပ်ထည့်လို့ (nest လုပ်လို့) ရသည်။ လက်ခံပါတယ်။ ပုံမှန်အားဖြင့် def statement ကို module file ထဲမှာ ထည့်ရေးထားလေ့ရှိသည်။ Module file ကို imprt လုပ်လိုက်တဲ့ အချိန်ကြမှ module file ထဲက def statement က ထ run ပြီး function တစ်ခု အဖြစ် တည်ဆောက် လိုက်ခြင်း ဖြစ်သည်။ (run to generate functions)

(ii) def creates an object and assigns it to a name

Python က ကုဒ် တစ်ကြောင်းချင်းစီ run နေရာက **def** statement နေရာ ရောက်မှ function object တစ်ခုပြီးလုပ်ပြီး function နာမည်ပေးလိုက်တာပါ။ (assigns it to the function’s name)။ Function နာမည်က သက်ဆိုင်သည့် function object ကို ကိုယ်စားပြုသည်။ Function object မှာ arbitrary user-defined attributes တွေ ရှိနိုင်သည်။ Function က lambda expression တွေ ထုတ်ပေးနိုင်သည်။

(iii) return sends a result object back to the caller.

Function တစ်ခုကို ခေါ်ပြီး အလုပ်လုပ်ခိုင်းလိုက်ပြီးနောက် ခေါ်ပြီးတော့ အလုပ်လုပ်ခိုင်းတဲ့သူက function အလုပ်ပြီးမြောက်အောင်လုပ်ပြီး အဖြေပြန်ပေးသည့် အချိန်အထိ ဘာမှ မလုပ်ဘဲ စောင့်နေပါတယ်။ Function လုပ်ခိုင်းသည့်အလုပ်ကို ပြီးမြောက်အောင် ဆောင်ရွက်ပြီး ခိုင်းတဲ့သူ(caller)ဆီကို return statement ပြန်ပို့ ပေးသည်။ return value သည် ခိုင်းတဲ့သူ(caller) လိုအပ်သည့် အဖြေများ ရလဒ်များ ဖြစ်ကြသည်။ ခိုင်းတဲ့သူ(caller)က return value တောင်းပြီး return ပြန်ပေးရန် function ပြန်ပေးရန် ရန် မရှိသည့်အခါတွင် Non ပြန်ပေးသည်။

Global Variable နှင့် Local Variable

Function တစ်ခုအတွင်းမှာ variable တွေကို define လုပ်ထားရင် ၄င်းတို့ကို local variable ဟု ခေါ်သည်။ Local variable များသည် ထို function အတွင်းမှာပဲ တည်ရှိသည်။

Module အဆင့်မှာ ရှိတဲ့ variable တွေ သို့မဟုတ် ပရိုဂရမ်၏ အစနေရာတွင် ရေးထားသည့် variable တွေက global variable များ ဖြစ်ကြသည်။ ပရိုဂရမ်တစ်ခုလုံးနှင့် သက်ဆိုင်သည်။ ပရိုဂရမ်ရပ်သည့် အခါမှာ global variable များ၏ သက်တမ်းကုန်ဆုံးသည်။ Function နှင့် သက်ဆိုင်သည့် global variable ၊ local variable ၊ global keyword နှင့် nonlocal word အကြောင်းများကို Chapter-13 scopes အခန်းတွင် အသေးစိတ် ဖတ်ရှုနိုင်သည်။

Python တွင် function ထဲသို့ argumentများကို ထည့်ပေး(pass လုပ်)သည့်အခါ object reference ကို အသုံးပြုသည်။ Object reference ဖြင့်ထည့်ပေးသည့်နည်းကို assignment ဟုလည်း ခေါ်သည်။ Function နှင့် function ကို ခေါ်ခိုင်းသူ(caller)တို့သည် object reference ကို အသုံးပြု၍ object များကို အတူတကွ access လုပ်သည်။

ဘာမှ မပြောလျှင်(သတ်မှတ်ချက် တစ်ခုခုထည့်မရေးလျှင်) function အတွင်းသို့ argument များကို နေရာ အစီအစဉ်တကျ(position အတိုင်း)ထည့်ပေးသည်။ တစ်နည်းအားဖြင့် keyword များ, variable နာမည်များ မပါလျှင် position argument အဖြစ် သတ်မှတ်သည်။

Argument များ return value များ နှင့် variables များကို အသုံးမပြုခင် ကြိုတင် ကြေငြာ(declare) ရန် မလိုအပ်ပါ။ ထို့ကြောင့် function တစ်ခုကို object type မျိုးစုံအတွက် အသုံးပြုနိုင်သည်။

def Statements

def statement သည် function တစ်ခုကို တည်ဆောက်(create) ပေးသည့် အလုပ်နှင့် ထို function ကို နာမည်တပ်ပေးသည့် အလုပ်(၂)ခုကို လုပ်ပေးသည်။ **Python** တွင် အရာရာတိုင်းသည် object ဖြစ်သော ကြောင့် function object တစ်ခုကို တည်ဆောက်(create) ပေးသည်။ သတိပြုရန်အချက်မှ def statement သည် function အတွင်း object များကို တည်ဆောက်ပေး(create)ခြင်း မဟုတ်ပါ။ Function ကိုသာ တည်ဆောက်(create)ပေးခြင်း ဖြစ်သည်။

```
def name(arg1, arg2,... argN):
       statements
```

def statement သည် header line နှင့် block of statement တို့ပါဝင်သည့် compound statement တစ်ခု ဖြစ်သည်။ Statement block ကို function's body ဟုလည်း ခေါ်သည်။ Statement block ကို indentation ဖြင့် ရေးရသည်။ Function ကို ခေါ်သုံးသည့်(call)အခါတိုင်း statement block များကို တစ်ကြောင်းပြီး တစ်ကြောင်း execute လုပ်သည်။

def statement တွင် function နာမည် ထည့်ပေးရသည်။ လက်သည်းကွင်း(parentheses()) ထဲတွင် argument သို့မဟုတ် parameter များကို ထည့်ပေးရသည်။ Argument သို့မဟုတ် parameter အသုံး မလိုသည့် အခါ ထည့်ပေးရန် မလိုပေ။

Function body အောက်ဆုံးတွင် return statement ကို ရေးလေ့ရှိသည်။ **return** value ပြန်ပေးရန် မလိုသည့် function များတွင် return statement မပါဝင်ပါ။ Function body တွင် return function

တိုင်းတွင် မဖြစ်မနေပါရမည့် statement မဟုတ်ပါ။ Optional statement သာ ဖြစ်သည်။ **return** statement ရောက်ပြီးနောက် function ၏ အလုပ် ပြီးဆုံးပြီ ဖြစ်သည်။

def Executes at Runtime

Python **def** statement သည် true executable statement တစ်ခုဖြစ်သည်။ ခေါ်သုံးသည့်အချိန် (runtime) ၌ execute လုပ်သည်။

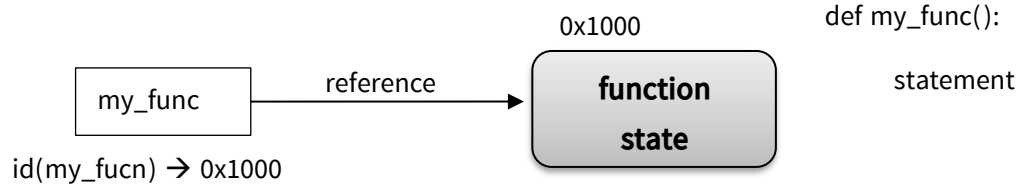
12.4 Function အားသာချက်များ

- ကုဒ်အစုအဝေးတစ်ခုကို တစ်ခါရေးရုံ ဖြင့် အကြိမ်ကြိမ် အခါခါ အသုံးပြုနိုင်သည်။
- Function တစ်ခုရှိ ကုဒ်များသည် မှန်ကန်၍ လုပ်ချင်သည့် အလုပ်ကို အောင်မြင်စွာ လုပ်ပေးနိုင်ပါက နောင်ဘယ်တော့မှ စိုးရိမ်ကြောင့်ကြစိုက်ရန် မလိုတော့ပါ။ ကြီးမားသည့် ပရိုဂရမ်များအတွက် အရမည်း အဆင်ပြေသည်။
- တစ်စုံတစ်ခု ပြောင်းလဲ(modify လုပ်)လိုပါက ပရိုဂရမ်နေရာ အနှံအပြားမှ ကုဒ်များကို လိုက်ပြင် နေရန် မလိုပါ။ Function အတွင်းရှိ ကုဒ် တချို့ကိုသာ ပြင်လိုက်ရုံဖြင့် အဆင်ပြေသည်။

Python တွင် အဓိကအားဖြင့် function (၃)မျိုးရှိသည်။ (three main types of functions)

- Built-in function
- User-defined function နှင့်
- Anonymous function တို့ ဖြစ်သည်။

Function ၏ memory address



Function သည် object တစ်ခုဖြစ်သည်။ Object တိုင်းတွင် memory address ကိုယ်စီ ရှိသောကြောင့် function တွင်လည်း memory address ရှိသည်။ Function ထဲသို့ မည်သည့် object ကို မဆို parameter အဖြစ် သို့မဟုတ် argument အဖြစ် ထည့်ပေး(pass လုပ်)နိုင်သည်။

ထိုအပြင် function တစ်ခုအတွင်းသို့ တခြား function များကိုလည်း ထည့်ပေးနိုင်သည်။(Decotrator အခန်းတွင် အသေးစိတ် ရှင်းပြထားသည်။) Function မှလည်း မည်သည့် object ကို မဆို ပြန်ထုတ်ပေး(return လုပ်)နိုင်သည်။ Function တစ်ခု တခြားသော function များကို return အဖြစ် ပြန်ပေးနိုင်သည်။

12.2 Built-In Functions

Python interpreter ထဲတွင် built-in function များစွာ ပါဝင်သည်။ Built-in function များသည် အဆင်သင့် အသုံးပြုနိုင်အောင် Python ထဲတွင် ထည့်သွင်းထားပြီးသား ဖြစ်သည်။ Built-in function များကို

import လုပ်ရန်မလိုဘဲ ပရိုဂရမ်၏ မည်သည့်နေရာတွင်မဆို အသုံးပြုနိုင်သည်။ **print( )** ၊ **map( )** ၊ **len ()** စသည်တို့သည် built-in function များ ဖြစ်ကြသည်။ **print()** built-in function ကို အသုံးပြုပြီးတော့ ပရိုဂရမ်မှ ထုတ်ပေးသည့် ရလဒ်(result) များကို ဖော်ပြနိုင်သည်။ ဥပမာ - print("Hello world")

တခြား program language များတွင် function များကို subroutine သို့မဟုတ် procedure ဟုခေါ်သည်။ Built-in function နှင့် မိမိဖာသာရေးထားသည့်(user defined) function ဆိုပြီးတော့ (၂)မျိုး ရှိသည်။ Built-in function အမျိုးမျိုးကို သုံးတတ်ပြီးသား ဖြစ်သည်။ ဥပမာ string တစ်ခု၏ အရှည်(length)ကို ရှာရန် အတွက် **len()** built-in function ကို အသုံးပြုတတ်ပြီးသား ဖြစ်သည်။ **Built-in function နှင့် user define function အလုပ်လုပ်ပုံ တူညီကြသည်။ Code ရေးသူ မတူသောကြောင့် နာမည် ကွဲပြားသွားခြင်း ဖြစ်သည်။**

Function ထဲတွင် ပါရှိသည့် statement များကို expression ဟုခေါ်သည်။ Function ထဲသို့ လိုအပ်သည့် တန်ဖိုးများထည့်ပေးရသည်။ ထိုသို့တန်ဖိုးများ ထည့်ပေးခြင်းကို value pass လုပ်သည် ဟုခေါ်သည်။ Function ထဲတွင် ရေးထားသည့် expression/statement များအတိုင်း အလုပ်လုပ်ပေးပြီ ရလဒ်(result) ပြန်ပေးသည်။ ထိုသို့ function မှ ရလဒ်(result) ပြန်ပေးခြင်းကို return ဟုခေါ်သည်။

အောက်တွင် built-in function များဖြစ်ကြသည့် **print()** function, **len()** function, **type()** function နှင့် **id( )** function တို့ကို ပြထားသည်။

```
print("Python")                    # Built in print()funtion
print(len("Python"))               # Built in len()funtion
print(type(12))                    # Built in type() funtion
print(id("Python"))                # Built in id()funtion
```

```
Python
6
<class 'int'>
1910261969968
```

Built-in Functions

abs()	**delattr()**	**hash()**	**memoryview()**	**set()**
all()	**dict()**	**help()**	**min()**	**setattr()**
any()	**dir()**	**hex()**	**next()**	**slice()**
ascii()	**divmod()**	**id()**	**object()**	**sorted()**
bin()	**enumerate()**	**input()**	**oct()**	**staticmethod()**
bool()	**eval()**	**int()**	**open()**	**str()**
breakpoint()	**exec()**	**isinstance()**	**ord()**	**sum()**
bytearray()	**filter()**	**issubclass()**	**pow()**	**super()**
bytes()	**float()**	**iter()**	**print()**	**tuple()**

callable()	**format()**	**len()**	**property()**	**type()**
chr()	**frozenset()**	**list()**	**range()**	**vars()**
classmethod()	**getattr()**	**locals()**	**repr()**	**zip()**
compile()	**globals()**	**map()**	**reversed()**	**__import__()**
complex()	**hasattr()**	**max()**	**round()**	

12.3 User-Defined Functions

အသုံးပြုသူများက မိမိတို့၏ သင့်လျော်ရာ အလုပ်အတွက် ရေးထားတဲ့ function များကို တော့ User-Defined Function ဟုခေါ်သည်။ Python function တစ်ခုရဲ့ syntax ကတော့ အောက်ပါအတိုင်း ဖြစ်သည်။

```
def function_name(parameter_one, parameter_two, parameter_n):
        # Logic goes here
        return
```

Function တစ်ခု တည်ဆောက်ရန် သို့မဟုတ် function တစ်ခု define လုပ်ရန် အဆင့်များ

၁။ ပထမဆုံး အကြောင်းမှာ **def** ဆိုသည့် keyword ဖြင့်စတင် ရေးသားရမည်။ ထို့နောက် function နာမည် (name) ပါရမည်။ (Use the def keyword, followed by the function name.)။ Keyword **def** ၏ အဓိပ္ပာယ်မှာ function တစ်ခု တည်ဆောက်တော့မည် define လုပ်တော့မည်ဟု Python ကို အသိပေးလိုက်ခြင်း ဖြစ်သည်။

၂။ လက်သည့်ကွင်း () ထဲမှာ function အတွက် လိုအပ်သည့် parameter များကို ထည့်ပေးရမည်။ ပထမဆုံး **def** နှင့် စပြီး code စာကြောင်း၏အဆုံးမှာ **: (full colon**) နဲ့ အဆုံးသတ်ပေးရပါမည်။ (Add parameters (if any) to the function within the parentheses. End the function definition with a full colon.)

၃။ ဒုတိယအကြောင်းကစပြီး logic တွေ စရေးလို့ ရပါပြီ။ (Write the logic of the function.)

၄။ နောက်ဆုံးမှာတော့ **return** ဆိုသည့် keyword ကို အသုံးပြုပြီးတော့ function ၏ ရလဒ်(result)ကို output အဖြစ် ပြန်ပေးလိမ့်။ Function ၏ result ကို output အဖြစ် ထုတ်ပေးဘို့ မလိုတဲ့အခါမျိုးတွင် “None” ပြန်ပေးသည်။ (return လုပ်သည်)။ return ပြန်ထုတ်ပေးသည် ဘာမှ မရှိပါ ဆိုသည့် အဓိပ္ပာယ် ဖြစ်သည်။

Function ၏ output ပြန်ထုတ်ပေးရန် အတွက် **return** ဆိုသည့် keyword ကို နောက်ဆုံးလိုင်းတွင် ရေးရပါမည်။ နောင်ဆုံးလိုင်းတွင် return မပါက မည်သည့် ရလဒ်(result)ကိုမျှ output အဖြစ် ထုတ်ပေးလိမ့်မည် မဟုတ်ပေ။ ထို့ကြောင့် output ပြန်ထုတ်ပေးရန် မလိုအပ်သည့် function များတွင် return လိုင်းကို ချန်ထား နိုင်သည်။

User-defined function များတွင် ကိုယ်ပိုင် နာမည်များ ရှိရန် လိုအပ်သည်။ မိမိကြိုက်တဲ့ နာမည်ပေးလို့ ရပါသည်။ အဓိပ္ပာယ်ရှိသည့် နာမည်မျိုး၊ အလုပ်ပေးတဲ့ အလုပ်နဲ့ကိုက်ညီတဲ့နာမည်မျိုး ပေးသင့်

ပါသည်။ ဥပမာ- လစဉ် မိုးရေချိန် တွက်ပေးသည့် function ကို calculate_ monthly_average_ rainfall()လို့ နာမည် ပေးသင့်သည်။

Code များကို ကွန်ပျူတာက လက်ခံပြီးခိုင်းသည့် အလုပ်များလုပ်ပေး လိမ့်မည် ဖြစ်သော်လည်း အမှန်တကယ် ဖတ်ရှုရန် လိုအပ်သူများသည် လူသားများ ဖြစ်သောကြောင့် ဖတ်သူများ နားလည် လွယ်အောင် ရေးသင့်သည်။ ကိုယ်ပေးထား function နာမည်တစ်ခုကို မိမိကိုယ်တိုင် ပြန်နားမလည် နိုင်ခြင်းမျိုး မဖြစ်သင့်ပါ။

Parameter ဆိုတာမှာ function ထဲကို ထည့်ပေးရမည့် အချက်အလက်များ ဖြစ်သည်။ Function ထဲကို အချက်အလက်များ ထည့်ပေးခြင်းကို pass လုပ်သည့်ဟု ခေါ်သည်။ Parameter များကို ထည့်ပေး သည့်အခါ parentheses ဆိုသည့် လက်သည်းကွင်း()ထဲမှာ **, (commas)** ခံပြီးတော့ ထည့်ပေး ရပါမည်။ Parameter အရေ အတွက် လိုသလောက် ထည့်နိုင်ပါသည်။ Parameter အရေအတွက် ကန့်သတ်ချက် မရှိပါ။ List ၊ tuple စသည်တို့ ကိုလည်း parameter အဖြစ် ထည့်ပေး နိုင်သည်။

အောက်တွင် ဥပမာအဖြစ် function တစ်ခု တည်ဆောက်ပါမည်။ ထို့နောက် ထို function တည်ရှိသည် မရှိ သည်ကို စစ်ကြည့်ပါမည်။ Function တစ်ခု define လုပ်ပြီး logic statement များ/ expression များ အထည့်သေးခင် pass ဆိုသည့် keyword ကို ခဏထည့်ထားနိုင်သည်။ pass ဆိုတာက ဘာ logic မှ မထည့်ထားရသေးလို့ ကျော်သွားပါဆိုသည့်အဓိပ္ပာယ် ဖြစ်သည်။

```
def my_function():                          # Create and assign function
    pass
```

အခုဆိုရင် my_function လို နာမည်ပေးထားသည့် function တစ်ခု တည်ဆောက် ပြီးသွားပါပြီ။ အဲဒီ function လေးက ဘာအလုပ်မှ လုပ်မှာ မဟုတ်သေးပါဘူး။ Function လေး ရှိမရှိ **type()** နဲ့စစ်ကြည့်ရင် function တစ်ခု ဖြစ်ကြောင်း ပြန်ဖြေပါလိမ့်မည်။ အဲဒီ function လေးရှိနေတဲ့ memory location ကိုလည်း ကြည့်နိုင်သည်။

```
type(my_function)
```

```
function
```

```
id(my_function)
```

```
3006791887320
```

အခု Hello! လို့ print ထုတ်ပေးတဲ့ function တစ်ခု တည်ဆောက်ကြရအောင်။

```
def print_greeting():       # print_greeting သည် function နာမည်ဖြစ်သည်။
    print('Hello!')

print_greeting()     # တည်ဆောက်သည့် print_greeting function ကို ခေါ်သုံးသည်။
```

```
Hello!
```

Function ကို ပြန်ခေါ်(call လုပ်)သည့်အခါ လက်သည်းကွင်း() ပါထည့်ရေးရမည်။ Function ကို ပြန်ခေါ်(call လုပ်)သည့်အခါ သတိပြုရန်အချက်မှာ function name တစ်ခုတည်း ရေး၍ မရပါ။ လက်သည်းကွင်း() မပါဘဲ function name တစ်ခုတည်း ရေးလျှင် function object ကိုသာ ဆိုလိုခြင်း ဖြစ်သည်။ Function name ကို လက်သည်းကွင်း() နှင့် တွဲရေးပါမှ function ကို ပြန်ခေါ်(call လုပ်)ခြင်း ဖြစ်သည်။ Function ကို ခေါ်သုံးခြင်း(call လုပ်ခြင်း)ကို invoke the function , run the function စသည်ဖြင့်လည်း သုံးနှုန်း ပြောဆိုလေ့ ရှိသည်။

```
print_greeting          # လက်သည်းကွင်း မပါဘဲရေးလျှင် function object ကိုသာ ပြန်ပေးသည်။
```

```
<function __main__.print_greeting()>
```

12.4 Function တစ်ခုကို ခေါ်သုံးခြင်း (Calling a Function)

def statement သည် function တစ်ခုကို တည်ဆောက်(define)ရုံသာ ဖြစ်သည်။ တစ်စုံတစ်ယောက်က ခေါ်(call)မခိုင်းမချင်း အလုပ်လုပ်လိမ့်မည် မဟုတ်ပေ။ ထို function အလုပ်လုပ်(run)စေရန် call လုပ်ရသည်။

Calling a function ဆိုတာ function ထဲ၌ ရေးထားသည့် statement များကို အလုပ်လုပ်ခိုင်းခြင်း (executing the logic statement that is defined inside of the function) ဖြစ်ပါသည်။ Python interactive prompt မှာ function တစ်ခုကို ခေါ်သုံးလို့ ရသလို code တွေထဲမှာလည်း function တစ်ခုကို ခေါ်သုံးလို့ ရပါသည်။ တခြား function မှ တဆင့် နောက်ထပ် function တစ်ခုကို ခေါ်သုံးလို့လည်း ရသည်။

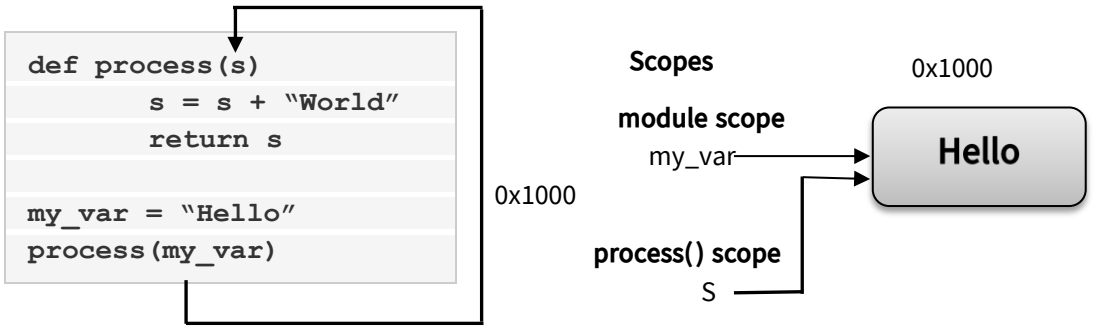
Function တစ်ခုကို ခေါ်သုံးသည့်အခါ argument ဆိုသည့် စကားလုံးတစ်လုံးကို သေသေချာချာ နားလည်သဘောပေါက်ထားရန် လိုအပ်ပါသည်။ Function တည်ဆောက်တုန်းက (define လုပ်တုန်းက) သုံးခဲ့သည့် စကားလုံးသည် parameter ဖြစ်ပါသည်။ Function တစ်ခုကို ခေါ်သုံးတဲ့အခါကြတော့မှ argument ဟု ပြောင်းသုံး လိုက်ပါသည်။ ထည့်ပေးသည့် ဒေတာမှ အတူတူပင် ဖြစ်သော်လည်း အခေါ်အဝေါ် အသုံးအနှုန်း မတူသည်ကို သတိပြုပါ။

12.4.1 Function Argument

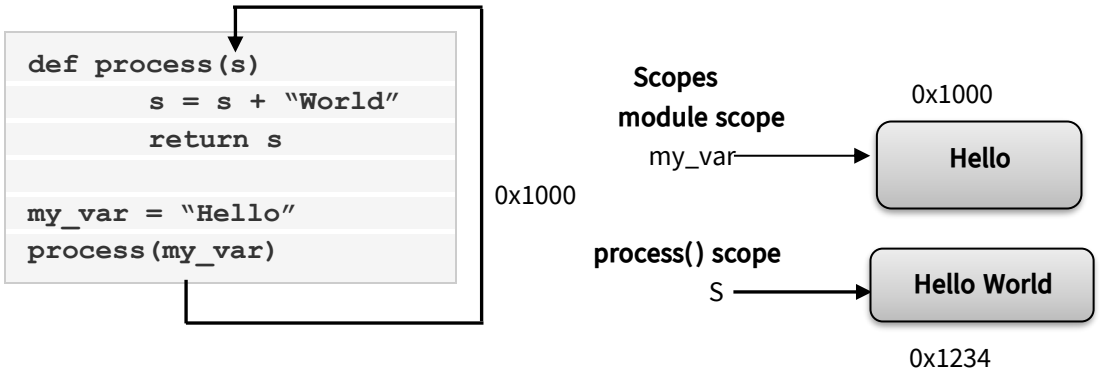
အောက်တွင် function တစ်ခု အတွင်းသို့ argument ထည့်(pass လုပ်)ပုံကို ရှင်းပြထားသည်။

အောက်က process() ဆိုသည့် function တစ်ခု define လုပ်ထားသည်။ **my_var** ဆိုသည့် variable ကို “Hello” ဖြင့် assign လုပ်ထားသည်။ my_var သည် memory ပေါ်က “Hello” သိမ်းဆည်းထားသည့် address(0x1000)ကို point လုပ်သည်။ သို့မဟုတ် refenrece လုပ်သည်။ Variable များတွင် module scope နှင့် process() scope ဟူ၍ Scope (၂)မျိုးရှိသည်။

process() function ကို call လုပ်သည့်အခါ process() ထဲသို့ my_var ဆိုသည့် argument ကို ထည့်(pass လုပ်)သည်။ my_var ၏ refenrece ကို process() ဆိုသည့် function ထဲသို့ ထည့် (passလုပ်) သည်။ သတိပြုရန်အချက်မှာ သည်။ my_var ဆိုသည့် variable ကို process() ထဲသို့ pass လုပ်ခြင်း မဟုတ်ပါ။ my_var ၏ refenrece ကိုသာ pass လုပ်ခြင်း ဖြစ်သည်။ ထိုအခါ process() scope မှ S ထဲတွင် my_var ၏ refenrece ကိုသိမ်းဆည်းလိုက်သည်။ S နှင့် my_var တို့သည် “Hello” ဆိုသည့် object ကို ပြိုင်တူ point လုပ်နေသည်။



`s = s + "World"` လိုင်းကို execute လုပ်ပြီးသည့်အခါ memory က တခြား address(0x1234) တွင် Hello World ဆိုသည့် object တစ်ခုကို ပြုလုပ်လိုက်ပြီး S သည် ထို address ကို point လုပ်သည်။ ထိုနောက် အောက်တွင် ပြထားသည့်အတိုင်း S ၏ refenrece ပြောင်းသွားသည်။ (`s = Hello World`)



my_var ၏ refenrece သည် မပြောင်းလဲဘဲ S ၏ refenrece သာ ပြောင်းလဲသွားသည်။

အောက်က ဥပမာတွင် `times(2, 4)` ကုဒ်သည် function ခေါ်ခြင်းဖြစ်သသည်။ `times` သည် function နာမည်ဖြစ်သည်။ `(2, 4)` သည် function ထဲသို့ ထည့်ပေးသည့် argument ဖြစ်သည်။

```
def times(x, y):                    # Create and assign function
       return x * y                 # Body executed when called
times(2, 4)                         # Arguments in parentheses
```

```
8
```

အထက်က ဥပမာတွင် အဖြေ(returned object)ကို printထုတ်ပေးသည်။ Function မှ ပြန်လာသည့် returned object သို့မဟုတ် returned value ကို variable တစ်ခုနှင့် assign လုပ်လိုပါက အောက်ပါ အတိုင်း ရေးနိုင်သည်။

```
x = times(3.14, 4)              # Save the result object
x
```

```
12.56
```

Parameter X သည် integer သို့မဟုတ် float တစ်မျိုးမျိုးဖြစ်မည်ဟု ယူဆထားသော်လည်း string တစ်ခု ထည့်သောအခါ

```
times('Ni', 4)                    # Functions are "typeless"
```

```
'NiNiNiNi'
```

12.5 Function Return

Function ကို ခေါ်ခိုင်းသူအား ရလဒ်အဖြစ် ပြန်ပေးရန်အတွက် function အတွင်း၌ return statement ပါဝင်သည်။ **return** statement မပါခဲ့သော် function မှ မည်သည့် ရလဒ်ကိုမှ ပြန်ပေး လိမ့်မည် မဟုတ်ပေ။ အောက်တွင် return statement မပါသည့် ကိန်း(၂)လုံးပေါင်းသည့် function တစ်ခုကို ဥပမာ အဖြစ် ဖော်ပြထားသည်။ ကိန်း(၂)လုံးကို ပေါင်းခိုင်းပြီး ပေါင်းလဒ်ကို ဖော်ပြရုံသာ ဖော်ပြပြီး တခြား တစ်နေရာ တွင် အသုံးပြုစရာ မလိုသောကြောင့် return ပြန်မခေါ်ခြင်း ဖြစ်သည်။

```
def summation(first, second):    # summation သည် function နာမည် ဖြစ်သည်။
    total = first + second
    print("The total is" , str(total))

summation(10, 20)                        # summation function ကို ခေါ်သုံးသည်။
```

အထက်က summation function ဥပမာတွင် return statement မပါရှိပါ။ **return** statement သည် function တိုင်းတွင် မရှိမဖြစ် ပါဝင်ရန် မလိုအပ်ပါ။ ရလဒ်(total)ကို print out လုပ်ပုံကို ဖော်ပြလိုသည့် ဥပမာ ဖြစ်သည်။ Function အတွင်း၌လည်း print လုပ်နိုင်ပုံကို ဖော်ပြရန် ရည်ရွယ်သည်။

Function တစ်ခုကို ခေါ်ခိုင်းပြီးနောက် result ကို ရယူပြီး တခြားနေရာတွင် အသုံးပြုလိုသည့်အခါ return statement မဖြစ်မနေ လိုအပ်ပုံကို အောက်တွင် ဥပမာဖြင့် ဖော်ပြထားသည်။

```
def summation(first, second):
    total = first + second
    return total

summation(10, 20)/4                    # summation function ကို ခေါ်သုံးသည်။
```

```
7.5
```

summation function သည် ထည့်ပေးလိုက်သည့် တန်ဖိုး(passed-in value) နှစ်ခုကို ပေါင်းပြီး function return အဖြစ် ရလဒ်(30)ကို ပြန်ပေးသည်။

seq1 နှင့် seq2 နှစ်ခုစလုံးတွင် ဘုံပါဝင်သည့်ကိန်းကိုသာ ယူသည့် ဥပမာ ဖြစ်သည်။

```
def intersect(seq1, seq2):
    res = []                          # empty တစ်ခု ပြုလုပ်သည်။
    for x in seq1:                    # seq1 ကို Scan ဖတ်သည်။
        if x in seq2:                 # နှစ်ခုစလုံးတွင် ပါမပါစစ်သည်။(Common item?)
            res.append(x)             # Add to end
    return res
```

```
seq1 = 1,2,3
seq2 = 2,3,4
intersect(seq1, seq2)
```

```
[2, 3]
```

အထက်က ဥပမာ ဤကဲ့သို့ တစ်ကြောင်းတည်းဖြင့်လည်း ရေးနိုင်သည်။ နောက်ပိုင်းတွင် ရှင်းပြထားသည်။

```
[x for x in seq1 if x in seq2]
```

12.5.1 Non Return Function

Non return function ဆိုသည်မှာ return ပြန်မပေးသည့် function များ ဖြစ်ကြသည်။ တစ်နည်းအားဖြင့် return statement မပါဝင်သည့် function များ ဖြစ်ကြသည်။

```
a = print("Python")
print(a)
```

```
Python
None
```

12.5.2 Return value

Function တစ်ခုတွင် return statement တစ်ခုထက် ပိုပြီးရှိနိုင်သည်။ Function ၏ အောက်ဆုံးတွင် return statement သာ ပုံမှန်အားဖြင့် တွေ့မြင်ရလေ့ရှိသည်။ အပေါင်းကိန်းလား? အနှုတ်ကိန်းလား? ဆုံးဖြတ်ပေးမည့် အောက်က function တွင် condition (၃)ခုအတွက် return statement သုံးခု ပါရှိသည်။

ထည့်ပေးသည့် value သည် အပေါင်းကိန်း(positive) ဖြစ်လျှင် ပထမ return statement သည် အလုပ် လုပ်လိမ့်မည်။ အနှုတ်ကိန်း(Negative) ဖြစ်လျှင် ဒုတိယ return statement သည် အလုပ် လုပ်လိမ့်မည်။ ထည့်ပေးသည့် value သည် သုည(zero) ဖြစ်လျှင် ပထမနှင့် ဒုတိယ return statement နှစ်ခုစလုံး အလုပ် လုပ်လိမ့်မည်မဟုတ်ပေ။ ထိုအခါ တတိယ return statement အလုပ်လုပ် လိမ့်မည်။

```
def getSign(value):
    if value > 0:
        return "Positive", value
    elif value < 0:
        return "Negative", -value
    else:
        return "Zero", value

print("getSign(33.6):", getSign( 33.6 ))
print("getSign(-7):", getSign( -7 ))
print("getSign(0):", getSign( 0 ))
```

```
getSign(33.6): ('Positive', 33.6)
getSign(-7): ('Negative', 7)
getSign(0): ('Zero', 0)
```

Return statement မှ တစ်ခုထက် ပိုများသည့် return value များ ပြန်ပေးသည်အခါ **tuple အဖြစ် return လုပ်သည်**။ Item တစ်ခု ချင်းစီ အဖြစ် ရလိုပါက unpack လုပ်ရသည်။

```
def myFunction(value1, value2):
    total = value1 + value2
    difference = value1 - value2
    product = value1 * value2
    return total, difference, product

values = myFunction(3, 7)          # return value များကို tuple တစ်ခုအဖြစ် ရနိုင်သည်။
print("Results as a tuple:", values)

# တန်ဖိုးတစ်ခုချင်းစီ(individual values)အဖြစ်ရလိုပါက tuple ကို unpack လုပ်ရသည်။
x, y, z = myFunction( 3, 7 )
print("x:", x)
print("y:", y)
print("z:", z)
```

```
Results as a tuple: (10, -4, 21)
x: 10
y: -4
z: 21
```

12.6 Function Arguments

Function တည်ဆောက်(define)နေစဉ် အချိန်တွင် ထည့်ထားသည့် variable များကို parameter ဟု ခေါ်သည်။ Function ခေါ်သုံးသည့်အချိန်တွင် function ထဲရှိ parameter များအတွက် ထည့်ပေးသည့် variable များကို argument ဟုခေါ်သည်။

Arguments and parameters

Parameter ဆိုသည်မှာ function ကို define လုပ်စဉ်အခါက parentheses() ထဲတွင် define လုပ်ပြီး ထည့်ထားသည့် variable ဖြစ်သည်။ Argument ဆိုသည်မှာ function တစ်ခုကို ခေါ်သုံးသည့် အချိန် (calling a function)တွင် function အတွင်းရှိ parameter များ ဆီသို့ ထည့်ပေးမည့် တန်ဖိုးများ(passed value)ဖြစ်သည်။ Pass လုပ်မည့် value ဖြစ်သည်။

```
def print_name(name                    # name သည် parameter ဖြစ်သည်။
    print(name)

print_name('Alex')                     # 'Alex' သည် argument ဖြစ်သည်။
```

```
Alex
```

Python တွင် အသုံးပြု၍ ရသည့် argument တချို့မှာ အောက်ပါအတိုင်း ဖြစ်သည်။

- Required argument
- Keyword argument
- Default argument နှင့်
- A variable number of argument တို့ ဖြစ်သည်။

12.6.1 Mandatory Arguments

တချို့သော function များသည် အနည်းဆုံး argument တစ်ခုကို ထည့်ပေးရန်(pass လုပ်ရန်) လိုအပ်သည်။ ထိုသို့ function တစ် အလုပ်လုပ်ရန်အတွက် မဖြစ်မနေ ထည့်ပေးရသည့် argument များကို mandatory argument ဟုခေါ်သည်။ Mandatory argument မထည့်ပေးပါ function အလုပ်လုပ်လိမ့်မည် မဟုတ်ပေ။ Exception(error) ထုတ်ပေးလိမ့်မည်။

```
def square(number):                    # mandatory argument တစ်ခု လိုအပ်သည်။
    y = number*number
    return y

square(2)
```

```
4
```

```
square()          # မည်သည့် argument မှ မထည့်ပေးသောကြောင့် TypeError ပေးသည်။
```

```
TypeError     Traceback (most recent call last)
TypeError: square() missing 1 required positional argument: 'number'
```

12.6.2 Required Arguments

Function တစ်ခုကို ခေါ်သုံးသည့်အခါ မဖြစ်မနေထည့်ပေးရမည့် argument များရှိသည်။ ထို argument များကို လိုအပ်သည့်(required) argument ဟုခေါ်သည်။ ထို required argument များကို function ထဲသို့ ထည့်ပေးသည့်အခါ အစဉ်အတိုင်း ဖြစ်အောင်(correct order) ထည့်ပေးရမည်။

12.6.3 Positional Arguments

Function တစ်ခုထဲသို့ positional argument များ သော်လည်းကောင်း keyword argument များ လည်းကောင်း ထည့်ပေးနိုင်သည်။ Positional argument သည် နေရာ(position)ကို အထူးပြု၍ ခေါ်ဆိုသည့် နာမည်ဖြစ်သောကြောင့် ထည့်ပေးသည့် argument ၏ ရှေ့နောက်နေရာအစဉ်(order)သည် အဓိကကြသည်။

```
def division(first, second):
    return first/second

quotient = division(10,2)
print(quotient)
```

```
5.0
```

Function အလုပ်လုပ်စေရန် first နှင့် second ဟူ၍ အမည်ပေးထားသည့် argument နှစ်ခုကို ထည့်ပေးရန်(pass လုပ်ရန်) လိုအပ်သည်။ Argument များကို ထည့်ပေး(pass လုပ်)သည့်အခါ သတ်မှတ်

ထားသည့် အတိုင်းအစီအစဉ်တကျ(right order) ထည့်ပေးရန် လိုအပ်သည်။ ရှေ့နောက် အစီအစဉ်မှာ ထည့်လျှင် ရလဒ် မှန်ကန်လိမ့်မည် မဟုတ်ပေ။

```
def add_one(a, b, c):
    print(a+1, b+1 , c+1)

add_one(1, 2, 3)                    # positional argument ဖြစ်သည်။
```

```
2 3 4
```

Positional argument ကို အသုံးပြုသည့်အခါ argument ထည့်ပေးသည့် ပထမနေရာ ဒုတိယနေရာ စသည်တို့ကို သတိပြုရန်လိုသည်။ အောက်တွင် argument နေရာများထည့်သောကြောင့် ရလာဒ်များပုံကို ဥပမာ အဖြစ်ဖော်ပြထားသည်။

```
def describe_person(first_name, last_name, age):
    print("First name: %s" % first_name)
    print("Last name: %s" % last_name)
    print("Age: %d\n" % age)

describe_person('brian', 'kernighan', 71)   # first_name အမှန်
describe_person('kernighan','brian', 71)    #first_name များထည့်ထားသည်။
```

```
First name: Brian
Last name: Kernighan
Age: 71

First name: Kernighan
Last name: Brian
Age: 71
```

Fuction ပထမထည့်ပေးသည့် တန်ဖိုးကို first_name အဖြစ် သတ်မှတ်သည်။

12.6.4 Keyword Arguments

Keyword argument သည် နာမည်(parmeter or argument name)နှင့်တွဲ၍ ထည့်ပေးရသောကြောင့် နေရာအစီအစဉ်(position order)သည် အဓိက မကြေပေ။

```
def add_one(a, b, c):
    print(a+1, b+1 , c+1)

foo(a=1, b=2, c=3)      # keyword arguments
foo(c=3, b=2, a=1)      # Note that the order is not important here
```

```
2 3 4
2 3 4
```

Function ကို ခေါ်ခိုင်းသည့်အခါ argument များကို parameter name ဖြင့် မထည့်ပေးပါက အစီအစဉ်တကျ(right order) ထည့်ပေးရန် လိုအပ်သည်။ အောက်တွင် parameter name မပါဘဲ argument ကို ထည့်ပေးသည်။

အောက်က ဥပမာတွင် argument ကို တမင်တကာ ရှေ့နောက် မှားအောင် ထည့်ပေးထားသည်။ သို့သော် argument name ဖြင့် ထည့်ပေးသောကြောင့် အဖြေမှန်သည်။

```
def division(first, second):
    return first/second

quotient = division(second=2, first=10)
print(quotient)
```

```
5.0
```

Keyword argument များကို နာမည်နှင့်တွဲ၍ သုံးလျှင် ကြိုက်သည့် အစီအစဉ်ဖြင့် ထည့်ပေးနိုင်သောကြောင့် အလွန် အဆင်ပြေသည်။ Keyword argument များကို ထည့်ပေးသည့်အခါ ထည့်ပေးသည့် နေရာ အရှေ့အနောက် အစဉ်သည် အရေးမကြီးပါ။ အောက်တွင် ရှေ့နောက် ကြိုက်သလို ထည့်လို့ရပုံကို ဥပမာ အဖြစ်ဖော်ပြထားသည်။ ရလဒ်မှာ အတူတူပင်ဖြစ်သည်။

```
def describe_person(first_name, last_name, age):
    print("First name: %s" % first_name.title())
    print("Last name: %s" % last_name.title())
    print("Age: %d\n" % age)

describe_person(age=71, first_name='brian', last_name='kernighan')
describe_person(last_name='thompson', age=70, first_name='ken')
describe_person(age=68, last_name='goldberg',first_name='adele')
```

```
First name: Brian
Last name: Kernighan
Age: 71

First name: Ken
Last name: Thompson
Age: 70

First name: Adele
Last name: Goldberg
Age: 68
```

12.6.5 Positional နှင့် Keyword Argument တို့ကို ရောၜ ထည့်ပေးခြင်း

အောက်တွင် လူတစ်ယောက်၏ အချက်အလက်များ ပါသည့် function တစ်ခု တည်ဆောက်သည့် အခါ နာမည် ကလွဲလို တခြားအချက်အလက်များ မရှိနိုင်သည့်အခါမျိုး ရှိနိုင်သည်။ ထိုအခါ အသက် သို့မဟုတ် မွေးသက္ကရာဇ် မသိပါက function parameter မှာ None ဟု assign လုပ်ခဲ့နိုင်သည်။ ကြိုက်နှစ်သက်သည့် language မသိပါကလည်း function parameter မှာ None ဟု assign လုပ်ခဲ့နိုင်သည်။ သေဆုံးသည့်နှစ် မသိပါကလည်း function parameter မှာ None ဟု assign လုပ်ခဲ့နိုင်သည်။

Fuction ကို ခေါ်သည့်အခါ positional argument နှင့် keyword argument ကို ရောၜ အောက် ဥပမာတွင် ပြထားသည့်အတိုင်း ထည့်ပေးနိုင်သည်။

```
def describe_person(first_name, last_name, age=None,
```

```
                    favorite_language= None, died=None):
    print("First name: %s" % first_name)
    print("Last name: %s" % last_name)
    # Optional information:
    if age:
        print("Age: %d" % age)
    if favorite_language:
        print("Favorite language: %s" % favorite_language)
    if died:
        print("Died: %d" % died)

    print("\n")          # Blank line at end.

describe_person('brian', 'kernighan', favorite_language='C')
describe_person('ken', 'thompson', age=70)
describe_person('dennis','ritchie', favorite_language='C',died=2011)
describe_person('guido', 'van rossum', favorite_language='Python')
```

```
First name: brian
Last name: kernighan
Favorite language: C

First name: ken
Last name: thompson
Age: 70

First name: dennis
Last name: ritchie
Favorite language: C
Died: 2011

First name: guido
Last name: van rossum
Favorite language: Python
```

Positional နှင့် keyword argument နှစ်မျိုးကို ရောရွေလည်း ထည့်ပေးနိုင်သည်။

```
def add(a, b, c):
    return a+b+c

add(1, c=3, b=2)
```

အောက်ပါအတိုင်း keyword argument ၏ နောက်တွင် positional argument ထည့်ပေးခွင့်မပြုပါ။ (not allowed)

```
add(1, b=2, 3)      # positional argument after keyword argument
```

```
SyntaxError: positional argument follows keyword argument
```

အောက်တွင် argument 'a' ကို နှစ်ခါထည့်ပေးသည့်အတွက် error ပေးသည်။

```
add(1, b=2, a=3)      # multiple values for argument 'a'
```

```
TypeError: add() got multiple values for argument 'a'
```

Positional နှင့် keyword argument နှစ်မျိုးကို ရောရှုလည်း ထည့်ပေးနိုင်သော်လည်း စည်းမရှိ ကမ်းမရှိ ထည့်ချင်သလို ထည့်ခွင့်မရှိပါ။

12.6.6 Default Arguments(Arguments with default values)

Function အတွင်းရှိ တချို့သော parameter များအတွက် အဆင်သင့်သုံးနိုင်အောင် တန်ဖိုး(value) များ ထည့်ထားကြသည်။ ထိုသို့ အဆင်သင့်ထည့်ထားသည့် တန်ဖိုး(value)ကို အသုံးမပြုချင်ပါက function ကို ခေါ်သုံးသူက မိမိကြိုက်နှစ်သက်သည့် တန်ဖိုးများ ထည့်ပေးနိုင်သည်။

Default argument ဆိုသည်မှာ function ကို ခေါ်သုံးသည့်အခါ argument value မထည့် ပေးလျှင် သုံးရန်အတွက် function အတွင်း၌ ကြိုတင်သတ်မှတ်ပေးထားသည့်တန်ဖိုး(default value)များဖြစ်သည်။ Default value များကို သတ်မှတ်ပေးသည့်အခါ assignment operator (=) ကို အသုံးပြုသည်။

```
def division(first, second =2):
    return first/second

quotient = division(10)
print(quotient)
```

```
5.0
```

အထက်က ဥပမာတွင် function ကို ခေါ်သုံးသည့်အချိန်တွင် second ဟုခေါ်သည့် argument အတွက် မထည့်ပေးသောကြောင့် default value ကို သုံး၍ အဖြေထုတ်ပေးသည်။ တန်ဖိုးတစ်ခုခု ထည့်ပေး လျှင်(passed value) default value ကို override လုပ်ပုံကို အောက်တွင် ဖော်ပြထားသည်။

```
def division(first, second =2):
    return first/second

quotient = division(10,5)
print(quotient)
```

```
2.0
```

မှတ်ရန်။ ။ Default argument များကို caller က တန်ဖိုးတစ်ခုခုထည့်ပေးလျှင်(passed value)) default value ကို မသုံးဘဲ ထည့်ပေးသည့် တန်ဖိုးကို အသုံးပြုသည်။

Function တစ်ခုတွင် ကြိုတင်သတ်မှတ်ထားသည့် တန်ဖိုး(predefined value) များသည် default argument များ ဖြစ်ကြသည်။ Function ခေါ်သည့်အချိန်တွင် default argument များ မထည့်ပေးမိပါက ကြိုတင်သတ်မှတ်ထားသည့် တန်ဖိုး(predefined value)ကို အသုံးပြုသည်။ ထည့်ပေးပါ ထိုထည့်ပေးသည့် default argument ကို အသုံးပြုသည်။

```
def add_one(a, b, c, d=4):
    print(a+1 , b+1, c+1, d+1)
```

```
add_one(1, 2, 3)                    # default argument မထည့်ပေးပါ။
add_one(1, b=2, c=3, d=100)
```

```
2 3 4 5
2 3 4 101
```

12.6.7 Variable Number of Arguments (*args)

တစ်ခါတစ်ရံ function တွင်းသို့ ထည့်ပေးရမည့် argument များ၏ အရေအတွက်ကို ကြိုမသိနိုင်သည့်အခါ များရှိသည်။ ထိုအခါမျိုးတွင် * (asterisk) syntax ကို အသုံးပြုနိုင်သည်။ ***args** ကို သုံးလျှင် မိမိကြိုက်သလောက် ထည့်နိုင်သည်။ ***args** ဥပမာ တစ်ခုကို အောက်တွင် ဖော်ပြထားသည်။

```
def addition(*args):
    total = 0
    for i in args:
        total += i
    return total

answer = addition(20, 10, 5, 1)
print(answer)
```

```
36
```

12.6.8 Variable Length Arguments (*args and **kwargs)

Parameter ကို one asterisk (*)ဖြင့် စထားလျှင် ထို function ထဲသို့ positional argument အရေအတွက် ကြိုက်သလောက် ထည့်ပေးနိုင်သည်။ (ဥပမာ - *args)

Parameter ကို two asterisks (**) ဖြင့် စထားလျှင် ထို function ထဲသို့ keyword argument အရေအတွက် ကြိုက်သလောက်ထည့်ပေးနိုင်သည်။ (ဥပမာ - **kwargs)

```
def add(a, b, *args, **kwargs):
    print(a, b)
    for arg in args:
        print(arg)
    for kwarg in kwargs:
        print(kwarg, kwargs[kwarg])

# 3, 4, 5 are combined into args
# six and seven are combined into kwargs
add(1, 2, 3, 4, 5, six=6, seven=7)
print()

# omitting of args or kwargs is also possible
add(1, 2, three=3)
```

```
1 2
3
4
```

```
5
six 6
seven 7

1 2
three 3
```

12.6.9 Accepting an Arbitrary Number of Arguments

အောက်မှ **adder()** function ကိန်း(၂)လုံးပေါင်းသည့် functionတစ်ခုဖြစ်သည်။ သို့သော် ကိန်း(၂) တည်းသာ အတိအကျ ပေါင်းနိုင်သည့် function ကို မည်သူမျှ မရေးလိုပါ။ ကိန်းအလုံးမည်မျှ ဖြစ်ပါစေ ပေါင်းပေး နိုင်သည့် function မျိုးရေးသင့်သည်။ Argument အရေအတွက် ကြိုက်သလောက်ထည့်နိုင် ရန်တွက် *args နှင့် **kwargs ကို အသုံးပြုသည်။

```
def adder(num_1, num_2):
    sum = num_1 + num_2
    print("The sum of your numbers is %d." % sum)

adder(1, 2)
adder(-1, 2)
```

```
The sum of your numbers is 3.
The sum of your numbers is 1.
```

arg_1 နှင့် arg_2 တို့သည် positional argument များ ဖြစ်ကြသည်။ ကျန် argument များသည် *arg_3 ဖြစ်ကြသည်။

```
def example_function(arg_1, arg_2, *arg_3):
    print('arg_1:', arg_1)
    print('arg_2:', arg_2)
    for value in arg_3:
        print('arg_3 value:', value)

example_function(1, 2, 3, 4, 5)
```

```
arg_1: 1
arg_2: 2
arg_3 value: 3
arg_3 value: 4
arg_3 value: 5
```

```
def adder(num_1, num_2, *nums):
    sum = num_1 + num_2

    for num in nums:
        sum = sum + num
    print("The sum of your numbers is %d." % sum)

adder(1, 2, 3, 4, 5, 6)
```

```
The sum of your numbers is 21.
```

12.6.10 Accepting an Arbitrary Number of Keyword Arguments

Python တွင် keyword argument အရေအတွက် ကြိုက်သလောက် ထည့်ပေးနိုင်သည်။

```
def example_function(arg_1, arg_2, **kwargs):
    print('arg_1:', arg_1)
    print('arg_2:', arg_2)
    print('arg_3:', kwargs)

example_function('a', 'b')
example_function('a', 'b', value_3='c', value_4='d', value_5='e')
```

```
arg_1: a
arg_2: b
arg_3: {}

arg_1: a
arg_2: b
arg_3: {'value_3': 'c', 'value_4': 'd', 'value_5': 'e'}
```

တတိယ argument သည် asterisk နှစ်ခုပါသည့် argument ဖြစ်သည်။ ပထမ နှစ်ခုမှ လွဲ၍ ကျန် key-value argument များကို ****kwargs** က လက်ခံပေးသည်။ kwargs သည် key argument ၏ အတိုခေါက် ဖြစ်သည်။ Function ထဲသို့ ထည့်မည့် value များကို dictionary ပုံစံဖြင့် ထည့်ပေးသည်။

```
def example_function(arg_1, arg_2, **kwargs):
    print('arg_1:', arg_1)
    print('arg_2:', arg_2)
    for key, value in kwargs.items():
        print('arg_3 value:', value)

example_function('a', 'b')
example_function('a', 'b', value_3='c', value_4='d', value_5='e')
```

```
arg_1: a
arg_2: b

arg_1: a
arg_2: b
arg_3 value: c
arg_3 value: d
arg_3 value: e
```

12.6.11 Forced Keyword Arguments

- Function parameter list ထဲတွင် '*,' ထည့်ရေးနိုင်သည်။ ထို'*,' ၏နောက်တွင် ထည့်ထားသည့် parameter အားလုံးသည် keyword argument များ ဖြစ်သည်။ keyword argument များ အဖြစ် ထည့်ပေးရမည်။
- Variable-length argument ၏ နောက်တွင် ရှိသည့် argument အားလုံးသည် keyword argument များသာဖြစ်သည်။

အောက်က ဥပမာတွင် `(a, b, *, c, d):` မှ a နှင့် b သည် positional argument ဖြစ်သည်။ ၄င်းနောက်တွင် Variable-length argument တစ်ခုရှိသည်။ ထည့်ပေးမည့် အရေအတွက် အတိအကျမရှိပါ။ ထို့နောက်တွင် keyword argument များသာ ထည့်ပေးခွင့်ရှိသည်။

```
def foo(a, b, *, c, d):
    print(a, b, c, d)

foo(1, 2, c=3, d=4)
```

```
1 2 3 4
```

Variable-length argument ၏ နောက်တွင် keyword argument များသာ ရှိရမည်။

```
#arguments after variable-length arguments must be keyword arguments
def my_func(*args, last):
    for arg in args:
        print(arg, end= " ")
    print  ("\n", last)

my_func(8, 9, 10, last=50)
```

```
8 9 10
 50
```

12.6.12 Unpacking into Arguments (Asterisk(*) တစ်ခုသာ ပါသည့် Argument)

List များ သို့မဟုတ် tuple များကို **one asterisk(*) argument** များ အဖြစ် unpack လုပ်နိုင်သည်။ container ထဲတွင် ရှိသည့် အရေအတွက်သည် function parameter အရေအတွက် နှင့် ကိုက်ညီရန်လိုသည်။ (the length of the container matches the number of function parameters.)

```
def add_one(a, b, c):
    print(a+1, b+1, c+1)

# list/tuple unpacking, length must match
my_list = [4, 5, 6]                # list or tuple
add_one(*my_list)
```

```
5 6 7
```

asterisks (**) နှစ်ခု ပါသည့် argument

Dictionary များကို **two asterisks (**) argument** များ အဖြစ် unpack လုပ်နိုင်သည်။ Dictionary ထဲတွင် ရှိသည့် key များအရေအတွက်သည် function parameter အရေအတွက် နှင့် ကိုက်ညီရန် လိုသည်။ (the the length and the keys match the function parameters.)

```
# dict unpacking, keys and length must match
my_dict = {'a': 1, 'b': 2, 'c': 3}
add_one(**my_dict)
```

```
2 3 4
```

Function parameter တွင် `def add_one(a, b, c):` ဟုသာထည့်ထားသော်လည်း function ခေါ်သည့် argument တွင် 'd' ပါသောကြောင့် TypeError ပေးသည်။

```
# dict unpacking, keys and length must match
my_dict = {'a': 1, 'b': 2, 'd': 3} #not possible since wrong keyword
add_one(**my_dict)
```

```
TypeError                 Traceback (most recent call last)
TypeError: add_one() got an unexpected keyword argument 'd'
```

12.6 Global and Local Variables

Function ထဲမှာ define လုပ်ထားတဲ့ variable ကို **local variable** ဟု ခေါ်သည်။ Local variable တွေက သူ့ကို define လုပ်ထားတဲ့ function တွေထဲမှာပဲ အကျုံးဝင်သည်။ သုံးလို့ ရသည်။ (accessible ဖြစ်သည်။ Access လုပ်လို့ ရသည်။) Local scope လို့လည်းခေါ်သည်။

Function ရဲ့ အပြင်ဘက်မှာ define လုပ်ထားတဲ့ variable ကို **global variable** ဟုခေါ်သည်။ Global variable ကို function ရဲ့ အတွင်းမှာကော function ပြင်ပမှာပါ သုံးလို့ ရပါသည်။ (access လုပ်လို့ ရသည်)။ Global scope လို့လည်းခေါ်ပါတယ်။

Global variable များကို function body အတွင်း၌သာ access လုပ်နိုင်သည်။ Global variable များကို modify လုပ်လိုပါက `global var_name` အဖြစ် ကြေငြာရန် လိုအပ်သည်။

```
total = 0                   # This is global variable.

def sum( arg1, arg2 ):
   total = arg1 + arg2;   #total is local variable(inside the function)
   print("Inside the function local total : ", total)
   return total

sum( 10, 20 )                # Now you can call sum function

print("Outside the function global total : ", total)
```

```
Inside the function local total :  30
Outside the function global total :  0
```

12.7 Using main()

Java နှင့် C++ စသည့် programming language တို့တွင် special function ဟုခေါ်သည့် **main()** function လိုအပ်သည်။ Program ကို ခေါ်ခိုင်းသည့်အခါ(program is invoked) code ကို execute လုပ်ရန် operating system ကို ညွှန်ကြားလိုက်သည်။ Python တွင် ထိုသို့ main() ပါရှိ ရန် မဖြစ်မနေ မလိုအပ်သော်လည်း program structure ကောင်းစေရန် အလေ့အကျင့်ကောင်းတစ်ခု ဖြစ်ရန်အတွက် main() function ကို ထည့်သွင်း ရေးသားလေ့ရှိသည်။

Python interpreter သည် program တစ်ခုကို execute မလုပ်ခင် special variable တစ်ချို့ကို define လုပ်သည်။ ထို special variable ထဲမှ တစ်ခုသည် **__name__** ဖြစ်သည်။ Double underscores ကို *dunders* ဟု အတိုခေါက်ခေါ်သည်။ Program တစ်ခုသည် သူ့ အလိုအလျောက် execute လုပ်မည်ဆိုလျှင် **__main__** ဟု set(==) လုပ်သည်။ တစ်နည်းအားဖြင့် **__name__ == "__main__":** သည် standalone program ဖြစ်သည်။ Standalone fashion ဖြစ်သည်

Program တစ်ခုကို တစ်ခြား program တစ်ခုမှ လှမ်းပြီး import လုပ်မည်ဆိုလျှင် **__name__** သည် တခြား program ၏ နာမည် ဖြစ်လိမ့်မည်။ ထို့ကြောင့် program တစ်ခုသည် standalone program ဖြစ်မည် သို့မဟုတ် တစ်ခြား program မှ ခေါ်သုံးမည် (import မည့် program ဖြစ်မည်)ကို **__main__** ကို ကြည့်၍ ခန့်မှန်းနိုင်သည်။ **main()** function ကို အသုံးပြုထားသည့် ဥပမာ

```
def summation(first, second):
    total = first + second
    return total

def main():
    outer_total = summation(10,20)* 2
    print('Double the total is '+ str(outer_total))

if __name__ == "__main__":
    main()
```

```
Double the total is 60
```

__name_ == __main__ မပါလျှင် function အလုပ်လုပ်စေရန် **summation()** ဖြင့် လှမ်းခေါ်ရမည်။ __name_ == __main__ ပါခြင်း မပါခြင်းသည် အဓိကကြသည့် အချက်မဟုတ်ပါ။

အချုပ်အားဖြင့် __name_ == __main__ ပါလျှင် function ကို ခေါ်စရာမလိုပါ။ ထို function ပါသည့် program ကို run လျှင် သူ့အလိုလို အလုပ်လုပ်လိမ့်မည်။ __name_ == __main__ မပါလျှင် function အလုပ်လုပ်စေရန် function ကို ခေါ်ပေးရသည်။

```
print(__name__)
```

```
__main__
```

Programming language များတွင် **__main__** သည် execute စလုပ်သည့် နေရာဖြစ်သည်။ (starting point of execution)။ Python project တစ်ခုတွင် module ပေါင်းများစွာ ရှိနိုင်သည်။ ထိုအခါ

ဘယ် module ကို အရင်ဆုံး run မလဲဆိုတာကို အသိပေးရန် လိုအပ်သည်။ အရင်ဆုံး run မည့် module နာမည်ကို **__main__** နှင့် ညီပေးရသည်။ (first module name is always main.)။ ထို့ကြောင့် **__main__** ပါသည့် module သည် အရင်ဆုံး run ရမည့် module ဖြစ်သည်။ Point of execution ဖြစ်သည်။

```
print(f"First Module's Name:{__name__} ")
```

```
First Module's Name:__main__
```

Python inerpeter က ပရိုဂရမ်ဖိုင်ထဲက ကုဒ်တွေကို မ run ခင် special variable တွေကို အရင် လိုက်ရှာသည်။ Name သည် special variable တစ်ခုဖြစ်သည်။ Module ကို import လုပ်ရင် **__name__** ကို file name ဆီကို ပို့ပေးသည်။

```
 def main():
    pass

if __name__ == '__main__':
    main()

type(main)          # main သည် fuction ဖြစ်သည်။
```

```
function
```

ဒီလို ထည့်ရေးထားရင် main method ကို Python က တိုက်ရိုက် run လို့ရသလို import လုပ်၍လည်း ရသည်။

```
def main():                     # file name is first_module.py
    print(f"First Module's Name:{__name__} ")

if __name__ == '__main__':
    main()
```

```
First Module's Name:__main__
```

ပိုပြီး ရှင်းအောင်လို့ ဖိုင်ကို တိုက်ရိုက် run တာလား သို့မဟုတ် တခြား Module တစ်ခုခုက လှမ်းခေါ်ပြီး run တာလား သိရအောင် "Run Directly" နှင့် "Run From Import" နှစ်မျိုးခွဲပြီး print ထုတ်ကြည့်ရအောင်။

```
if __name__ == '__main__':
    print("Run Directly")
else:
    print("Run From Import")
```

```
Run Directly
```

တခြား module ကနေ main method ပါနေတဲ့ module ကို လှမ်းပြီးတော့လည်း import လုပ်လို့ရသည်။

```
def main():                          # file name is first_module.py
```

```
    print(f"First Module's Name:{__name__} ")

if __name__ == '__main__':
    print("Run Directly")
else:
    print("Run From Import")
```

တခြား Module တစ်ခုခု ကလှမ်းခေါ်ပြီး run သည်။

```
import first_module                    # file name is second_module.py
    print(f"Second Module's Name:{__name__} ")
```

```
Run From Import
Second Module's Name:__main__
```

`first_module.py` ကို `second_module.py` import လုပ်ပြီး run သည်.

ဥပမာ outer_name အမည်ပေးထားသည့် function တစ်ခု တည်ဆောက်သည်။

```
def outer_name(new_name):
    name = "outer " + new_name

    def inner_name():
        name = "inner " + new_name
    inner_name()

    return name
```

12.8 Lambda Functions

Pythonတွင် lambda function များသည် နာမည်မရှိသည့် anonymous function များဖြစ်ကြသည်။ အတိုခေါက်အားဖြင့် lambdas ဟုခေါ်ဆိုကြသည်။ Function များတွင် function name ရှိရမည် ဟုသတ်မှတ်ထားသော်လည်း lambda function များအတွက် နာမည် မရှိပါ။

syntax: `lambda parameters: expression.`

lambda function တစ်ခု ဖြစ်ကြောင်း ကြေငြာရန်အတွက် lambda keyword ကို အသုံးပြုသည်။ pertinent parameters ကို (We then list the pertinent parameters, the number of which can be zero to multiple.)

colon ၏ နောက်တွင် expression ကို ရေးသည်။ Expression သည် လုပ်ဆောင်ပေးရမည့် အလုပ်ဖြစ်သည်။ Lambda function တစ်ခု ပြုလုပ်သည်။

```
triple = lambda x: x*3
triple
```

```
<function __main__.<lambda>(x)>
```

Lambda function ကို ခေါ်သုံးသည်။

```
triple(5)
```

```
15
```

12.8.1 Sort Sequences of Data

ဥပမာ # Create a sequence of data

```
grades = [{'name': 'Jennifer', 'final': 95},
        {'name': 'David', 'final': 92},
        {'name': 'Aaron', 'final': 98}]
grades
```

```
[{'name': 'Jennifer', 'final': 95},
 {'name': 'David', 'final': 92},
 {'name': 'Aaron', 'final': 98}]
```

အထက်က list ကို နာမည်မှ အက္ခရာဖြင့် ရှေ့နောက်စဉ်သည့် lambda function ကို ရေးသည်။

```
sorted(grades, key=lambda x: x['name'])
```

```
[{'name': 'Aaron', 'final': 98},
 {'name': 'David', 'final': 92},
 {'name': 'Jennifer', 'final': 95}]
```

နောက်ဆုံးနှစ်အဆင့်(final grades)များဖြင့် ရှေ့နောက်စဉ်သည့် lambda function ကို ရေးသည်။ (descending order)

```
sorted(grades, key=lambda x: x['final'], reverse=True)
```

```
[{'name': 'Aaron', 'final': 98},
 {'name': 'Jennifer', 'final': 95},
 {'name': 'David', 'final': 92}]
```

ပေးထားသည့် data များထဲမှာ အများဆုံး(maximal values)နှင့် အနည်းဆုံး(minimal values)ကို ရှာသည့် built-in function **min()** နှင့် **max()** ရှိသည်။

- အမှတ်အများဆုံးရသူကို lambda function ဖြင့် ရှာသည်။ (Get the highest final score)
- key argument t ကို lambda x: x[‘final’] ဖြင့် သတ်မှတ်သည်။

```
max(grades, key=lambda x: x['final'])
{'name': 'Aaron', 'final': 98}
```

```
{'name': 'Aaron', 'final': 98}
```

အမှတ် အနည်းဆုံးရသူကို lambda function ဖြင့် ရှာသည်။ (Get the highest final score)

```
min(grades, key=lambda x: x['final'])
```

```
{'name': 'David', 'final': 92}
```

12.9 Glossary

Function

Function ဆိုသည်မှာ ထပ်ခါထပ်ခါ လုပ်ရမည့် အလုပ်တစ်ခုခုကို လုပ်ပေးနိုင်ရန် အစီအစဉ်တကျ ရေးထားပြီး(sequence) နာမည်ပေးထားသည့် statement များ စုစည်းထားသည့် ကုဒ် အစုအဝေး ဖြစ်သည်။ (A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.)

Function definition

Function definition ဆိုသည်မှာ function အသစ်တစ်ခု တည်ဆောက်ရန် ၊ function နာမည်ပေးရန် နှင့် parameter များ, နှင့် statement များကို execute လုပ်ရန်အတွက် ဖြစ်သည်။ (A statement that creates a new function, specifying its name, parameters, and the statements it executes.)

Function object

Function object ဆိုသည်မှာ function definition တစ်ခုက ထုတ်ပေးလိုက်သည့် တန်ဖိုး(value) တစ်ခု ဖြစ်သည်။ (A value created by a function definition.)

Header

Header ဆိုသည်မှာ function definition တွင် ပါရှိသည့် ပထဆုံးလိုင်း ဖြစ်သည်။ (The first line of a function definition.)

Body

Body ဆိုသည်မှာ function definition အတွင်း၌ရှိနေသည့် အစီအစဉ်တကျ ရေးထားသည့် statement များဖြစ်သည်။ (The sequence of statements inside a function definition.)

Parameter

Parameter ဆိုသည်မှာ function အတွင်းသို့ argument အနေဖြင့် ထည့်ပေးလိုက်သည့် တန်ဖိုး (value)၏ နာမည် ဖြစ်သည်။ (A name used inside a function to refer to the value passed as an argument.)။ တစ်နည်းအားဖြင့် function အတွင်းသို့ ထည့်ပေးသည့် နာမည် argument ဖြစ်သည်။

Function call

Function call ဆိုသည်မှာ တည်ဆောက်ထားသည့် function ကို execute လုပ်ရန် စေခိုင်းသည့် statement ဖြစ်သည်။ ထို call statement တွင် function name နှင့် argument list တို့ပါဝင်သည်။ Function name နှင့် အရင် စရသည်။ Function name နောက်တွင် argument list လိုက်သည်။ (A statement that executes a function. It consists of the function name followed by an argument list.)

Argument:

Argument ဆိုသည်မှာ ခေါ်သုံးမည့် function ထဲသို့ ထည့်ပေးမည့် တန်ဖိုး(value) ဖြစ်သည်။ Function ကို ခေါ်လိုက်သည့်အခါ(call လုပ်သည့်အခါ) ထည့်ပေးမည့် တန်ဖိုး(value) ဖြစ်သည်။ ထို တန်ဖိုး (value)သည် function ထဲ၌ သတ်မှတ်ထားသည့် parameter နှင့် ကိုက်ညီမှု ရှိရန်လိုအပ်သည်။ (A value

provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.)

Local variable

Local variable ဆိုသည်မှာ function တစ်ခု အတွင်း၌ define လုပ်ထားသည့် variable ဖြစ်သည်။ Local variable ထို define လုပ်ထားသည့် function တစ်ခု အတွင်း၌ အသုံးပြုနိုင်သည်။ (A variable defined inside a function. A local variable can only be used inside its function.)

Return value

Return value ဆိုသည်မှာ function တစ်ခုမှ ပြန်ထုတ်ပေးသည့် ရလဒ်(result) ဖြစ်သည်။ (The result of a function.)

Fruitful function

Fruitful function ဆိုသည်မှာ တန်ဖိုး(value)တစ်ခုခုကို ပြန်ထုတ်ပေးသည့် function ဖြစ်သည်။ (A function that returns a value.)

Void function

Void function ဆိုသည်မှာ တန်ဖိုး(value)တစ်ခုခု ပြန်မထုတ်ပေးသည့် function ဖြစ်သည်။ (A function that doesn’t return a value.)

Module

Module ဆိုသည်မှာ function များ နှင့် definition များ စုစည်းထားသည့် ဖိုင် ဖြစ်သည်။ (A file that contains a collection of related functions and other definitions.)

Import statement

Import statement ဆိုသည်မှာ module file ကို ဖတ်ပြီး module object များကို ပြုလုပ်ပေးမည့် statement ဖြစ်သည်။ (A statement that reads a module file and creates a module object.)

Module object

Module object ဆိုသည်မှာ module များကို import လုပ်သည့် statement မှ တဆင့် Module ထဲတွင် define လုပ်ထားသည့် တန်ဖိုး(value)များကို လှမ်းယူ(access လုပ်)လိုက်သည့် တန်ဖိုး(value) ဖြစ်သည်။ (A value created by an import statement that provides access to the values defined in a module.)

Dot notation

Dot notation ဆိုသည်မှာ တခြား module ရှိ function ကို ခေါ်သုံးရန်အတွက်(for calling a function) syntax ဖြစ်သည်။ ရေးရမည့် အစီအစဉ်မှာ module name.function name ဖြစ်သည်။ (The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.)

Flow of execution

Flow of execution ဆိုသည်မှာ program run သည်အခါ execute လုပ်ရမည့် statement များ၏ အစီအစဉ် ဖြစ်သည်။ (The order in which statements are executed during a program run.)

Stack diagram

Stack diagram ဆိုသည်မှာ stack of function များနှင့် ၎င်းတို့၏ variable များ နှင့် ရည်ညွှန်းထားသည့်value များကို ဂရပ်ပုံစံဖြင့် ဖော်ပြထားသည့် diagram ဖြစ်သည်။ (A graphical representation of a stack of functions, their variables, and the values they refer to.)

Frame

Frame ဆိုသည်မှာ function call ကို ရည်ညွှန်းရန် stack diagram ထဲတွင် ဖော်ပြထားသည့် လေးထောင့်ကွက် ဖြစ်သည်။ ထိုထဲတွင် function ၏ local variable များနှင့် parameter များ ပါဝင်သည်။ (A box in a stack diagram that represents a function call. It contains the local variables and parameters of the function.)

-End-