

# Begin Modern Programming

## with

C  
O  
D  
I  
T  
Y  
A

Pyi Soe

## မာတိကာ

၁	ခက်ရပ်ကားရဲလှဖြင့် ပရိုဂရမ်းမင်းမိတ်ဆက်	၁
၁.၀	ခက်ရပ် ကားရဲလှ .....	၁
၁.၂	Meet Karel ပရိုဂရမ် .....	၃
၁.၃	ကားရဲလှ ပရိုဂရမ် run ခြင်း .....	၈
၁.၄	Python အင်စတောလ်လုပ်တဲ့အခါ ဘာတွေပါလဲ .....	၉
၁.၅	Move Beeper to Other Side .....	၁၀
၂	ကားရဲလှနှင့် ကွန်ထရီးလ် စတိတ်မန္ဒား	၁၃
၂.၀	<b>for</b> loop .....	၁၃
၂.၂	အင်ဒန်ထလုပ်ခြင်းနှင့် ကွန်စထရက်ချာ .....	၁၆
၂.၃	<b>while</b> loop .....	၁၆
၂.၄	<b>if</b> စတိတ်မန္ဒာ .....	၁၉
၂.၅	<b>if...else</b> စတိတ်မန္ဒာ .....	၂၀
၂.၆	Nested ကွန်ထရီးလ် စတိတ်မန္ဒား .....	၂၂
၃	ဖန်ရှင်များ (Functions)	၂၆
၃.၀	ဖန်ရှင် သတ်မှတ်ခြင်း .....	၂၆
၃.၂	ပရိုကွန်ဒီရှင်နှင့် ပို့စ်ကွန်ဒီရှင် .....	၂၂
၃.၃	ဖန်ရှင်များဖြင့် abstraction များ တည်ဆောက်ခြင်း .....	၂၇
၃.၄	Bottom-up Programming .....	၂၈
၃.၅	Top-down Programming .....	၂၉
၄	ကားရဲလှနှင့် ရိုကားဆိုင် ဖန်ရှင်များ	၅၁
၄.၀	ရိုကားဆိုင် ဖန်ရှင် ဘယ်လို အလုပ်လုပ်လဲ .....	၅၀
၄.၂	ရိုကားဆိုင်စွဲ့ဖြင့် ပုံစံဖြန့်ခြင်း .....	၅၂
၅	ဒေတာများနှင့် ဖန်ရှင်များ	၆၅
၅.၁	ကိန်းဂဏ်းများ .....	၆၅

၅.၂ ‘တိက်ပဲ’ ဆိတာ ဘာလ .....	၃၂
၅.၃ ဖေရီရေတဲ့လုံး .....	၃၃
၅.၄ ဓမ္မားများ .....	၃၄
၅.၅ အီပိုစ်ပရက်ရှင်များ .....	၃၅
၅.၆ ဘူလီယန် တိက်ပဲနှင့် ဘူလီယန် အီပိုစ်ပရက်ရှင် .....	၃၆
<b>၆ အော်ဂျက်များ</b>	<b>၃၇</b>
၆.၀ <b>date, time and datetime</b> .....	၃၈
၆.၂ <b>list</b> .....	၄၀
<b>၇ ကွန်ထရီးလ် စတိတ်မန္ဒြား</b>	<b>၄၃</b>
၇.၁ <b>if</b> စတိတ်မန္ဒြား .....	၄၃
၇.၂ <b>for</b> Loop .....	၄၅
၇.၃ Python Arcade Library .....	၁၀၅
၇.၄ <b>while</b> loop .....	၁၀၀
၇.၅ လောက့်ခန်း ဥပမာများ .....	၁၀၉
<b>၈ ဖန်ရှင်များ</b>	<b>၁၂၀</b>
၈.၁ တန်ဖိုးပြန်ပေးတဲ့ ဖန်ရှင်များ .....	၁၂၀
၈.၂ တန်ဖိုးပြန်မပေးတဲ့ ဖန်ရှင်များ .....	၁၂၅
၈.၃ Exceptions .....	၁၂၇
၈.၄ ဖန်ရှင်နှင့် ပရိုဂရမ်ဒီဇိုင်း ဥပမာ (၁) “Insurance Premium” .....	၁၃၀
၈.၅ ဖန်ရှင်နှင့် ပရိုဂရမ်ဒီဇိုင်း ဥပမာ (၂) “Colorful House” .....	၁၃၃
၈.၆ ပါရာမီတာ ခွေးချယ်သတ်မှတ်ခြင်း .....	၁၄၂
၈.၇ တန်ဖိုးပြန်ပေးခြင်း၊ မပေးခြင်း .....	၁၄၅
၈.၈ <b>return</b> .....	၁၄၆
<b>၉ From Classes to Objects (ကလပ်စ်များမှ အော်ဂျက်များသိ)</b>	<b>၁၅၉</b>
၉.၁ <b>Account Class</b> (အကောင့် ကလပ်စ်) .....	၁၅၉
၉.၂ Access Control .....	၁၆၄
၉.၃ Composition or Has-a Relationship .....	၁၆၆
၉.၄ Properties .....	၁၇၀
၉.၅ Class Attributes and Methods .....	၁၇၃
၉.၆ Static Methods .....	၁၇၆
၉.၇ Object-Oriented Programming .....	၁၈၀

<b>၁၀ Inheritance and Polymorphism</b>	<b>၁၉၅</b>
၁၀.၀ Inheritance ဥပမာ ‘ <b>Account</b> ’ Class Hierarchy <sup>၅</sup> . . . . .	၁၉၅
၁၀.၂ Overriding. . . . .	၁၉၉
၁၀.၃ Multilevel Inheritance . . . . .	၂၀၀
၁၀.၄ Class Hierarchy and UML . . . . .	၂၀၀
၁၀.၅ Is-A Relationship and Inheritance . . . . .	၂၀၀
၁၀.၆ အသံချုံ ဥပမာ (၁) Breakout Game. . . . .	၂၀၀
၁၀.၇ Breakout တည်ဆောက်ခြင်း. . . . .	၂၀၂
<b>၁၁ Exceptions and Exception Handling</b>	<b>၂၂၅</b>
၁၁.၀ Raising Exception . . . . .	၂၂၆
၁၁.၂ Handling Exception. . . . .	၂၂၈
၁၁.၃ တယ်နေရာမှာ handle လုပ်သင့်လဲ . . . . .	၂၃၀
၁၁.၄ Exception Handling နှင့် Control Flow . . . . .	၂၃၂
၁၁.၅ Built-in and User-defined Exceptions . . . . .	၂၃၆
၁၁.၆ Handling Multiple Exception. . . . .	၂၃၇
<b>၁၂ ဒေတာကွောစိများနှင့် ဆက်သွယ်ဆောင်ရွက်ခြင်း</b>	<b>၂၄၃</b>
<b>၁၃ File Input and Output</b>	<b>၂၄၅</b>
<b>၁၄ Graphical User Interface (GUI) Programming</b>	<b>၂၄၇</b>
<b>၁၅ Basic Concurrency</b>	<b>၂၄၉</b>
<b>၁၆ Miscellaneous</b>	<b>၂၅၀</b>
<b>က လိုအပ်သည့် ဆောင်ရွက်ခြင်း</b>	<b>၂၅၃</b>
<b>ခ ကားရဲလ်ပရီဂရမ် ဖီချာများ</b>	<b>၂၇၉</b>



# အခန်း ၁

## စက်ရုပ်ကားရဲလိုဖြင့် ပရီဂရမ်းမင်းမိတ်ဆက်

ကွန်ပျိုတာတွေဟာ သက်မဲ့ စက်ပစ္စည်းတော်ပါပဲ။ ကားတို့ လေယာဉ်တို့နဲ့ မတူတာက ကွန်ပျိုတာတွေဟာ စက်ချဉ်းသက်သက် ဘာအစွမ်းမှ မယ်မယ်ရရ မရှိဘူး။ ဒါပေမဲ့ ဆောင်ရွက်လိုတဲ့ ကိစ္စအဝေဝအတွက် ပရီဂရမ်းမျိုးမျိုး ထည့်ပေးလိုက်တဲ့ အခါမာ သူ့အစွမ်းက အတိုင်းအဆမဲ့ပဲ။ နေရာမျိုးစံ၊ နယ်ပါယ်မျိုးစံ မှာ အကူးအညီပေးနိုင်တဲ့ စွယ်စုံသုံး ပစ္စည်းတစ်ခုဖြစ်သွားတယ်။ ဂိတ်သံစဉ်တွေကို ဖွင့်ပေးနိုင်သလို အသံလည်းသွင်းပေးနိုင်တယ်။ ရုပ်ရှင်တည်းဖြတ် လုပ်ချင်တာလား။ ပြဿနာမရှိဘူး၊ ကူညီပေးနိုင်တယ်။ နျှော လီးယား ပါတ်ပေါင်းဖို့တွေကို စီမံခိုင်သလို မောင်းသူမဲ့ အုံပျံတွေကိုလည်း ပဲထိန်းပေးနိုင်တယ်။

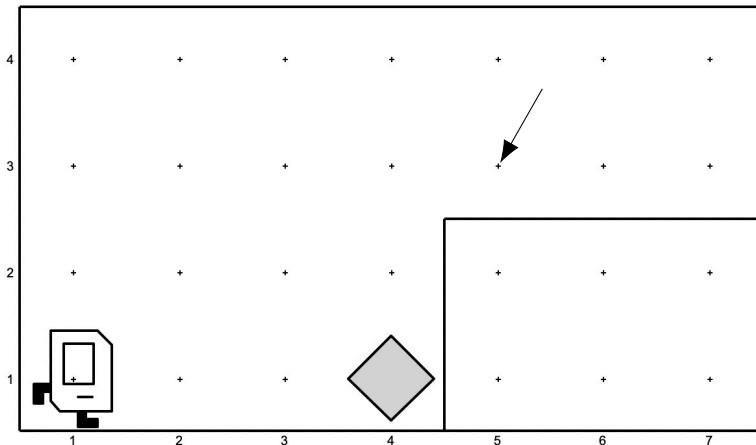
ကျွန်တော်တို့တွေ နိုစ္စဝေ အသုံးပြုနေကြတဲ့ ကား၊ စမတ်နှင့်း၊ လက်ပါတ်နာရီ၊ မိုက်ခရှိဝေဖွံ့ဖို့ အဝတ်လျှော့စက် စတဲ့ စက်ပစ္စည်း အမျိုးမျိုးဟာလည်း ကွန်ပျိုတာတွေနဲ့ မကင်းပြန်ပါဘူး။ “ကွန်ပျိုတာနည်းပညာ အကူးအညီမပါတဲ့ အတိုင်းဆက်သံစုံထွေ့မှုဆိုတာ မရှိဘူး” လို့ ဆိုနိုင်ပါတယ်။

တစ်ချက်တစ်ချက် ရိုက်ခတ်လိုက်တဲ့ ကွန်ပျိုတာနည်းပညာ လိုင်းလုံးကြီးတွေဟာ ကမ္မာတစ်စုံမှုးလုံး ပုံစံပြောင်းသွားလောက်အောင် အဟုန်ပြင်းထန်လာတယ်။ ဘီလီယံချွဲ့ခြုံတဲ့ လူတွေ ဆီရှယ်မီဒီယာတွေပေါ်က နေ ရုပ်သံတွေနဲ့ ချိတ်ဆက်ပြောင်းလို့ ရစေတာဟာလည်း ကွန်ပျိုတာစနစ်တွေပါပဲ။ Artificial Intelligence (AI) နည်းပညာကြောင့် သက်ရှိတွေမှာပဲတွေ့ရတဲ့ ညာက်ရည်မျိုးကို ကွန်ပျိုတာတွေမှာလည်း တွေ့လာရပါပြီ။ သံချုပ်စွာတွေ ဖြေရှင်းခြင်း၊ စစ်တုံးရှင်းထိုးခြင်း၊ စတဲ့ကိစ္စမျိုးတွေအပြင် ပန်းချီဆွဲခြင်း၊ ကဗျာရေးစပ်ခြင်း၊ သီချင်းရေးဖွံ့ခြင်း ကဲ့သို့ အနုပညာဖန်တီးမှုတွေကိုပါ AI က လုပ်ဆောင်ပေးနိုင်ပါတယ်။ နှစ်ဆယ်တစ်ရာစုံ၊ အထူးမြှားဆုံး AI နည်းပညာလိုင်းဟာ အရှိန်အဟုန်ပြင်းပြင်း ရိုတ်ခတ်ဖို့ အားယူစ ပြုနေပါပြီ။

‘ကွန်ပျိုတာ’ လိုပြောတဲ့ အခါ စက်ပစ္စည်းသက်သက် မဟုတ်ဘဲ ကွန်ပျိုတာမှုတ်ညာက်ထဲက ပရီဂရမ်တွေလည်း ပါဝင်တယ်ဆိုတာ သတိချုပ်ရပါမယ်။ ကွန်ပျိုတာတွေ တစ်စုံတစ်ရာ စွမ်းဆောင်နိုင်စေတဲ့ ပရီဂရမ်တွေ ရေးတဲ့ အလုပ်ကို ပရီဂရမ်းမင်း (Programming) လို့ခေါ်တယ်။

### ၁.၁ စက်ရုပ် ကားရဲလို

ပရီဂရမ်းမင်းဆိုတာ ဘယ်လိုမျိုးလဲ သဘောပေါက်အောင် စာတွေတစ်သီဥို့ရေး ရှင်းပြတာထက် ပရီဂရမ်လေးတွေ လက်တွေ၊ ရေးကြည့်လိုက်တာ ပိုပြီးထိရောက်ပါတယ်။ ဒါကြောင့် စက်ရုပ်ကားရဲလိုကို ပရီဂရမ်လေးတွေရေးပြီး အလုပ်တွေခုံးကြည့်ကြမယ်။ ပုံ (၁.၁) မှာ တွေ့ရတာက ကားရဲလို ရောက်ရှိနေတဲ့ နဗုံနာကမ္မာတစ်ခုပါ။ မီးခါးရောင် မှုန်ကူကုံးပုံလေးကို ဘိပါ (beeper) လို့ ခေါ်တယ်။ အဲဒီဘိပါကို မြားပြထားတဲ့ နေရာကို ရွှေခိုင်းချင်တယ်။ မျှေားမည်းအထူးတွေက နံရံတွေပါ။ ကားရဲလိုကို ကိစ္စတစ်ခု ဆောင်ရွက်စေ



ပုံ ၁.၁ ခက်ရုပ်လေး ကားရဲလ်

ချင်တဲ့အခါ အခြေခံ ကားရဲလ်ကွန်မန်းတွေကို အသုံးပြုရပါတယ်။ ကွန်မန်းတွေကို နှုတ်နှုံးပြောပြီး ခိုင်းရတာ မဟုတ်ဘဲ ပရိုကရမ်ရေးပြီး ခိုင်းရတာပါ။ ကားရဲလ်နားလည်တဲ့ ကွန်မန်းတွေကို ကြည့်ကြရအောင်။

### ကားရဲလ်ကွန်မန်းများ

မဖြစ်မနေ သိထားရမဲ့ အခြေခံ ကားရဲလ်ကွန်မန်း လေးခုပဲ ရှိတယ်။ move, turn\_left, put\_beeper နဲ့ pick\_beeper တို့ဖြစ်တယ်။ အခြား ကားရဲလ်ကွန်မန်း တွေလည်း ရှိပါသေးတယ်။ ဒါပေမဲ့ ကားရဲလ် ပရိုကရမ်းမင်း စလေ့လာဖို့ ဒီလေးခုနဲ့ပဲ လုံလောက်ပါပြီ။

move ကွန်မန်းက ကားရဲလ်ကို ရှေ့တစ်ကွန်နာကို ရွှေ့ခိုင်းတာ။ ကားရဲလ်ကမ္ဘာထဲမှာ တစ်ခုနဲ့တစ်ခု အကွာအဝေးတူ ခြားထားတဲ့ အတန်းလိုက် အတန်းလိုက် အစက်ကလေးတွေဟာ ကွန်နာ (corner) တွေ ဖြစ်တယ်။ ကမ္ဘာကို ပျော်မည်းမည် အထူး နံရုံတွေနဲ့ ထောင့်မှန်စတုဂံပဲ ပါတ်လည် ဘောင်ခတ်ထားတယ်။ ကွန်နာတွေကြားမှာလည်း နံရုံတွေရှိနိုင်တယ်။ နူးနာကမ္ဘာမှာ ဘေးတိုက် နံပါတ်စဉ် ၄ နဲ့ ၅ ကြား ထောင်လိုက် နံရုံတစ်ခု၊ အထက်အောက် နံပါတ်စဉ် ၂ နဲ့ ၃ ကြား အလျေားလိုက် နံရုံတစ်ခုကို တွေ့ရှုပါမယ်။ ကွန်နာရှေ့မှာ နံရုံကာနေရင် ကားရဲလ်ကို move ခိုင်းလို့မရပါဘူး။

put\_beeper က ကားရဲလ် လက်ရှိ ရှိနေတဲ့ ကွန်နာမှာ ‘ဘိပါတစ်ခုချု’ ထားခိုင်းတာ၊ pick\_beeper က ရပ်နေတဲ့ ကွန်နာမှာ ‘ဘိပါတစ်ခုကောက်’ ခိုင်းတာပါ။ ကွန်နာမှာ ဘိပါရှိနေမှ ကောက်ခိုင်းလို့ရမှုပါ။ မရှုရင် ကောက်ခိုင်းလို့ မရှုဘူး။ ဘိပါချိခိုင်းရင်လည်း ကားရဲလ်မှာ ဘိပါရှိမှ ချိခိုင်းလို့ရတယ်။ ကားရဲလ်ကို ဘိပါတွေ လိုသလောက် ဖြည့်ပေးထားတယ်လို့ ယူဆပါ။ turn\_left က ‘ဘယ်လှည့်’ ခိုင်းတာ။

### ဘိပါကို ဘယ်လိုရွှေ့ခိုင်းမလဲ

ပုံ (၁.၁) အနေအထားကနေ ရေ့ကို သုံးနေရာရွှေ့ ဘိပါကောက်၊ ဘယ်ဘက်လှည့်၊ အပေါ် နှစ်နေရာရွှေ့ ညာဘက်လှည့်၊ ရှေ့တစ်နေရာထပ်ရွှေ့ပြီး ဘိပါချုထားခိုင်းလိုက်ရင် အလုပ်ပြီးသွားပါပြီ။

ကားရဲလ်ကို ညာဘက်လှည့်ခိုင်းဖို့ turn\_right ကွန်မန်း မရှိဘူး။ ဒါပေမဲ့ ဘယ်သုံးခါလှည့်တို့ရတယ်။ ဒါကြောင့် ညာဘက်ချင်တဲ့အခါ ဘယ်သုံးခါလှည့်ခိုင်းလို့ရတယ်။

## ၁.၂ Meet Karel ပရိဂရမ်

ပရိဂရမ် ရေးတယ်ဆိုတာ ကွန်ပျူးတာကို ကိစ္စတစ်ခုခဲ့ ဆောက်ရွက်ပေးဖို့ ခိုင်းစေတဲ့ ညွှန်ကြားချက်တွေ ရေးတာပါပဲ။ ဒီလို ညွှန်ကြားချက်တွေကို ပရိဂရမ်ကုဒ် (program code) လို ခေါ်တယ်။ ပရိဂရမ်ကုဒ် တွေကို ကွန်ပျူးတာနားလည်တဲ့ programming language တစ်ခုခဲ့တဲ့ ရေးရတယ်။ ဒီတာအုပ်မှာ အသုံးပါမဲ့ programming language ကတေသာ Python ပါ။ Programming language တစ်မျိုးပဲ ရှိတာ မဟုတ်ပါဘူး။ ရာနဲ့ခြီးပြီး ရှိတာပါ။ လူဘာသာစကားတွေ အမျိုးမျိုးရှိသလိုပေပါ။ Python ဟာ ဒီလို ရာနဲ့ချိတဲ့ထဲက လက်ရှိအသုံးအများဆုံး ထိပ်ဆုံးဆယ်ခု ထဲမှာ ပါဝင်တယ်။ Python နဲ့ ဘိပါရွှေ့ခိုင်းတဲ့ ပရိဂရမ်ကို လေ့လာကြည့်ရအောင်။ ကားရဲလ်နဲ့ ပထမဆုံး မိတ်ဆက်ပေးတဲ့ ပရိဂရမ်မို့လို့ ဒီပရိဂရမ် နံမည်ကို ‘Meet Karel’ လို ခေါ်ပါမယ်။

```
# File: meet_karel.py
# About: This is
from stanfordkarel import *

def main():
    """Karel code goes here!"""
    move()
    move()
    move()
    pick_beeper()
    turn_left()
    move()
    move()

    turn_left()
    turn_left()
    turn_left()

    move()
    put_beeper()
# End of main

if __name__ == "__main__":
    run_karel_program("meet_karel")
```

ဒါဟာ ‘Meet Karel’ ပရိဂရမ်အတွက် Python နဲ့ရေးထားတဲ့ ပရိဂရမ်ကုဒ် တွေဖြစ်ပါတယ်။ ‘Python ကုဒ်’ လို အတိုချိုးပဲ ပြောတာများတယ်။ Python ‘စာ/စကား’ မတတ်ရင် ဒီ ‘Python ကုဒ်’ တွေကိုလည်း နားလည်မှာ မဟုတ်ဘူး။ ဒီတော့ Python ‘စာ/စကား’ အခြေခံက စုပြုး လေ့လာဖို့လိုပါမယ်။

### ကွန်းမန် (Comment)

ပထမဆုံး # သက်တနဲ့ စတဲ့ စာကြောင်းတွေက ကွန်းမန်တွေပါ။ ကွန်းမန်တွေက ကွန်ပျူးတာ ဆောင်ရွက်ပေးရမဲ့ ညွှန်ကြားချက်တွေ မဟုတ်ဘူး။ ပရိဂရမ်ကုဒ်နဲ့ ပါတ်သက်ပြီး ကုဒ် ဖတ်ရှုသူ အတွက် မှတ်ချက်

ရေးတာ သို့မဟုတ် ရှင်းပြထားတာပါ။ တနည်းအားဖြင့် ဖတ်ရှုသူ (လူ) ပရီဂရမ်မာအတွက် ရည်ရွယ်တာ။ ကွန်ပျူးတာ (စက်) အတွက် ရည်ရွယ်တာ မဟုတ်ဘူး။ ကွန်ပျူးတာက ကုဒ်ထဲက ကွန်းမန်တွေ အားလုံးကို လစ်လျှော့ရှုမှုပါ။ ဒါပေမဲ့ ပရီဂရမ်ကုဒ်ကို ဖတ်တဲ့လူ နားလည်ဖို့ အထောက်အကြူ ဖြစ်စေတဲ့အတွက် ကွန်းမန်ရေးတာကို ပေါ့ပေါ့တန်တန် အရေးမပါသလို သဘောထားလို့ မရပါဘူး။ မိမိရေးတဲ့ ကုဒ်ကို ရှင်းပြဖို့ လိုအပ်ရင် ကွန်းမန်ရေးရပါမယ်။ ရေးသင့်တဲ့ နေရာတွေကိုလည်း မကြာခင်တွေရမှုပါ။

## import စတိတ်မန်

```
from stanfordkarel import *
```

ကတော့ အင်ပိုစတိတ်မန် ဖြစ်ပါတယ်။ “stanfordkarel လိုက်ဘရီမှ အာလုံးကို ထည့်သွင်းပေးပါ” လို တောင်းဆိုတဲ့ အဓိပ္ပာယ်။ \* သက်တာကို ‘အားလုံး’ လို့ ယူဆပါ။ stanfordkarel လိုက်ဘရီမှာ ကား ရဲလ်ပရီဂရမ်အတွက် လိုအပ်တာအားလုံး ပါဝင်တယ်။ ဒီလိုက်ဘရီကို အင်ပိုလုပ်ထားမှ ကားရဲလ်ကွန်းမန်း တွေ သုံးလို့ရမှုပါ။ သီး၌ြား ကားရဲလ်ပရီဂရမ် တစ်ခုစီတိုင်းအတွက် အင်ပိုလုပ်ရမှာ ဖြစ်တယ်။

## လိုက်ဘရီ

လိုက်ဘရီ (*library*) ဆိုတာ ပညာရပ်နယ်ပယ် တစ်ခုအတွက် ရည်ရွယ်ရေးထားတဲ့ ပရီဂရမ်ကုဒ်တွေ ပါပဲ။ သချာအတွက်အချက် လိုက်ဘရီ ဂိမ်းရေးဖို့ လိုက်ဘရီ၊ 2D/3D ဂရပ်ဖစ်ဆွဲဖို့ လိုက်ဘရီ၊ အေ အိုင်အတွက် လိုက်ဘရီ စသည်ဖြင့် နယ်ပယ်အသီးသီး၊ ကိစ္စရပ်အဖို့အတွက် သက်ဆိုင်ရာ ကျမ်းကျင် ပညာရှင်တွေ ထုတ်လုပ်ဖြန့်ချီပေးထားတဲ့ လိုက်ဘရီတွေရှိတယ်။ Matrix အော်ပရေးရှင်းတွေအတွက် numpy၊ ဂရပ်ဆွဲမယ်ဆိုရင် matplotlib စတဲ့ လိုက်ဘရီတွေကို အင်ပိုလုပ် အသုံးပြုစိုင်ပါတယ်။ မေ့ ထရစ် A ကို B နဲ့ ၌ြားက်ရင် ဒီလိုပါ

```
from numpy import *
```

```
A = array([[1, 1, 2, 2],
           [2, 2, 1, 1],
           [2, 2, 1, 1]])
```

```
B = array([[2, 2],
           [2, 2],
           [1, 1],
           [2, 2]])
```

```
result = matmul(A, B)
```

```
print(result)
```

ရုလ် အခုင်းထွက်ပါမယ်။

```
[[10 10]
 [11 11]
 [11 11]]
```

ဒါကတော့ ဘားချက် အတွက် numpy နဲ့ matplotlib လိုက်ဘရီ သုံးထားတာပါ။

```
from matplotlib.pyplot import *
from numpy import *

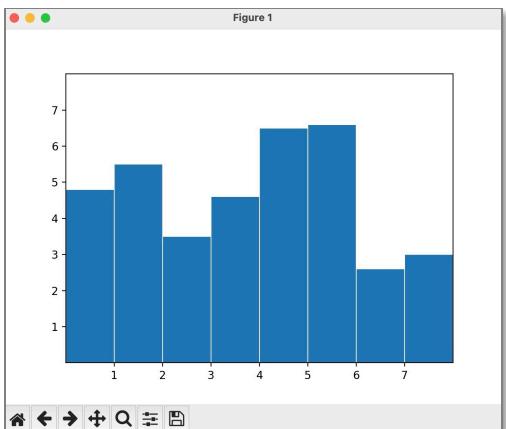
# make data:
x = 0.5 + arange(8)
y = [4.8, 5.5, 3.5, 4.6, 6.5, 6.6, 2.6, 3.0]

# plot
fig, ax = subplots()
bar(x, y, width=1, edgecolor="white", linewidth=0.7)

ax.set(xlim=(0, 8), xticks=arange(1, 8),
       ylim=(0, 8), yticks=arange(1, 8))

show()
```

ဘားချုတ်ကို ဒီလို ထတ်ပေးပါတယ်။



○ C.J

လိုက်ဘရီတွေဟာ ပရိုဂရမ်တွေ တည်ဆောက်ရာမှာ အင်မတန်မှ အရေးပါတယ်။ ဖော်ပြထားတဲ့ မူထရစ် နဲ့ ဘားချုပ် ကုဒ်တွေကို (အခုတော့) နားလည်မှာ မဟုတ်သေးဘူးပေါ့။ ဒါပေမဲ့ သက်ဆိုင်ရာ လိုက်ဘရီတွေနဲ့ ဒီလိုက်စံတွေကို သိပ်မခက်ခဲဘဲ လုပ်လို့ရနိုင်တယ်ဆိုတာ ပြုမယ် ထင်ပါတယ်။ လိုက်ဘရီတွေ သာမရှိရင် ပရိုဂရမ်တွေကို အခုထက် အဆပေါင်းများစွာ အချိန်ပေးပြီး ရှုပ်ရှုပ်ထွေးထွေး ခက်ခက်ခဲ့တည်ဆောက်ကြရမှာပါ။

## တံကါး၊ ဓတိတ်မန္ဒာန့် ဆင်းတက်ခြ

ဂျိန်ပန်စာ၊ ပြင်သစ်စာ စတဲ့ လူဘာသာစကားတွေဟာ စကားလုံးတွေ ဝါကျတွေနဲ့ ဖွဲ့စည်းထားသလို ပရီ ဂရမ်ကုဒ်တွေဟာလည်း စကားလုံးတွေ၊ ဝါကျတွေနဲ့ ဖွဲ့စည်းထားတာပါပဲ။ Programming language တွေမှာ စကားလုံးတွေကို တံ့ကင် (token) လိုပေါ်ပြီး ဝါကျတွေကိုတော့ စတိတ်မန် (statement) လို ပေါ်ပါတယ်။ ဝါကျတွေကို စကားလုံးတွေနဲ့ ဖွဲ့စည်းထားသလို စတိတ်မန်တွေကတော့ တံ့ကင်တွေနဲ့ ဖွဲ့စည်းထားတာပါ။ စတိတ်မန် ပုံစံတစ်မျိုးကို တော့ပြုပါပြီ။ အဲဒါကတော့ ရွှေစုမျက်နှာက အင်ပိစတိတ် လိုပါတယ်။

မန်ပဲ ဖြစ်ပါတယ်။

လူဘာသစကားတွေမှ ဖွဲ့စည်းတည်ဆောက်ပဲ စထရက်ချုပြုသလို programming language တွေမှာလည်း စထရက်ချုပြုဖို့ လိုအပ်တာပေါ့။ ဖွဲ့စည်းပဲ စထရက်ချုပ် မှန်/မမှန်ကို သဒ္ဓါစည်းမျဉ်တွေနဲ့ ထိန်းကွပ်ထားတာပါ။ ပရီဂရမ်ကုဒ် စထရက်ချုပ် မှန်/မမှန် ထိန်းကွပ်ပေးတဲ့ သဒ္ဓါစည်းမျဉ်တွေကိုတော့ ဆင်းတက်စ် (*syntax*) လိုခေါ်တယ်။

မြန်မာလိုရေးရင် မြန်မာသဒ္ဓါကို လိုက်နာရသလို Python နဲ့ ရေးရင် Python ဆင်းတက်စ်ကို လိုက်နာရမှာပေါ့။ မြန်မာသဒ္ဓါမှားရင် ဖတ်တဲ့သူက သည်းခံနားလည် ပေးပေမဲ့ ဆင်းတက်စ်မှားရင်တော့ Python က လုံးဝ လက်ခံမှာ မဟုတ်ပါဘူး။ ဆင်းတက်စ် စည်းကမ်းတွေဟာ ပိုပြီး တင်းကျပ်တယ်။ လွှဲချော်လို့ မရဘူး။ ဆင်းတက်စ်မှားနေတဲ့ ပရီဂရမ်ကို Python က run ခွင့် ပြုမယူတဲ့ အမှားနဲ့ သက်ခိုင်တဲ့ အယ်ရာမက်ဆွေချုပ်တွေ ပြုပေးမှာပါ။ ဖြစ်လေ့ရှိတဲ့ ဆင်းတက်စ်အမှားတွေကို မကြာခင် တွေ့ရပါမယ်။

## Keywords

`from, import, def, if` စတာတွေဟာ keyword တွေဖြစ်တယ်။ Python ရေးတဲ့အခါ သူ့နေ့ရန် သူ အဓိပ္ပာယ်ကိုယ်စိန့် အသုံးပြုရတဲ့ စကားလုံးတွေဖြစ်တယ်။ `from` နဲ့ `import` ကို လိုက်ဘရိ အင်ပူလုပ်ဖို့ သုံးတယ်။ `def` ကို ဖန်ရှင် သတ်မှတ်တဲ့အခါ သုံးတယ်။ Python က သတ်မှတ်ထားတဲ့ နေရာတွေက လွှဲလို့ အခြားကိစ္စတော့အတိုက် keyword တွေကို အသုံးပြုလို့ မရပါဘူး။ ဒါကြောင့် keyword တွေကို reserved word လိုလည်း ခေါ်တယ်။

## main ဖုန်ရှင်

‘Meet Karel’ ပရီဂရမ် အင်ပိုစတိတ်မန် အပြီးမှာ တွေ့ရတာကတော့ `main` ဖုန်ရှင်သတ်မှတ်ချက်ပါ။ ကြည့်ရအဆင်ပြအောင် သူ့ချဉ်းသီးသန် တစ်ဖြတ် ပြန်ပြပေးထားပါတယ်။

```
def main():
    """Karel code goes here!
    move()
    move()
    move()
    pick_beeper()
    turn_left()
    move()
    move()

    turn_left()
    turn_left()
    turn_left()

    move()
    put_beeper()
# End of main
```

`main` ဖုန်ရှင်သတ်မှတ်ချက်ကို အပိုင်းနှစ်ပိုင်းခဲ့ ကြည့်နိုင်တယ်။ ပထမတစ်ပိုင်း

```
def main():
```

ဖန်ရှင် (function) ဆိုတာ ကိစ္စတစ်ခု လုပ်ဆောင်ပေးဖို့အတွက် ယူနစ်တစ်ခုအနေနဲ့ ဖွံ့ဖည်းထားတဲ့ ပရီဂရမ်ကုဒ် အစုအဝေးတစ်ခုပါပဲ။ ဖန်ရှင်ကို အသုံးပြုတဲ့အခါ ငါးရဲ့ လုပ်ငန်းတာဝန်အတိုင်း ဖန်ရှင် က လုပ်ဆောင်ပေးမှာ ဖြစ်တယ်။ ဖန်ရှင်အသုံးပြုတာကို ‘ဖန်ရှင်ကောလ်’ (function call) လုပ်တာ လို့ ပြောတယ်။

ကို ဖန်ရှင်ခေါင်းစီး (function header) လို့ခေါ်တယ်။ ဖန်ရှင်ခေါင်းစီးမှာ ဖန်ရှင်နံမည်နဲ့ ဖန်ရှင်ပါရာ မီတာတွေကို ပိုက်ကွင်းထဲမှာ သတ်မှတ်ပေးရပြီး colon ‘:’ နဲ့ အဆုံးသတ်တယ်။ ဥပမာ x, y ပါရာ မီတာ နဲ့ myfun ဖန်ရှင် အတွက်

**| def myfun(x, y):**

ပါရာမီတာမပါရောင်လည်း ပိုက်ကွင်းအလွတ် တစ်စုံ ‘()’ တော့ပါရမယ်။ main ဖန်ရှင်မှာ ပါရာမီတာ ဘူး။ ပါရာမီတာတွေအကြောင်း နောက်ပိုင်းအခန်းတွေမှာ အသေးစိတ် လေ့လာရမှာပါ။ ကားရဲလ်ပရီဂရမ် တွေမှာ ပါရာမီတာအကြောင်း သိမ့်မလိုသေးပါဘူး။ ပါရာမီတာ မလိုတဲ့ ဖန်ရှင်တွေပဲ တွေ့ရမှာပါ။

ဖန်ရှင်ခေါင်းစည်းအောက် ဒုတိယပိုင်းကတော့ main ဖန်ရှင် ကုဒ်ဘလောက် (code block) ပါ။ ကုဒ်ဘလောက် ဆိုတာ ကုဒ်တွေကို အုပ်စုတစ်စုံ ဖြစ် ဖွံ့ဖည်းထားတာကို ဆုံးလိုတာပါ။ ဘလောက်လို့ လည်း ခေါ်တယ်။ ဘလောက်တစ်ခုမှာ ပါဝင်တဲ့ ကုဒ်လိုင်းတွေကို ညာဘက် ဆွဲရေးရပါမယ်။ အင်ဒန်ထုတ် (indent) လုပ်တာလို့ ခေါ်တယ်။ တက်သာကိုတစ်ချက် (သို့) စပေါ်လေးခုစာ ဆွဲလေ့ရှိတယ်။ ဘော်ဒီပထမတစ်ခြား

**| """Karel code goes here!"""**

ကို docstring လို့ ခေါ်တယ်။ quote သုံးခုတွဲ ‘“”’ တစ်စုံကြား ဉာဏ်ရေးတဲ့ ကွန်းမန်တစ်မျိုးလို့ ယူဆ နိုင်တယ်။ ဖန်ရှင်နဲ့ ပါတ်သက်တဲ့ ရှင်းလင်းဖော်ပြချက်တွေ ရေးဖို့အတွက် သုံးတာပါ။

Docstring အောက်မှာ တွေ့ရတာကတော့ ကားရဲလ်ကွန်းမန်းတွေဆိုတာ သိပါလိမ့်မယ်။ ကားရဲလ် ကွန်းမန်းတွေဟာ stanfordkarel လိုက်ဘရီ ဖန်ရှင်တွေပါ။ တနည်းအားဖြင့် stanfordkarel လိုက် ဘရီမှာ ကားရဲလ်ကွန်းမန်းတွေအတွက် ဖန်ရှင်သတ်မှတ်ချက်တွေ ပါဝင်တယ်။ ဖန်ရှင်တစ်ခုကို အသုံးပြုဖို့ အတွက် အဲဒီဖန်ရှင်ကို ခေါ်ခြာပါတယ်။ ဒါကို ဖန်ရှင်ကောလ် (function call) လုပ်တယ်လို့ ပြောတယ်။ ကားရဲလ်ကို ဘယ်ဘက်လုပ်နည့်စေချင်ရင် turn\_left ဖန်ရှင်ကောလ် လုပ်ရပါမယ်။ ဘိပါကောက်ခိုင်းချင်ရင် pick\_beeper ဖန်ရှင်ကောလ် လုပ်ရပါမယ်။ ဖန်ရှင်ကောလ် လုပ်တဲ့ ပုံစံက

**turn\_left()  
pick\_beeper()**

စသည်ဖြင့် ဖြစ်တယ်။

Python မှာ အင်ဒန်ထုတ် ဖြစ်ကတတ်ဆန်း လုပ်လိုမရဘူး။ ဘေးမျဉ်းကနေ ဆာတဲ့ အကာအဝေး မည်တဲ့ ဆင်းတက်စံအမှုး ဖြစ်တယ်။ မလိုတဲ့နေရာမှာလည်း ဆွဲရေးလို့ မရဘူး။ ခေါင်းစီးကို ဘေးမျဉ်းနဲ့ ဆာထားကြည့်ပါ။ အယ်ရာဖြစ်တာကို တွေ့ရှုရမယ်။ အင်ပိုစတိတ်မန်းလည်း ဘေးမျဉ်းနဲ့ ကွာနေ လိုမရဘူး။ အခြား language တွေမှားလည်း အင်ဒန်ထုတ်လုပ် ရေးကြပေမဲ့ Python မှာလောက် မတင်းကျပ်ဘူး။ အင်ဒန်ထုတ်မလုပ်လည်း ဆင်းတက်စံမှားတာ မဖြစ်ဘူး။

ကားရဲလ်ပရီဂရမ်တစ်ခုမှာ main ဟာ အထူးတာဝန်တစ်ခု လုပ်ဆောင်ပေးရတယ်။ အဲဒီကတော့ ပ

ရရှိရမ်ဝင်းဒီးမှာ Run Program ခလုတ် (ပုံ ၁.၅ မှာကြည့်ပါ) နှိပ်လိုက်ရင် တဲ့ပြန် လုပ်ဆောင်ပေးရတာပါ။ ဒါကြောင့် ကွန်မန်းတွေဟာ အဲဒီခလုတ် နှိပ်တော့မှပဲ စအလုပ်လုပ်တာ ဖြစ်တယ်။

### Entry Point

'Meet Karel' ပရိုဂရမ်မှာ main ဖန်ရှင်နောက် အောက်ဆုံးအပိုင်းဟာ ပရိုဂရမ် run တဲ့အခါ ပထမဆုံး စတင်လုပ်ဆောင်ပေးရမဲ့ ဖန်ရှင်ကို ဖော်ပြပေးတာပါ။ အန်ထရိုပိုင် (entry point) လိုခေါ်တယ်။

```
if __name__ == "__main__":
    run_karel_program("meet_karel")
```

run\_karel\_program ဖန်ရှင်ဟာ ကားချုပ်ရရှိရမ် တစ်ခုအတွက် အန်ထရိုပိုင် ဖြစ်တယ်။ ပရိုဂရမ် တက်လာတဲ့ တစ်ပါတ်ညံး ခေါ်တင်ချင်တဲ့ ကဲ့ကို ဒီဖန်ရှင်မှာ ထည့်ပေးတယ်။ meet\_karel.w ကဲ့ကို စစ်ဆေးခေါ်တင်ထားချင်ရင် "meet\_karel" ထည့်ပေးရမယ်။ ကဲ့ဖိုင်မရှိရင် အယ်ရာတက်ပြီး ပရိုဂရမ်ပွင့်လာမှာ မဟုတ်ဘူး။ ကဲ့မထည့်ပေးထားဘဲ ဒီလို

```
run_karel_program()
```

ဆိုရင် 8 × 8 အရွယ် default ကဲ့ကို တင်ပေးပါတယ်။

ကားခဲ့လဲကဲ့တစ်ခုကို လိုချင်တဲ့ပုံစံ ဒီဇီဝ်ဆဲပြီး ဖိုင်နဲ့ သိမ်းထားရတာပါ။ ကဲ့ပုံစံချုတဲ့ ပရိုဂရမ်လည်း ရှိတယ်။ ကဲ့ဖိုင်တွေက .w ဖိုင် အိပ်တန်းရှင်းနဲ့ ဖြစ်တယ်။ ဒီစာအုပ်မှာပါတဲ့ ဥပမာတွေ လေ့ကျင့်ခန်းတွေ အားလုံးအတွက် လိုအပ်တဲ့ ကဲ့တွေကို အဆင်သင့်ပေးထားမှာပါ။ ကိုယ့်ဟာကို လုပ်ဖို့ မလိုဘူး။ စိတ်ဝင်စားရင် စမ်းကြည့်လိုရအောင် စာမျက်နှာ (၂၂၉) နောက်ဆက်တဲ့ (က) မှာ အကျဉ်းဖော်ပြပေးထားပါတယ်။

### ၁.၃ ကားခဲ့လဲ ပရိုဂရမ် run ခြင်း

လိုအပ်တဲ့ဆော့ဖဲတွေ ထည့်သွင်းနည်းကို စာမျက်နှာ (၂၂၃) နောက်ဆက်တဲ့ (က) မှာ တစ်ဆင့်ချင်း ဖော်ပြပေးထားပါတယ်။ အခုက Python ပရိုဂရမ်တစ်ခုကို အရိုးရှင်းဆုံး (လွယ်တယ်လို့ မဆိုလို) run လိုရတဲ့ နည်းကိုဖော်ပြပေးမှာပါ။ သဘောတရားပိုင်း နားလည်ဖို့ အထောက်အပံ့ဖြစ်မယ်။ အခုနည်းလမ်းကို အကြမ်းဖျေား နားလည်အောင် ဖတ်ပြီးမှ နောက်ဆက်တဲ့ (က) ကို ဖတ်စေချင်ပါတယ်။

မိုက်ခရိုဆော့ဖဲ ဝင်းဒီးမှာ Notepad ၊ အက်ပဲနဲ့ မက်ခုအက်စဲမှာTextEdit စတဲ့ တက်စ်အယ်ဒီတာတစ်ခုခုနဲ့ ပရိုဂရမ်ကြုံရတယ်။ ကုဒ်ဖိုင်ကို .py အိပ်တန်းရှင်းနဲ့ သိမ်းရပါမယ်။ ပလိုန်းတက်စ် (plain text) ဖိုင် ပါပဲ။ Python ကုဒ်ဖိုင်ထိလို .txt အစား .py နဲ့ သိမ်းတာပါ။ Python ဖိုင်နံမည်ကို စာလုံးအသေးနဲ့ပဲ ပေးတဲ့ ထုံးစံရှိတယ်။ စပေါ်စေရာမှာ \_ (underscore) သုံးတဲ့ ထုံးစံရှိတယ်။ ဒါကြောင့် 'Meet Karel' ပရိုဂရမ်ကုဒ်ကို meet\_karel.py ဖိုင်မှာ သိမ်းသင့်တယ်။

Python နဲ့ ရေးထားတဲ့ ပရိုဂရမ်ကို run မယဆိုရင် Python ဆော့ဖဲရှိရမှာပါ။ ဒီဆော့ဖဲ အင်စတောလ် လုပ်နည်းကို နောက်ဆက်တဲ့ (က) စာမျက်နှာ (၂၆၆) မှာ ဖော်ပြပေးထားပါတယ်။ Python ကုဒ်တွေကို ကွန်ပျူးတာက တိုက်ရှိက် နားမလည်ပါဘူး။ Python ဆော့ဖဲတွေ ကုဒ်တွေကို တိုက်ရှိက် နားလည်ပြီး ကွန်ပျူးတာပေါ်မှ run လိုရအောင် ကြားခံဆောင်ရွက်ပေးတဲ့ ဆော့ဖဲလို ယေဘုယျအားဖြင့် ယူဆနိုင်တယ်။

Python ထည့်ပြီးရင် stanfordkarel လိုက်ဘရှိကို အောက်ပါကွန်မန်းနဲ့ အင်စတောလ် လုပ်ရပါမယ်။ အင်တာန်က်ပေါ်ကနေ ဒေါင်းလုဒ် လုပ်ရတာမြဲလို ကွန်နှုက်ရှင်ရှိရမယ်။

```
| pip install stanfordkarel
```

ကားရဲလ်ပရိုကရမ်ား ခေါ်တင်ချင်တဲ့ ကမ္ဘာဖိုင်တွေလည်း ရှိရပါမယ်။ meet\_karel.w ကမ္ဘာဖိုင်က meet\_karel.zip ဖိုင် worlds ဖိုဒါထဲမှာ ရှိပါတယ်။ <http://tinyurl.com/3mmmm9c7j> လင့်ကနေ meet\_karel.zip ဖိုင်ကို ဒေါင်းလုံးလုပ်ပါ။ ဒီ zip ဖိုင်ထဲက worlds ဖိုကို meet\_karel.py ဖိုင်ရှိတဲ့ နေရာမှာ ကော်ပိုကူးထည့်ပါ။ ဝင်းသီးမှာ Command Prompt မက်ခံအိုအက်စမှာ Terminal ဖွင့်ပြီး cd ဂွန်မန်းနဲ့ ကုဒ်ဖိုင်ထားတဲ့ ဖိုဒါထဲကို သွားပြီး အောက်ပါအတိုင်း python ဂွန်မန်းနဲ့ ကုဒ်ဖိုင်ကို run ပေးရပါမယ် (လာမဲ့စာမျက်နှာမှာ နမူနာပြထားတာ ကြည့်ပါ)။

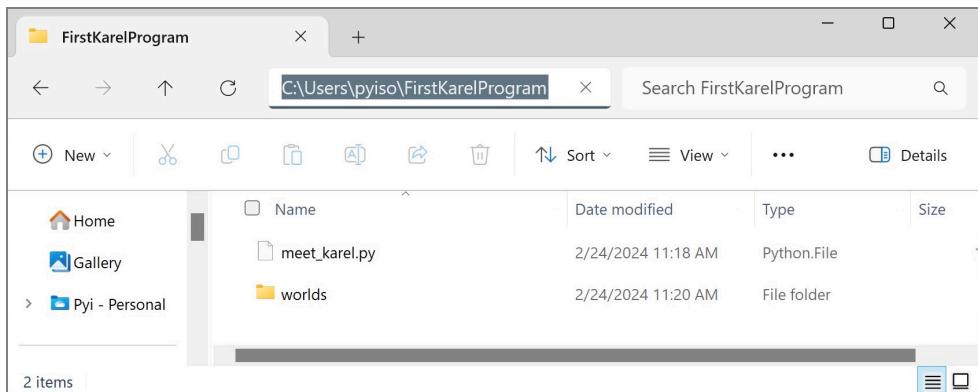
```
| python meet_karel.py
```

ကုဒ်ရေးထားတာ ဆင်းတက်စာမျက်နှာမှာ မရှိဘူးဆိုရင် ကားရဲပရိုကရမ် ပွင့်လာမှာပါ။

အခုဖော်ပြခဲ့တာက ကားရဲလ်ပရိုကရမ်တစ်ခု run ဖို့ မဖြစ်မနေလုပ်ရမဲ့ အနည်းဆုံးလိုအပ်ချက်ပါ။ Python ဆော်ဖို့ရှိရမယ်။ stanfordkarel လိုက်ဘရဲ့ အင်စတောလ် လုပ်ရမယ်။ ကမ္ဘာဖိုင်ပါတဲ့ worlds ဖိုဒါရှိရမယ်။ .py ဖိုင် တစ်ခုနဲ့ ပရိုကရမ်ကုဒ်ကို သိမ်းရမယ်။ worlds ဖိုဒါနဲ့ ကုဒ်ဖိုင်ကို တစ်နေရာ တည်းမှာ ထားရမယ်။ ပြီးရင် ဂွန်မန်းလိုင်းမှာ

```
| python your_karel_program.py
```

run ရှုပါပဲ။ C:\Users\pyiso\FirstKarelProgram ဖိုဒါထဲမှာ ကုဒ်ဖိုင်နဲ့ worlds ဖိုဒါကို ထားပြီး ဘယ်လို run ရလဲ နမူနာပြထားတာကို ကြည့်ပါ။



ပုံ ၁.၃

## ၁.၄ Python အင်စတောလ်လုပ်တဲ့အခါ ဘတွေပါလဲ

Python အင်စတောလ်လုပ်တယ်လို့ ယော်ယျု ပြောပေမဲ့ အင်စတောလ်လုပ်တဲ့အခါ တကယ်တမ်းက ပရိုကရမ်တစ်ခုတည်း ထည့်သွင်းပေးသွားတာ မဟုတ်ပါဘူး။ Python interpreter ပရိုကရမ်၊ Python standard library နဲ့ အားလုံးလိုအပ်တဲ့ ပရိုကရမ်တွေကို ထည့်ပေးသွားတာပါ။

### Python Interpreter

အင်တာပရ်ကတေသာ ဆိုတာ ပရိုကရမ်ကုဒ်တွေကို ဖတ်ပြီး ပရိုကရမ်ကုဒ်ထဲက ညွှန်ကြားချက်တွေအတိုင်း လုပ်ဆောင်ပေးတဲ့ ပရိုကရမ်ပါ။ Python အင်တာပရ်တာကတော့ Python နဲ့ရေးထားတဲ့ ကုဒ်တွေ

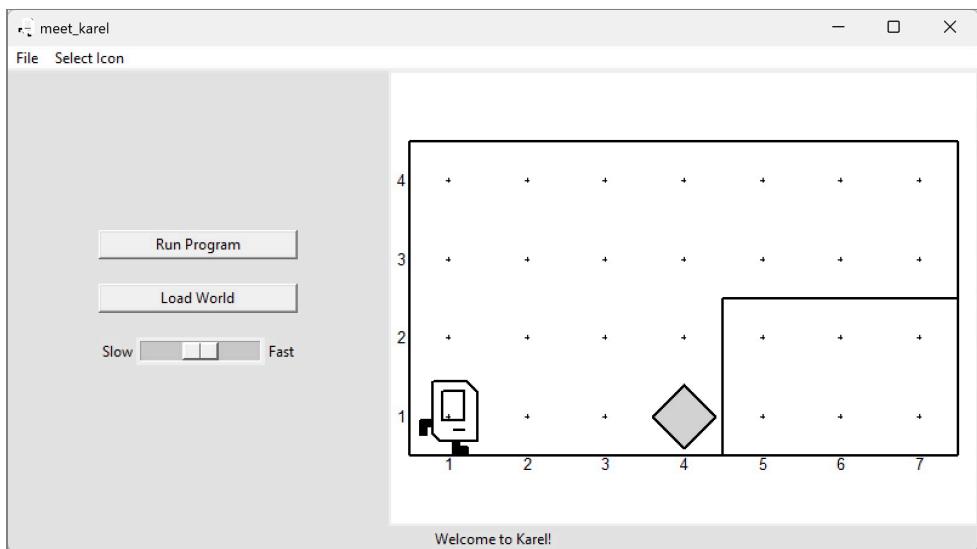
```

Command Prompt
Microsoft Windows [Version 10.0.22621.2861]
(c) Microsoft Corporation. All rights reserved.

C:\Users\pyiso>cd FirstKarelProgram
C:\Users\pyiso\FirstKarelProgram>python meet_karel.py
C:\Users\pyiso\FirstKarelProgram>

```

ပုံ ၁.၄



ပုံ ၁.၅

ကို ဖတ်နိုင်တဲ့အပြင် ညွှန်ကြားထားတဲ့အတိုင်းလည်း လုပ်ဆောင်ပေးနိုင်ပါတယ်။ Python ကုဒ်တွေကို ကွန်ပျူးတာက တိုက်ရှိက် နားလည်တာ မဟုတ်ပါဘူး။ အင်တာပရက်တာကသာ တိုက်ရှိက်နားလည်တာပါ။ ငါးက တစ်ဆင့်ခံ၍ Python ကုဒ်တွေကို ကွန်ပျူးတာပေါ်မှာ run ပေးရတာဖြစ်တယ်။ အင်တာပရက်တာကို python ကွန်မန်နဲ့ အသုံးပြု run ရပါတယ်။

`python meet_karel.py`

ဒီလို run တဲ့အခါ အင်တာပရက်တာက `meet_karel.py` ဖိုင်ထဲက ကုဒ်တွေကို ဖတ်ပြီး ပရိုဂရမ်ညွှန်ကြားချက်တွေအတိုင်း လုပ်ဆောင်ပေးတာဖြစ်ပါတယ်။

## Python Standard Library

Python အင်စတောလလုပ်တဲ့အခါ standard library လည်း တစ်ပါတည်း ထည့်သွင်းပေးသွားတယ်။ Standard library မှာ ဂွဲဗြားခြားနားတဲ့ ရည်ရွယ်ချက်အမျိုးမျိုးအတွက် ကွန်ပိုးနှုန်းတွေ အမြှော် အများ ပါဝင်ပါတယ်။ Standard library ကွန်ပိုးနှုန်းတွေကို နောက်ပိုင်းမှာ အသုံးပြုကြရမှာပါ။ ငါး

တိုကို အလျှပ်းသင့်သလို ဖော်ပြပေးသွားပါမယ်။

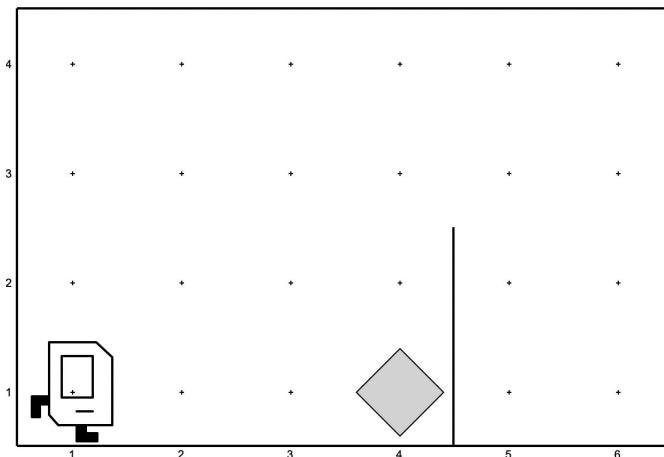
Standard အပြင် အခြားရရှိနိုင်တဲ့ လိုက်ဘရှိတွေလည်း အများအပြားရရှိပါတယ်။ pip ဟာ လိုက်ဘရှိတွေ အင်စတောလ်လုပ်ဖို့ အသုံးပြုတဲ့ ပရိုဂရမ်တစ်ခုပါပဲ။ Python documentation မှာတော့ pip ကို Python package တွေ အင်စတောလ် လုပ်ဖို့ အသုံးအများဆုံး tool (ထောက်ကူပြုပစ္စည်း) လို့ ဆိုထားပါတယ်။ Programming language တွေနဲ့ပါတ်သက်ပြီး မော်ဒူး (module)၊ ပက်ကော်ချုပ် (package)၊ လိုက်ဘရှိ (library) စတဲ့ အသုံးအနှစ်းတွေ တွေ့ရကြားရပါတယ်။ ဒီစကားလုံးတွေ အခေါ်အဝေါတွေရဲ့ အဓိပ္ပာယ်သတ်မှတ်ချက်ကလည်း သက်ဆိုက်ရာ programming language အလိုက် ကွာခြားလေ့ရှိတယ်။ ဘီက်နာအနေနဲ့ ဒါနဲ့ပါတ်သက်ပြီး သပ်ပြီးခေါင်းရှုပ်ခံစာရာ မလိုသေးပါဘူး။ ပရိုဂရမ်ကုဒ် သိမ်းဆည်း၊ ဖွဲ့စည်း၊ ထားသို့ ဖြန့်ဖြူးတဲ့ ကိစ္စတွေနဲ့ သက်ဆိုင်တဲ့ အခေါ်အဝေါတွေလို့ သိထားရင် လုံလောက်ပါပြီ။ နောက်ပိုင်းမှာ ဒီအခေါ်အဝေါတွေနဲ့ သက်ဆိုင်တဲ့ အဓိပ္ပာယ်သတ်မှတ်ချက်တွေကို အလျှပ်းသင့်ရင် သင့်သလို ဖော်ပြပေးပါမယ်။

## ၁.၅ Move Beeper to Other Side

ပရိုဂရမ်းမင်း လေ့လာတဲ့အခါ စာချည်းပဲ ဖတ်နေပြီး အမှန်တကယ် နားလည်သွားဖို့ဆိုတာ မဖြစ်နိုင်ပါဘူး။ လက်တွေ့ စမ်းသပ်ကြည့်၊ ရေးကြည့်မှုပဲ တကယ် နားလည်လာမယ်။ တကယ်လည်း ကျွမ်းကျွမ်းကျင်ကျင်ရေးတံ့လာမှုပါ။ ဒါကြောင့် လက်တွေ့ရေးကြည့်ပါ။ များများ လေ့ကျင့်ပါ။ ဥပမာတွေကိုလည်း နားလည်အောင် ဖတ်ပြီးရင် မိမိဘာသာ အလွတ် ပြန်ရေးကြည့်ပါ။

ပုံ (၁.၆) မှ ဘိပါကို နံရံအခြားတစ်ဘက် အောက်ခေါကို ရွှေပေးတဲ့ ပရိုဂရမ် ရေးကြည့်ပါ။ Python ထုံးစာရွက် move\_beeper\_to\_other\_side.py ဖိုင်နဲ့ သိမ်းသင့်ပါတယ်။ meet\_karel.zip ဖိုင်ထဲမှာပါတဲ့ worlds ဖို့မှာပဲ အခုံ ကမ္ဘာဖိုင် ထည့်ပေးထားပါတယ်။ move\_beeper\_to\_other\_side.w နံမည်နဲ့ပါ။ အနုထရီပိုင်အတွက် အခလို့ရေးရပါမယ်။

```
if __name__ == "__main__":
    run_karel_program("move_beeper_to_other_side")
```



ပုံ ၁.၆

•J

# အခန်း J

## ကားရဲလိန့် ကွန်ထရီးလ် စတိတ်မန်များ

ကားရဲလ်ပရိုကရ်တစ်ခု ဖွဲ့စည်းတည်ဆောက်ပုံကို ရှုံးအခန်းမှာ လေ့လာခဲ့ကြပြီး ကွန်ထရီးလ် စတိတ်မန် တွေ ဖြစ်ကြတဲ့ **for loop**, **while loop**, **if** နဲ့ **if...else** တို့ကို အခုဆက်လက် လေ့လာကြပါမယ်။ ပရိုကရ်တစ်ခုကို run တဲ့အခါ ပရိုကရ်ကုံးထဲက ညွှန်ကြားချက်တွေအတိုင်း ကွန်ပျူးတာက လုပ်ဆောင် ပေးတာပါ။ ဒီလိုလုပ်ဆောင်ပေးတာကို အကွွဲစီကျိုး (execute) လုပ်တယ်လို့ ခေါ်တယ်။ ကွန်ထရီးလ် စတိတ်မန်တွေဟာ ပရိုကရ်တစ်ခုရဲ့ execution flow ကို ထိန်းချုပ်ပေးတာ ဖြစ်တဲ့အတွက် control flow statements တွေလို့လည်း ခေါ်ပါတယ်။

### J.၁ for loop

ကွန်ထရီးလ်စတိတ်မန် တစ်မျိုးဖြစ်တဲ့ **for** loop ဟာ စတိတ်မန် တစ်ခု (သို့) စတိတ်မန် တစ်စုံကို သတ်မှတ်ထားတဲ့ အကြိမ်အရေအတွက် ပြည့်အောင် ထပ်ခါထပ်ခါ ပြန်ကျော်ပေးပါတယ်။ move ကို နှစ် ဆယ့်ငါးကြိမ် ကျော်ပေးဖို့ for loop နဲ့ အခုလို

```
for i in range(25):
    move()
```

ရေးရပါတယ်။ put\_beeper, move, turn\_left စတိတ်မန် သုံးကြောင်းတစ်စုံကို အကြိမ်တစ်ရာ ကျော်ချင်ရင် ဒီလိုပါ

```
for i in range(100):
    put_beeper()
    move()
    turn_left()
```

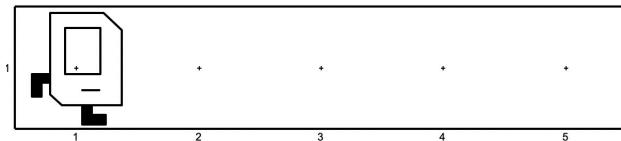
နှစ်ဆယ့်ငါးကြိမ်ကို range(25), အကြိမ်တစ်ရာကို range(100) စသည်ဖြင့် တွေ့ရတယ်။ ယေ ဘုယျအားဖြင့် အကြိမ်အရေအတွက် N ကြိမ် ကျော်ချင်ရင်

```
for x in range(N):
    statement1
    statement2
    statement3 etc.
```

ပုံစံနဲ့ သတ်မှတ်ရတာ။  $N$  က အကြိမ်အရေအတွက်ကို ဖော်ပြတဲ့ ကိန်းပြည့်ကဏ္န်းဖြစ်တယ်။ for loop ရေးတဲ့အခါ ကော်လံ : မကျွန်းခဲ့ဖို့ သတိပြုရပါမယ်။  $x$  ဟာ ပေရီရောင်တစ်ခုဖြစ်ပြီး  $i$ ,  $j$ ,  $k$  စတဲ့ အကွားရာတစ်ခုနဲ့ ကိုယ်စားပြုလေ့ရှိတယ်။

ပြန်ကျော်စေချင်တဲ့ စတိတ်မန်တွေကို for ရဲ့ ညာဘက်ကို အင်ဒွဲထုတ် လုပ်ပေးရပါမယ်။ အောက်ပါ အတိုင်းဆိုရင် put\_beeper ကိုပဲ အကြိမ်တစ်ရာ လုပ်မှာပါ။ move နဲ့ turn\_left ပြန်ကျော်မဲ့ထဲ မပါတော့ဘူး။

```
for i in range(100):
    put_beeper()
move()
turn_left()
```



ပုံ J.၁

ပုံ (J.၁) ကားရဲလ်ကဗ္ဗာမှာ ကွန်နာအားလုံးမှာ ဘိပါတစ်ခုစီ ချထားပေးရမယ်။ ကွန်နာဝါးခုရှိတာမို့ လို့ စုစုပေါင်း ဘိပါဝါးခု ချထားရမှာပါ။ move ကတော့ လေးကြိမ်ပဲ လုပ်ရမယ် (ကားရဲလ်ရှေ့မှာ ကွန်နာ လေးခုပဲ ရှိတယ်)။ ဒီအတွက်ကို အခုလိုရေးနိုင်ပါတယ်။

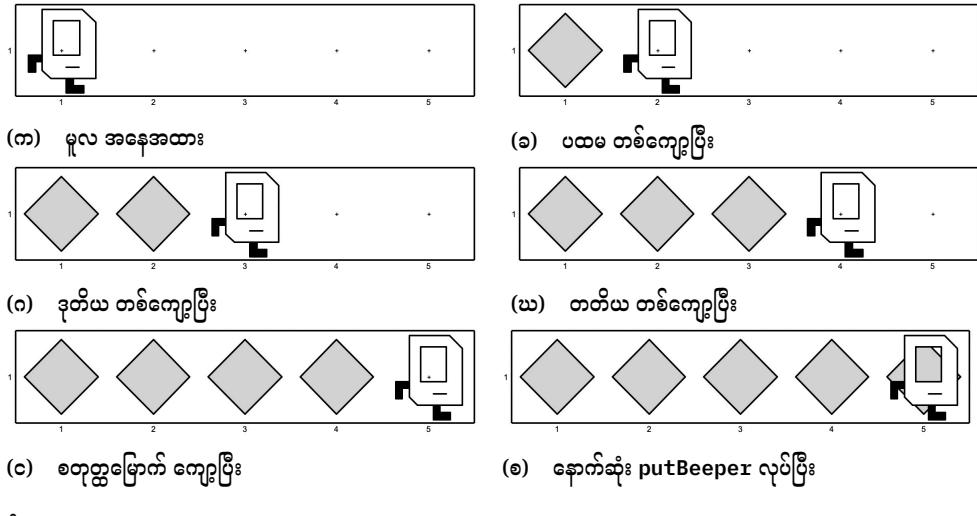
```
# File: make_row_of_5_beeper.py
from stanfordkarel import *

def main():
    for i in range(4):
        put_beeper()
        move()

    put_beeper()

if __name__ == "__main__":
    run_karel_program("make_row_of_5_beeper")
```

for loop ဟာ put\_beeper နဲ့ move ကို လေးကြိမ်ကျော်ပေးမှာပါ။ တစ်ကြိမ်ပြီးတိုင်း ရှိနေမဲ့ အနေအထား တစ်ခုချင်းစီကို ပုံ (J.၂) မှာ တွေ့နိုင်ပါတယ်။ လေးကြိမ်မြောက် နောက်ဆုံးတစ်ကျော်အပြီး ပုံ (J.၂) (c) မှာ ဘိပါတစ်ခုလုံးနေသေးတာ တွေ့ရမယ်။ for loop ပြီးတဲ့အခါ ပါပြုလော်ပေးတော့မှ တစ်တန်းလုံးအပြည့်ဖြစ်သွားမှာပါ။



ဦး J-J

### Off-by-one bug

ပြီးခဲ့သူပမာဏ နောက် အပြီး `for` loop အခြောက် မလုပ်မိရင် ပရီဂရမ်ဟာ လိုချင်တဲ့အတိုင်း မဖြစ်ပါဘူး။ ဘိပါတစ်ခုလိုနေမယ်။ ဒီလိုပြဿနာမျိုးဟာ အဖြစ်မှားတဲ့ အမှားဖြစ်ပြီး *off-by-one bug* လိုခေါ်ပါတယ်။ Loop သုံးတဲ့အခါ ကြိုရလေ့ရှိတဲ့ ဖြစ်တတ်တဲ့ အမှား (bug) ဖြစ်တယ်။ သတ်မှတ်ထားတဲ့အတိုင်း ဖြစ်သင့်တဲ့အတိုင်း ပရီဂရမ်က အလုပ်မလုပ်ဘဲ ဖြစ်နေတဲ့ အမှားကို ကွန်ပျိုးတာအသုံးအနှစ်းမှာ ပေးလိုက်တယ်။ ပေးလိုက်ရှာပြီး မှန်အောင်ပြင်ပေးတာကိုတော့ debug လုပ်တယ်လိုခေါ်ပါတယ်။

ဘိပါတစ်ခုကျွန်ခဲ့တဲ့ အမှားမျိုးဟာ off-by-one bug သာဓကတစ်ခုမျှသာ ဖြစ်တယ်။ ခုနှစ်နဲ့ ဆယ့်သုံးကြား ကေန်း ခုနှစ်လုံးရှိပါတယ် (ခုနှစ်နဲ့ ဆယ့်သုံးအပါဝင် ဆိုလျှင်)။  $13 - 7 = 6$  လိုတွေက်မိရင် တစ်လုံး လိုနေပါလိမ့်မယ်။ သုံးပေါ်ခြား တစ်တိုင် အလျားပေသုံးဆယ်ရှိ တပြောင့်တည်း ခြေစည်းရှိးတစ်ခု အတွက် တိုင်စုစုပေါင်း တစ်ဆယ့်တစ်တိုင် လိုပါမယ်။  $30 \div 3 = 10$  လို တွေက်ရင် မမှန်ပါဘူး။ တစ်ခု လိုတာ အပြင် တစ်ခုပို့နေတာလည်း ဖြစ်တတ်တယ်။ စောက် ခြေစည်းရှိးမှာပဲ ထရံအချပ်အရေအတွက် ကတော့ ကိုချပ်ဖြစ်ရမှာပါ။  $30 \div 3 = 10$  လို တွေက်ရင် တစ်ခုပို့နေပါမယ်။ ခုနှစ်နဲ့ ဆယ့်သုံးကို ထည့်မစေားစားရင် ကြားမှာ ကေန်းဝါလုံးရှိတာပါ။  $13 - 7 = 6$  လိုတွေက်မိရင် တစ်လုံး ပို့နေပါလိမ့်မယ်။ ဒီ ဥပမာ အားလုံးဟာ အခြေခံသဘောတရားအားဖြင့် သိပ်မက္ဘာခြားတဲ့ off-by-one bug ပုံစံကဲ့အမျိုးမျိုး ဖြစ်တယ်။

### ‘Make Row of Five Beepers’ ဒုတိယ ဗျားရှင်း

ပရီဂရမ်ရေးတဲ့အခါ ပရီဂရမ်မာတွေ တစ်ယောက်နဲ့တစ်ယောက် စဉ်းစားပဲ စဉ်းစားနည်း၊ ဖော်ရှင်းနည်း ထပ်တူကျလေ့မရှိပါဘူး။ ဘိပါငါးခဲ့ အတန်းလိုက် ချပေးတဲ့ ပရီဂရမ်ကိုပဲ စောက်ဗျားရှင်းတစ်မျိုးနဲ့ ရေးနိုင်ပါတယ်။

```
def main():
    put_beeper()
    for i in range(4):
        move()
        put_beeper()
```

for loop မစခင် put\_beeper လုပ်ထားတာ၊ move နဲ့ put\_beeper အစဉ်ပြောင်းသွားတာ (ပထမ ဘာရှင်းနဲ့ ပြောင်းပြန်ဖြစ်နေတာ) သတိထားကြည့်ပါ။

## J.၂ အင်ဒန်ထုတ်လုပ်ခြင်းနှင့် ကုဒ်ခွဲချက်ချုပ်

Python ဟာ အင်ဒန်ထုတ်လုပ်ခြင်းဖြင့် ပရီဂရမ်ကုဒ် ဖွဲ့စည်းပုံ စထရက်ချာကို ဖော်ပြတဲ့ programming language ဖြစ်ပါတယ်။ Python ကုဒ်တွေကို ဘလောက် (block) တွေနဲ့ ဖွဲ့စည်းထားတယ်လို့ ရှုမြင် နိုင်တယ်။ ဘလောက်ဆိုတာ ကုဒ်တွေကို အပ်စုတစ်စု အဖြစ် ဖွဲ့စည်းထားတာကို ဆိုလိုတာပါ။

```
def main():
    put_beeper()
    for i in range(4):
        move()
        put_beeper()
    pick_beeper()
turn_left()
```

အခု Python ကုဒ်မှာ အင်ဒန်ထုတ်လုပ်ထားဘဲ ဘယ်ဘက်စွန်း ကပ်ရေးထားတဲ့ def main(): (ဖန် ရှင်ခေါင်းစည်း) ဟာ အပေါ်ခံးအဆင့် (top level) ဖြစ်တယ်လို့ ယူဆတယ်။ ဖန်ရှင်ခေါင်းစည်းအောက် အင်ဒန်ထုတ်လုပ်ထားတဲ့ ကုဒ်လိုင်းအားလုံးဟာ main ဖန်ရှင် ဘလောက်ထဲမှာ အကျိုးဝင်တယ်။ pick\_beeper အထိပိုတယ်။ put\_beeper (ပထမတစ်ခု)၊ for loop နဲ့ pick\_beeper တို့ဟာ ပထမအဆင့် အင်ဒန်ထုတ်ဖြစ်တယ်။

တစ်ခါး for အောက်မှာ ဒုတိယတစ်ဆင့် အင်ဒန်ထုတ်လုပ်ထားတဲ့ ကုဒ်လိုင်းအားလုံး for ဘလောက် ထဲမှာ အကျိုးဝင်တယ်။ move နဲ့ put\_beeper ပါဝင်တယ်။ pick\_beeper ကတော့ ပထမအဆင့် ပြန် ဖြစ်သွားတဲ့အတွက် for ဘလောက်ထဲမှာမပါဘူး။

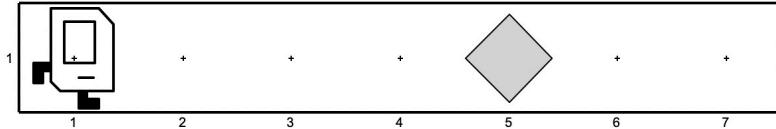
အောက်ဆုံး turn\_left() ကလည်း အင်ဒန်ထုတ်လုပ် မထားတဲ့အတွက် အပေါ်ဆုံးအဆင့်ပဲ။ def main(): နဲ့ အဆင့်တူတော်ပေါ့။ main ဖန်ရှင်ဘလောက် အပြင်ဘက်မှာလို့ ယူဆရမယ်။

အင်ဒန်ထုတ်လုပ်ထားတဲ့ အဆင့်ကို ကြည့်ပြီး ဘလောက်တွေရဲ့ ဖွဲ့စည်းပုံကို ကွက်ကွက်ကွင်းကွင်း ထင်း ကနဲ့ မြင်နိုင်တယ်။ အပေါ်ကကုန်ကို ကြည့်တာနဲ့ main ဖန်ရှင်ထဲမှာ for loop ရှိတယ်။ for loop ထဲမှာ move နဲ့ put\_beeper ပါဝင်တယ်ဆိုတာ သိသာတယ်။ pick\_beeper ဟာ for loop အပြင်မှာ၊ turn\_left ဟာ main အပြင်မှာ ဆိုတာကို အင်ဒန်ထုတ်လုပ်ထားတဲ့ အဆင့်က ဖော်ပြန်တယ်။

## J.၃ while loop

အခြေအနေတစ်ရပ် မှန်နေသူ၏ စတိတ်မန်တွေကို တစ်ကြိမ်ပြီးတစ်ကြိမ် ပြန်ကျော့ လုပ်ဆောင်စေချင် ရင် while loop ကို အသုံးပြုပါတယ်။ for နဲ့ while loop နှစ်ခုလုံးက ပြန်ကျော့ပေးတာ ဖြစ်ပေါ့။ for ကို အကြိမ်အရေအတွက် အတိအကျသိတဲ့ကိစ္စမျိုးမှာ သုံးလေ့ရှိပြီး while ကိုတော့ ဘယ်နှစ်ကြိမ် လဲ ကြိုတွက်လို့မရတဲ့ အခြေအနေ အပေါ်မှုတည်ပြီး လုပ်ဆောင်ပေးရမဲ့ အကြိမ်အရေအတွက် ကွားဤော် နိုင်တဲ့ ကိစ္စမျိုးတွေမှာ သုံးလေ့ရှိတယ်။ ဥပမာတစ်ခုနဲ့ ကြည့်ရင် ပို့နားလည်ပါလိမ့်မယ်။

(၁) လမ်းပေါ်မှာ ဘိပါတစ်ခုရှိမယ်။ ဘိပါ ဘယ်ကွန်နာမှာရှိမှာလဲ၊ လမ်းဘယ်လောက်အရှည်ဖြစ်မ လဲ ကြိုတင်မသိဘူးလို့ ယူဆပါ။ နှုန်းကဗ္ဗာ တစ်ခုကို ပုံး (J.၃) မှာ တွေ့နိုင်တယ်။ ဘိပါကို သွားပြီး ကောက်ခိုင်းရပါမယ်။ အလားတူ မည်သည့်ကဗ္ဗာအတွက်မဆို ဘိပါကောက်ပေးနိုင်ရမှာ ဖြစ်တယ်။ ဘိပါရှိ



ဗိုလ်ချုပ်

မဲ့ကွန်နာကို ကြိုတင်မသိထားတဲ့အတွက် ဘယ်နှစ်ကြိုမ် move လုပ်ခိုင်းရမှာလဲ မသိဖြစ်နေတယ်။ အကြိုမ် အရေအတွက် မသိတဲ့အတွက် for loop နဲ့ အဆင်မပြေတော့ပါဘူး။

ဘိပါမရှိသူ၏ တစ်ကြိုမ်ပြီးတစ်ကြိုမ် move လုပ်ခိုင်းမယ်ဆိုရင် နောက်ဆုံးမှာ ဘိပါရှိတဲ့ကွန်နာကို ရောက်သွားမှာ သေချာပါတယ်။ အဒီလို လုပ်ခိုင်းဖွံ့ဖြိုးရာအတွက် while loop နဲ့ ရေးထားတာကို အခုလုံးတွေရုပ်မယ်။

```
while no_beeper_present():
    move()
```

ကားရဲလ်ဟာ အခြေခံကွန်မန်း လေးခုအပြင် လက်ရှိ ကွန်နာမှာ ဘိပါရှိ/မရှိ၊ ရှေ့မှာ နံရုပ်တော်/မနေ့၊ အရှေ့သာက်ကို ပျောက်နာမှထား/မထား စတဲ့ သူနဲ့ (သို့) သူ့ချွဲကွဲနဲ့ သက်ဆိုင်တဲ့ အခြေအနေတစ်ရပ်ရပ် မှန်/မမှန် စစ်ဆိုင်ပါတယ်။ no\_beeper\_present က လက်ရှိ သူရှိနေတဲ့ ကွန်နာမှာ ‘ဘိပါ မရှိဘူးလား’ ဆိုတဲ့ အခြေအနေ စစ်ခိုင်းတာပါ။ လက်ရှိကွန်နာမှာ ဘိပါ ‘မရှိ’ ရင် ဒီအခြေအနေက မှန်တယ်လို့ ယူဆ ရမှာပေါ့။ အကယ်၍ ‘ရှိ’ နေရင်တော့ မှားတယ်လို့ ယူဆရမယ်။ အမှန် (သို့) အမှား ရလဒ်ထွက်မဲ့ ဒီလို မျိုး အခြေအနေစစ် ကွန်မန်းတွေကို ကွန်ဒီရင် (condition) လို့ခေါ်ပါတယ်။

while loop အလုပ်လုပ်ပုံက ဒီလိုပါ။ no\_beeper\_present ကွန်ဒီရင် စစ်ပါတယ်။ မှန်ရင် move လုပ်ပါတယ်။ ပြီးရင် ကွန်ဒီရင်ပြန်စစ်တယ်။ မှန်ရင် move ကို နောက်ထပ်တစ်ကြိုမ် ထပ်ကျော်ပါတယ်။ ကွန်ဒီရင်စစ်လိုက်၊ မှန်ရင် တစ်ခါထပ်ကျော်လိုက်၊ ကွန်ဒီရင်ပြန်စစ်လိုက်၊ မှန်ရင် နောက်တစ်ခါ ထပ်ကျော်လိုက်၊ ... ဒီအတိုင်းဆက်ပြီး ကွန်ဒီရင်မှန်နေသူ၏ while loop က move ကို ပြန်ကျေားပေး မှာပါ။ ကွန်ဒီရင် စစ်လိုက်လို့ မှားသွားပြီဆုံးရင်တော့ ထပ်မကျော်တော့ ရပ်သွားမှာဖြစ်တယ်။ while loop တစ်ကြိုမ်ကျော်ပြီးတိုင်း ရှိနေမဲ့ အနေအထား တစ်ခုချင်းကို ပုံ (၂.၄) မှာ လေ့လာကြည့်ပါ။ လေးကြိုမ်မြောက် ကျော်ပြီးနောက် ကွန်ဒီရင်စစ်လိုက်တဲ့အခါ မှားနေပြီ။ ဒီအခါ while loop က ထပ်မကျော် တော့ဘူး။ ဒီလို loop က ပြန်ကျော်နေတာ ရပ်သွားရင် loop ကနေ ထွက်တယ် (loop exits) လို့ ပြောလေ့ရှိတယ်။

အပြည့်အစုံ ဖော်ပြပေးထားတဲ့ ပရှိကရမ်ကို လေ့လာကြည့်ပါ။

```
# File: go_pick_beeper.py
from stanfordkarel import *

def main():
    while no_beeper_present():
        move()

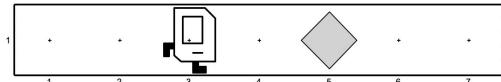
    pick_beeper()
```



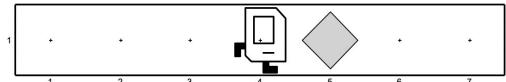
(က) မူလ အနေအထား



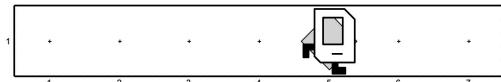
(ခ) ပထမ တစ်ကျွွှုပြီး



(ဂ) ဒုတိယ တစ်ကျွွှုပြီး



(ဃ) တတိယ တစ်ကျွွှုပြီး



(င) ဓာတ္တမြောက် ကျွွှုပြီး

ပုံ J-၄

```
if __name__ == "__main__":
    run_karel_program("go_pick_beeper")
```

### ကားရဲ့လွန်ခြင်များ

beepers\_present ဂွန်ဒါရှင် ကျတော့ လက်ရှိကွန်နာမှာ ဘိပါရိုရှင် အမှန် မရှိရှင် အမှား ရလဒ်ထွက်တယ်။ no\_b beepers\_present နဲ့ ဆန်ကျင်ဘက်ပေါ့။ ကားရဲ့လွန်ခြင်တွေအားလုံးကို တောဲလ် (J-၃) မှာ ကြည့်ပါ။ ပုံမှန်စစ်တာနဲ့ အပြင်းပုံစစ်တာ နှစ်မျိုးကို ယူလုပ်ပြတားပါတယ်။

ကွန်ချုပ်	ဆန်ကျင်ဘက် ကွန်ချုပ်	စစ်ပေးသည့် အခြေအနေ
front_is_clear	front_is_blocked	ရှေ့မှာ နံရုံကပ်လျက် ရှိမရှိ
left_is_clear	left_is_blocked	ဘယ်ဘက်မှာ နံရုံကပ်လျက် ရှိမရှိ
right_is_clear	right_is_blocked	ညာဘက်မှာ နံရုံကပ်လျက် ရှိမရှိ
beepers_present	no_b beepers_present	လက်ရှိကွန်နာမှာ ဘိပါရိုမရှိ
beepers_in_bag	no_b beepers_in_bag	ကားရဲ့လ်၏ ဘိပါအပ်ထဲ ဘိပါရိုမရှိ
facing_north	not_facing_north	အရှေ့ဘက် မျက်နှာမူလျက် ရှိမရှိ
facing_east	not_facing_east	အနောက်ဘက် မျက်နှာမူလျက် ရှိမရှိ
facing_west	not_facing_west	တောင်ဘက် မျက်နှာမူလျက် ရှိမရှိ
facing_south	not_facing_south	မြောက်ဘက် မျက်နှာမူလျက် ရှိမရှိ

တောဲလ် J-၁ ကားရဲ့လွန်ခြင်သည့် ကွန်ချုပ်များ

### while loop ဆင်းတက်စ်

while loop ရေးတဲ့ ပုံစံက ယေဘုယျအားဖြင့် ဒီလိုပါ။

```
while condition :
    statement1
```

*statement<sub>2</sub>*  
*statement<sub>3</sub>* etc.

*condition* က ကားရဲလ်ကွန်ဒါရှင် တစ်ခုဖြစ်တယ်။ ကော်လံ : မကျွန်ခဲ့အောင် ဂရုစိုက်ပါ။ *condition* မှန်နေသေးသူ၏ ပြန်ကျော့စေချင်တဲ့ စတိတ်မန်တွေကို while အောက်မှာ အင်ဒန်ထုတုလုပ်ထား ရပါမယ်။ ရှေ့မှာရင်းနေသူ၏ move လုပ်တဲ့ while loop ကို အခဲလို

```
while front_is_clear():
    move()
```

ရေးပါတယ်။

### ‘Make Beeper Row’ ဥပော

‘Make Row of Five Beepers’ ဥပမာမှာ လမ်းရဲအလျားဟာ မပြောင်းလဲဘူး ယူဆတာမို့လို့ for loop ကို အသုံးပြုတာ ဆိုလျော်ပါတယ်။ လမ်းရဲအရှည်ကို ကြိုမသိထားဘူး၊ တစ်လမ်းလုံး ဘိပါတွေ ဖြန့်ထားပေးရမယ်ဆိုရင် while နဲ့ ရေးရမှာပါ။

```

# File: make_beeper_row.py
from stanfordkarel import *

def main():
    while front_is_clear():
        put_beeper()
        move()

    put_beeper()

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

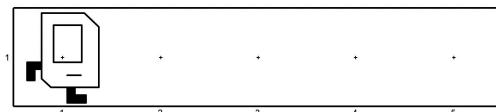
တစ်ကြိုမကျော်ပြီးတိုင်း ရှိနေမဲ့ အနေအထားကို ပုံ (၂၅) မှာ ကြည့်ပါ။ လေးကြိုမပြောက်အပြီး front\_is\_clear စစ်တဲ့အခါ မှားနေပြုဖြစ်လို့ ထပ်မကျော့တော့ဘဲ loop ကနေ ထွက်သွားမယ်။ ဒီ အခါမှာ ဘိပါတစ်ခု လိုနေသေးတယ်။ put\_beeper ထပ်လုပ်ရမယ်။ တစ်ခါပဲ လုပ်ရမှာပါ။ ဒါကြောင့် while loop ထဲမပါအောင် ပထမအဆင့် အင်ဒန်ထုပဲ ဖြစ်ရမယ်။

### J.၄ if စတိတ်မန်

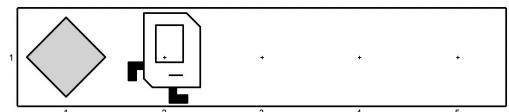
တစ်ခု (သို့) တစ်ခုထက်ပိုတဲ့ စတိတ်မန်တွေကို အခြေအနေတစ်ရပ် မှန်တော့မှုပဲ လုပ်ဆောင်စေချင်တဲ့ အခါ if ကို အသုံးပြုနိုင်တယ်။ ဥပမာ

```
if beepers_present():
    pick_beeper()
```

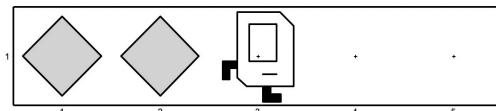
if စတိတ်မန်ဟာ beepers\_present ကွန်ဒါရှင် မှန်တော့မှုပဲ pick\_beeper လုပ်မှာပါ။ မှားရင် မ



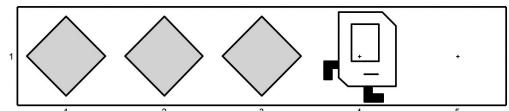
(က) မူလ အနေအထား



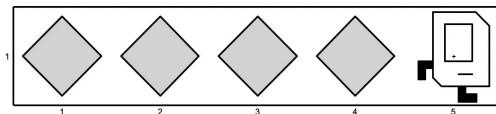
(ခ) ပထမ တစ်ကျွဲပြီး



(ဂ) ဒုတိယ တစ်ကျွဲပြီး



(ဃ) တတိယ တစ်ကျွဲပြီး

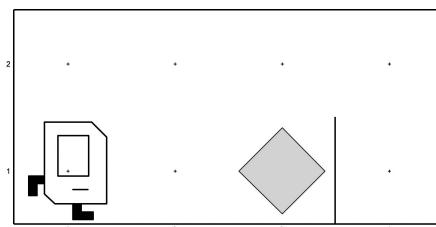


(င) စတုတ္ထပြောက် ကျွဲပြီး

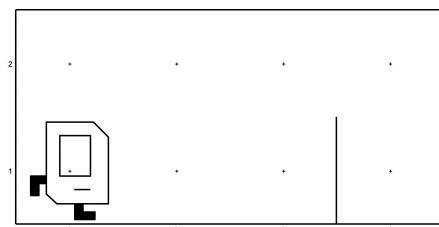
ပုံ ၂.၅

လုပ်ပါဘား။

ပုံမှာတွေရတဲ့ ကမ္ဘာနှစ်ခုထဲက တစ်ခုမှာ ကားခဲ့လုပ်ရှိနေမယ် ဆိုပါစို့။ ဘိပါရှိတဲ့ ကမ္ဘာဆိုရင် ဘိပါကို



(က)



(ခ)

ပုံ ၂.၆

နံရံအခြားဘက် အောက်ခြေကို ရွှေ့ပေးရမယ်။ မရှိတဲ့ ကမ္ဘာဆိုရင် နံရံ ဒီဘက် အောက်ခြေမှာပဲ ရပ်နေရမယ်။ ပရိုဂရမ်က ကမ္ဘာနှစ်ခုလုံးမှာ မှန်အောင် အလုပ် လုပ်နိုင်ရပါမယ်။ ဒီအတွက် if စတိတ်မန့် သုံးထားတာ လေ့လာကြည့်ပါ။

```
# File: move_beeper_to_other_side_if_any.py
from stanfordkarel import *
```

```
def main():
    move()
    move()
    if beepers_present():
        pick_beeper()
        turn_left()
        move()
```

J.၁

```
# turn_right
turn_left()
turn_left()
turn_left()
move()
# turn_right
turn_left()
turn_left()
turn_left()
move()
put_beeper()
turn_left()

if __name__ == "__main__":
    run_karel_program("mbtos1")
```

if အောက်က စတိတ်မန့်တွေကို အင်ဒန်ထဲပါ။ အဲဒီ စတိတ်မန်အားလုံး beepers\_present ဖြစ်မှ လုပ်ဆောင်မှာ ဖြစ်တယ်။ ဘိပါမရှိတဲ့ ကမ္ဘာမှာ စမ်းကြည့်ဖို့အတွက် Load World ခလုတ်နှင့်ပြီး ခေါ်တင်ပါ။ ဒါမှာဟုတ် ပရိုဂရမ်ကုံးမှာ "mbtos2" လိုပြင်ပြီး ပြန် run ပါ။  
if စတိတ်မန့် ယော့ယျာစထရက်ချောက် အခုလုံးတွေရတယ်။

```
if condition :
    statement1
    statement2
    statement3 etc.
```

condition က front\_is\_clear, beepers\_present စတဲ့ ကားရဲ့လုပ်ချက် ကွန်ဒါရှင် တစ်ခုခြားဖြစ်မယ်။  
တေဘာ် (၂.၃) မှာ ကားရဲ့လုပ်ကို စစ်ခိုင်းလိုရတဲ့ ကွန်ဒါရှင်အားလုံး ပြထားပါတယ်။

## J.၄ if...else စတိတ်မန့်

အခြေအနေတစ်ရပ် မှန်တဲ့အခါ လုပ်ဆောင်ရမှာနဲ့ မှားတဲ့အခါ လုပ်ဆောင်ရမှာ မတူကွဲပြားနေတဲ့အခါ if...else ကို သုံးပါတယ်။ ရှေ့က if စတိတ်မန့် ဥပမာ ပုံ (၂.၆) မှာ ဘိပါမရှိရင် နိုဂုံးလနေရာ ကို ပြန်လာခိုင်းချင်တယ်ဆိုပါစို့။ if...else နဲ့ အခုလုံး ရေးရပါမယ်။

```
from stanfordkarel import *
```

```
def main():
    move()
    move()
    if beepers_present():
        pick_beeper()
        turn_left()
        move()
```

```

# turn_right
turn_left()
turn_left()
turn_left()
move()
# turn_right
turn_left()
turn_left()
turn_left()
move()
put_beeper()
turn_left()

else:
    turn_left()
    turn_left()
    move()
    move()
    turn_left()
    turn_left()

if __name__ == "__main__":
    run_karel_program("mbtos2")

if...else စတိတ်မနဲ့က ဂွန်ဒီရှင်မှန်ရင် if ဘလောက်ကိုလုပ်ဆောင်ပေးပြီး ဂွန်ဒီရှင်မှားရင်
တော့ else ဘလောက်ကို လုပ်ဆောင်ပေးတာပါ။ ယေဘုယျပုံစံက ဒီလိုပါ

if condition :
    statement1a
    statement2a
    statement3a etc.

else:
    statement1b
    statement2b
    statement3b etc.

```

## J.၆ Nested ကွန်ထရီးလ် စတိတ်မန်များ

ကွန်ထရီးလ် စတိတ်မနဲ့ ဘလောက်ထဲမှာ ကွန်ထရီးလ် စတိတ်မန် ရှိလိုရတယ်။ Nested ကွန်ထရီးလ် စတိတ်မနဲ့လို့ ခေါ်ပါတယ်။

### Nested for loop

ဘိပါသုံးခုချလိုက် ရှေ့တိုးလိုက် လုပ်တာကို လေးကြိမ်ကျော်ချင်ရင် for loop နဲ့ အခုလို ရေးရမယ်ဆို တာ သိခဲ့ပြီးပါပြီ။

```
for i in range(4):
    put_beeper()
    put_beeper()
    put_beeper()
    move()
```

ဒီ for loop ထဲက put\_beeper သုံးကြောင်းကိုလည်း နောက်ထပ် for တစ်ခုနဲ့ ရေးလို ရရှိပေါ့။  
ဒီလို ဖြစ်သွားမှုပါ

```
for i in range(4):
    for j in range(3):
        put_beeper()
        move()
```

for loop နှစ်ခု ဆင့်ရေးထားလို nested for loop လိုပေါ်ပါတယ်။ Loop တစ်ခုချင်းကို ထိုးခြားရည်ရွှေ့ချင်တဲ့အခါ အပြင် for loop နဲ့ အတွင်း for loop လို ပြောလေ့ရှိတယ်။ အင်ဒန်ထဲပို့အရ အပြင် for loop ဘလောက်ထဲမှာ အတွင်း for loop နဲ့ move ပါဝင်တယ်။ အတွင်း for loop ဘလောက်ထဲမှာတော့ put\_beeper ပဲပါတယ်၊ move မပါ ပါဘူး။ အပြင် for loop မှာ ဖော်ရောကဲလို i ဆိုရင် အတွင်းမှာ i မဖြစ်ရပါဘူး။ j, ဒါမှုမဟုတ် k မတူတာ တစ်ခုခုဖြစ်ရပါမယ်။

အတန်းလိုက် ကွန်နာ ငါးခုမှာ တစ်ကွန်နာ ဘိပါ (၂၂) ခုစီ ချထားပေးဖို့ nested loop နဲ့ ရေးထားတာကို လေ့လာကြည့်ပါ။

```
# File: row_of_beeper_piles.py
from stanfordkarel import *

def main():
    for i in range(4):
        for j in range(25):
            put_beeper()
            move()

        for i in range(25):
            put_beeper()

    if __name__ == "__main__":
        run_karel_program("5x1")
```

### Nested for and while

ကမ္မာပါတ်လည် ဘိပါခြိစည်းရှိး ခတ်ပေးဖို့အတွက် for နဲ့ while nest လုပ်ထားတာကို လေ့လာကြည့်ပါ။ (၁, ၁) ကွန်နာမှာ အရောက်လည်း အနေအထားကနေ စမယ်လို ယူဆပါ။ ကမ္မာအရွယ်အစား အမျိုးမျိုးကို ခြိစည်းရှိး ခတ်ပေးနိုင်ရပါမယ်။

တောင်ဘက် နံရုတ်လျှောက် ဘိပါတွေချသွားမယ် (နောက်ဆုံး ကွန်နာမှာ ဘိပါမချဘဲ ချွှန်ထားမယ်)။  
ပြီးရင် အရော့ဘက် နံရုတ်အတွက် အဆင်သင့်ဖြစ်အောင် ဘယ်ဘက်လှည့်မယ်။ ပုံ (၂၇) (က) နဲ့ (ခ) ကို

ကြည့်ပါ။ အရှေ့ မြောက် နဲ့ အနောက်ဘက် နံရံတွေအတွက်လည်း ဒီအတိုင်း လုပ်သွားရုပါပဲ။ ပုံ (၂-၇) (က)၊ (ယ)၊ (ဇ) အသီးသီးကို ကြည့်ပါ။

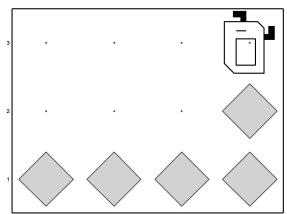
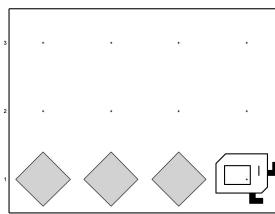
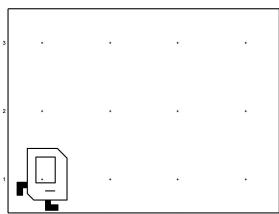
နံရံတစ်ဘက် အတွက် ဆိုရင် အခုလို

```
while front_is_clear():
    put_beeper()
    move()
    turn_left()
```

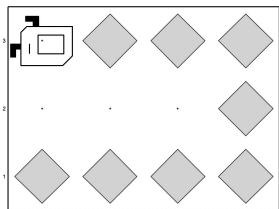
ဖြစ်မယ်။ (while ဘလောက်မှာ turn\_left မပါတာ ဂရုပြုပါ)။ နံရံလေးဘက်အတွက် လေးကြိမ် လုပ်ရမှာဆိုတော့ for နဲ့ အခုလို

```
# File: beeper_fence.py
for i in range(4):
    while front_is_clear():
        put_beeper()
        move()
        turn_left()
```

ရေးရပါမယ်။



(က) မူလ အနေအထား



(ခ) ပထမ တစ်ကျော်ပြီး

(ဂ) ဒုတိယ တစ်ကျော်ပြီး

(ယ) တတိယ တစ်ကျော်ပြီး

(ဇ) စတုထွေမြောက် ကျော်ပြီး

ပုံ ၂-၇

ပရိုကရမ် တစ်ခုလုံး အတွက်ဆိုရင် အခုလိုပါ

```
# File: beeper_fence.py
from stanfordkarel import *
```

```
def main():
    for i in range(4):
        while front_is_clear():
```

## ၂၅

```
    put_beeper()
    move()
    turn_left()

if __name__ == "__main__":
    run_karel_program("4x3")
```

### Nested while and if

while နဲ့ if nest လုပ်လည်း ရတာပေါ့။ လမ်းပေါ်မှာ ဘိပါတွေက ကြံရာကျပန်း ရှိနေမယ်။ ဘိပါတွေကို အမိုက်တွေလို ယူဆပြီး ရှင်းပေးရပါမယ်။ လမ်းအလျားကို ကြံမသိဘူး၊ ကွန်နာတစ်ခုမှာ ဘိပါတစ်ခုတက် ပိုမျို့စိန်ဘူးလို ယူဆပါ။ လမ်းအဆုံးထိ while loop နဲ့ ခွဲခိုင်းလို ရမယ်။ ရောက်တဲ့ ကွန်နာတိုင်းမှာ ဘိပါရှိရင် ကောက်ခိုင်းရပါမယ်။ if သုံးရမယ်။

```
from stanfordkarel import *

def main():
    while front_is_clear():
        if beepers_present():
            pick_beeper()
            move()

        if beepers_present():
            pick_beeper()

    if __name__ == "__main__":
        run_karel_program("clean_the_street")
```

if စတိတ်မန့် နဲ့ move ကို while loop ရဲ့ ဘလောက်ထဲမှာ ထည့်ပြီး ရှေ့မှာရှင်းနေသ၍ ပြန်ကျော့ခိုင်းထားတာပါ။ if စတိတ်မန့်က ဘိပါရှိတွေမှာပဲ ကောက်ပေးဖို့အတွက်။

ပြီးခဲ့ဟာနဲ့ အလားတူတဲ့ နောက်ထပ် ဥပမာ တစ်ခုကတွေ့ လမ်းတစ်လျှောက် ကွန်နာတွေမှာ ဘိပါရှိရင် ကောက်ခိုင်းပြီး မရှိရင် တစ်ခုချေပေးရပါမယ်။ တန်ည်းအားဖြင့် ကွန်နာတစ်ခုချင်းရဲ့ ဘိပါရှိ/မရှိ အခြေအနေကို ဆန့်ကျင်ဘက် ပြောင်းတာပေါ့။

```
# File: toggle_beeper.py
from stanfordkarel import *

def main():
    while front_is_clear():
        if beepers_present():
            pick_beeper()
        else:
```

```

        put_beeper()
        move()

    if beepers_present():
        pick_beeper()

if __name__ == "__main__":
    run_karel_program("toggle beepers")

if...else သုံးထားတယ်။ ကျွန်ုတ္တအားလုံး လမ်းရှင်းတဲ့ ပရိဂရမ်နဲ့ တူတူပဲ။

```

### Nested if

အခြေအနေ တစ်ရပ် မှန်တော့မှုပဲ လုပ်ချင်ရင် if ကို သုံးတယ်။ အခြေအနေ ‘နှစ်ရပ်’ လုံးနဲ့ ကိုက်ညီမှုလုပ်ဆောင်စေချင်တဲ့အခါ nested if သုံးနိုင်ပါတယ်။ ညာဘက်လည်းရှင်း လက်ရှိကွန်နာမှုလည်း ဘို့ပါမရှိမှ ဘိပါတစ်ခု ချထားခိုင်းမယ်ဆိုပါစို့။ အခုလိုရေးနိုင်ပါတယ်

```

if right_is_clear():
    if no beepers_present():
        put_beeper()

```

ညာဘက်မှာ ရှင်းနော့မှ အပြင် if က အတွင်း if ကို လုပ်ဆောင်စေမှုပါ။ အတွင်း if ကလည်း ဘိပါမရှိမှ ပဲ put\_beeper လုပ်မှုပါ။ ညာဘက်မှာ ပိတ်နေရင်သော်လည်းကောင်း ဘိပါရှိနေရင်သော်လည်းကောင်း pick\_beeper လုပ်မှာ မဟုတ်ပါဘူး။ right\_is\_clear နဲ့ no\_beepers\_present အခြေအနေ နှစ်ခုလုံး မှန်မှုပဲ လုပ်မှုပါ။

ပုံ (၂.၈) (က) မှ ဒုတိယ လမ်းတစ်လျှောက် လမ်းပြင်တဲ့အလုပ်ကို ကားရဲ့လိုက် တာဝန်ပေးတယ် လို ယူဆပါ။ ဘိပါရော ညာဘက်နဲ့ရုပါ မရှိတဲ့ ကွန်နာတွေက လမ်းအခြေအနေ တော်တော်ဆုံးနေတဲ့ နေရာ တွေ။ ဒီလိုနေရာတွေမှာ ဘိပါတစ်ခု ဖြော်ပြီး လမ်းပြင်ပေးရမှုပါ။

```

# File: repair_street.py
from stanfordkarel import *

def main():
    while front_is_clear():
        if right_is_clear():
            if no beepers_present():
                put_beeper()
            move()

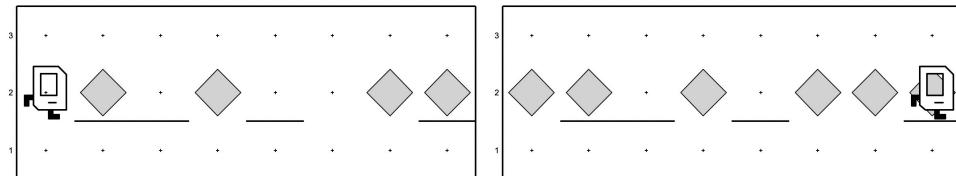
        if right_is_clear():
            if no beepers_present():
                put_beeper()

if __name__ == "__main__":

```

```
run_karel_program("repair_street")
```

while ဘလောက်ထဲမှ nested if နဲ့ move ပါဝင်တယ်။ nested if က ညာဘက်လည်းရှင်း ဘိပါလည်းမရှိမှ လက်ရှုံးကွန်နာမှာ ဘိပါတစ်ခု ချိုင်းထားတာပါ။ while ထဲမှာ if၊ အဲဒီ if ထဲမှာမှ နောက်ထပ် if တစ်ခု ဆင့်ထားတဲ့အတွက် သုံးဆင့် nest လုပ်ထားတာပါ။ အင်ဒန်ထု လုပ်ထားတော့ ဂရုစိက်ကြည့်ဖို့ လိုတယ်။ ဘယ်ဘလောက်ထဲမှာ ဘာရှိနေလဲဆိတာ မြင်အောင် လုပ်ရမှာပါ။ ကိုယ်တိုင် ရေးရင်လည်း အင်ဒန်ထု ဂရုစိက်ပြီး လုပ်ရပါမယ်။ လမ်းပြင်ပြီး အခြေအနေကို ညာဘက် ပုံ (၂.၈) (ခ) တွင် ကြည့်ပါ။



(က) မူလ အနေအထား

(ခ) လမ်းပြင်ပြီး အနေအထား

ပုံ ၂.၈



# အခန်း ၃

## ဖန်ရှင်များ (Functions)

ဖန်ရှင် (function) တွေဟာ ပရီဂရမ်းမင်းမှာ အရေးကြီးဆုံး အခြေခံသဘောတရားတစ်ခု ဖြစ်တယ်။ ဖန်ရှင်ဆိုတာ ဘာလဲ၊ ဘာကြောင့် အရေးပါရတာလဲ၊ ဖန်ရှင်တွေကို ပရီဂရမ် ဒီဇိုင်းပြုလုပ် ရေးသားတဲ့ အခါ ဘယ်လိုအသုံးချုပ်တဲ့ စတာတွေကို ဒီအခန်းမှာ လေ့လာကြပါမယ်။

### ၃.၁ ဖန်ရှင် သတ်မှတ်ခြင်း

ညာဘက် လှည့်ခိုင်းချင်တိုင်း turn\_left သုံးခါရေးနေရတာ ရေရှည်အဆင်မပြေပါဘူး။ turn\_right လိုပဲ တိုက်ရိုက် ရေးလိုရရင် ပိုပြီးတော့ အဆင်ပြေမှုပါ။ ဒီလို လိုအပ်ချက်မျိုးကို ဖြည့်ဆည်း ပေးဖို့အတွက် ဟာ ဖန်ရှင်တွေရဲ့ အဓိက ရည်ရွယ်ချက်တွေထဲက တစ်ခုဖြစ်တယ်။ turn\_right ဖန်ရှင်ကို အခုလို သတ်မှတ်နိုင်ပါတယ်။

```
def turn_right():
    turn_left()
    turn_left()
    turn_left()
```

ဒီလို သတ်မှတ်ထားပြီးရင် ညာဘက်လှည့်ချင်တဲ့အခါ

```
turn_right()
```

လို တိုက်ရိုက်ပြေလို ရသွားမှာ ဖြစ်ပါတယ်။ turn\_left သုံးခါ ရေးဖို့ မလိုတော့ပါဘူး။

ဖန်ရှင်တစ်ခု သတ်မှတ်တယ် (defining a function) ဆိုတာ ကိစ္စတစ်ခု ဖြေရှင်းဆောင်ရွက်ဖို့ အတွက် စတိတ်မန့်တွေကို ယူနစ်တစ်ခုအဖြစ် ဖွဲ့စည်းထားလိုက်တာပါပဲ။ ငှါးယူနစ်အတွက် အမည်တစ်ခု ကိုလည်း သတ်မှတ်ပေးတယ်။

ဖန်ရှင် သတ်မှတ်မယ် ဆိုရင် def keyword သုံးရပါတယ်။ အထက်ပါ turn\_right ဖန်ရှင် သတ်မှတ်ချက် (function definition) မှာ

```
def turn_right():
```

ကို ဖန်ရှင် ဟက်ဒေါ (function header) လို ဆော်တယ် (မြန်မာလို ဆိုရင်တော့ ဖန်ရှင် ခေါင်းစဉ်း ပေါ့)။ turn\_right က ဖန်ရှင် အမည်။ ပါရာမီတာပါတဲ့ ဖန်ရှင်ဆိုရင် ပိုက်ကွင်းထဲမှာ ပါရာမီတာတွေ

သတ်မှတ်ရတယ်။ ဥပမာ (x, y)။ ပါရေးမီတာ မပါရင်တော့ () ပဲဖြစ်မယ်။ ကားရဲလ်မှာ ဖန်ရှင်အားလုံးဟာ ပါရေးမီတာ မပါတွဲအတွက် () ပဲ ဖြစ်မှာပါ။ ဖန်ရှင်ဟက်ဒီ လိုင်းအဆုံးမှာ ကော်လံ ‘.’ ထည့်ပေးဖို့ လိုပါတယ်။

ဖန်ရှင် ဟက်ဒီအောက် အင်ဒန်ထုပ်ထားတဲ့ လိုင်းအားလုံးဟာ ငှင်းဖန်ရှင်နဲ့ သက်ဆိုင်တဲ့ ကုံးဘေးကဲ့တော်များ၊ အခြားအသုတေသနများ၊ ပြောပါတယ်။ အခြား turn\_right ဖန်ရှင် ဘလောက်မှာ turn\_left သုံးကြိမ်ပါတယ်။

`turn_right()`

လုပ်ခိုင်းတာက (သတ်မှတ်ထားတဲ့) ဖန်ရှင်ကို အသုံးပြုတာ ဖြစ်တယ်။ ဒီအခါမှာ ငှင်းဖန်ရှင်နဲ့ သက်ဆိုင်တဲ့ ဘလောက်ကို လုပ်ဆောင်ပေးမှာပါ။ ဖန်ရှင်ကို အသုံးပြုတာကို ဖန်ရှင်ကောလ် (function call) လုပ်တယ်လို့ ပြောပါတယ်။ (မြန်မာလိုတော့ ‘ဖန်ရှင်ခေါ်’ တယ်လို့ ပြောတာပေါ့)။

ဖန်ရှင်သတ်မှတ်တာနဲ့ ဖန်ရှင်ကောလ် လုပ်တဲ့ပုံစံကို အောက်ပါအတိုင်း ယေဘုယျအားဖြင့် တွေ့ရပါမယ်။ Python ထုံးစံအရ ဖန်ရှင်နံမည်မှာ စာလုံးအသေးကိုပဲ သုံးလေ့ရှိတယ်။ စကားလုံး နှစ်ခုနဲ့ အထက်ဆိုင် ကြားမှ underscore (\_) မြားပေးလေ့ ရှိတယ်။

```
def name_of_function():
    statement1
    statement2
    statement3 etc.
```

`name_of_function()`

### ဖန်ရှင်နံမည် အဓိပ္ပာယ် အရေးကြီးပါတယ်

တရေးတာပဲဖြစ်ဖြစ်၊ ပရိုဂရမ်ကုံး ရေးတာပဲဖြစ်ဖြစ် စိတ်ထဲ တွေးတဲ့အတိုင်း၊ စဉ်းစားတဲ့အတိုင်း ပေါ်လွှင်အောင် ဖော်ပြနိုင်တာဟာ အားသာချက်တစ်ခုပါပဲ။ ဖန်ရှင် သတ်မှတ်ထားခြင်း အားဖြင့် ညာဘက်လူည်းခိုင်းရင် turn\_right ဘိပါ နှစ်ဆယ့်ငါးခု ချရင် put\_25\_beeper တိုက်ရိုက် ဖော်ပြလို့ ရတာဟာ အရေးပါတဲ့ ကိစ္စဖြစ်ပါတယ်။ ဘာသာစကားတစ်ခုခဲ့ ဖော်ပြနိုင်စွမ်း ‘အား’ (expressive power) ကို ထပ်လောင်းအားဖြည့်ပေးတာလို့ ဆိုရမှာပါ။

ဖန်ရှင်လုပ်ဆောင်ပေးတဲ့ ကိစ္စကို သိသာမေ့မဲ့ နားလည်ရလွှာယ်မဲ့ နံမည်မျိုး ဂရုဏ်ရေးချယ်တာက လည်း အရေးပါပါတယ်။ ကားရဲလ်ပရိုဂရမ်တွေမှာ အမြိုက်ပေးခိုင်းစေတဲ့ ပုံစံနဲ့ ဖန်ရှင်နံမည်ပေးလေ့ရှိ တယ်။ ဥပမာ turn\_north, pick\_all\_beeper ။

ဖန်ရှင်သတ်မှတ်ချက်နဲ့ ဖန်ရှင်ကောလ် ဥပမာ တရာ့ကို လေ့လာကြည့်ပါ။ ဘိပါ နှစ်ဆယ့်ငါးခု ချပေးတဲ့ put\_25\_beeper ဖန်ရှင်ပါ

```
def put_25_beeper():
    for i in range(25):
        put_beeper()
```

`put_25_beeper()`

ဒါကတော့ ကွဲနှစ်နာတစ်ခုမှာ ရှိတဲ့ ဘိပါအားလုံးကောက်ပေးတဲ့ ဖန်ရှင်ဖြစ်ပါတယ်

```
def pick_all_beeper():
    while beepers_present():
        pick_beeper()
```

```
pick_all_beeper()
```

ဖန်ရှင် အသုံးပြုတဲ့အခါ ကိစ္စတစ်ခုကို ပြုပြီး အလွယ်တကူ လုပ်လို ရသွားတယ်။ ဘိပါအားလုံး ကောက် မယ်ဆိုရင် pick\_all\_beeper ဖန်ရှင်ခေါ်လိုက်ရဲ့ပဲ။ ဘိပါ နှစ်ဆယ့်ဝါးခဲ့ ချချင်ရင် put\_25\_beeper ဖန်ရှင်ခေါ်လိုက်ရဲ့ပြီ။ ဖန်ရှင်တစ်ခုကို အသုံးပြုတဲ့အခါ အဲဒီဖန်ရှင်ကို ဘယ်လိုသတ်မှတ်ထားလဲ ပြန် စဉ်းစားနေဖို့ မလိုပါဘူး။ ဥပမာ ...

အခန်း (၁) မှာ လိုက်ဘရီဆိုတာ ဘာလဲ အကျဉ်း ဖော်ပြခဲ့တယ်။ မေ့ထရှစ် A နဲ့ B မြှောက်လဒ် A × B ကို numpy လိုက်ဘရီ ဖန်ရှင် matmul နဲ့ အဖြော်ဆုံးပါတယ်။

```
result = matmul(A, B)
```

matmul ဖန်ရှင်ကို လိုက်ဘရီ ထုတ်လုပ်တဲ့ ပညာရှင်တွေက ရေးပေးထားတာ။ အဲဒီဖန်ရှင်ကို ဘယ်ပုံ ဘယ်နည်း သတ်မှတ်ထားတယ်ဆိုတာ အသုံးပြုသူအတွက် အရေးမကြီးဘူး။ မေ့ထရှစ်တွေကို ဒီဖန်ရှင်နဲ့ မြှောက်လို ရတယ်ဆိုတာ သိရင် သုံးလို ရနေတာပါပဲ။ အခြား မေ့ထရှစ် လုပ်ထုံးလုပ်နည်း ဖန်ရှင်တွေလည်း ကောင်းမာရိုက် ပါဝင်ပါတယ်။ မိမိ လုပ်ချင်တဲ့ မေ့ထရှစ် လုပ်ထုံးလုပ်နည်းအတွက် ဖန်ရှင်ကို သိရင် (လိုက်ဘရီ documentation ဖတ်ပြီး ရှာလို ရတယ်) လိုအပ်တဲ့အခါ ခေါ်သုံးလိုက်ရဲ့ပါပဲ။ ပရိုဂရမ် တွေ တည်ဆောက်ရာမှာ လိုက်ဘရီတွေ မရှိမဖြစ်လိုအပ်တယ်။ ဖန်ရှင်တွေဟာ လိုက်ဘရီတွေရဲ့ အဓိက အစိတ်အပိုင်းတွေ ပဲဖြစ်ပါတယ်။

## ဖန်ရှင် အခြေခံအုတ်ချပ်များ

pick\_all\_beeper ဖန်ရှင်ကို အခြေခံအုတ်ချပ်သဖွယ် အသုံးပြုပြီး အခြားဖန်ရှင်တွေကို သတ်မှတ်နိုင်ပါတယ်။ လမ်းတစ်လျှောက် ဘိပါ အားလုံး ရှင်းပေးမဲ့ clean\_the\_street ဖန်ရှင်ကို လေ့လာကြည့်ပါ။

```
def clean_the_street():
    while front_is_clear():
        pick_all_beeper()
        move()

    pick_all_beeper()
```

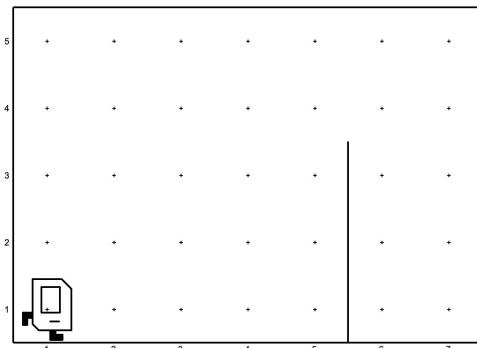
လမ်းတစ်လမ်းလုံး ရှင်းဖို့အတွက် ကွန်နာတစ်ခုက ဘိပါအားလုံး ကောက်ပေးတဲ့ pick\_all\_beeper ကို အခြေခံ အုတ်ချပ်သဖွယ် အသုံးပြုထားတာပါ။

ရိုးရှင်းတဲ့ အခြေခံ ဖန်ရှင်လေးတွေကနေ ပိုပြီး ရှုပ်ထွေးခက်ခဲတဲ့ ကိစ္စတွေ ဖြေရှင်း ဆောင်ရွက်ပေး နိုင်တဲ့ ဖန်ရှင်တွေကို တစ်ဆင့် တစ်ဆင့် တည်ဆောက်ယူလို ရတဲ့ သဘောကို တွေ့ရှုပါတယ်။ ကားရဲလ် ကမ္ဘာထဲက ရှိသမှု ဘိပါအားလုံး ရှင်းပေးမဲ့ clean\_the\_world ဖန်ရှင် သတ်မှတ်မယ် ဆိုပါစို့။ clean\_the\_street ကို အခြေခံအုတ်ချပ်သဖွယ် ဆက်လက် အသုံးပြုနိုင်မှာ ဖြစ်တယ်။

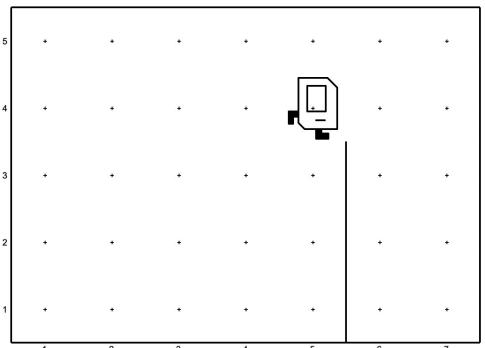
## ၃.J ပရီကွန်ဒီရှင်နှင့် ပိုစ်ကွန်ဒီရှင်

ပရီကွန်ဒီရှင် (precondition) နဲ့ (postcondition) ဟာ ဖန်ရှင်နဲ့ ပါတ်သက်ပြီး ကယာနကာ နားလည်ဖို့ လိုအပ်တဲ့ အရေးကြီးတဲ့ သဘောတရားနှစ်ခုပါ။ ဖန်ရှင် မလုပ်ဆောင်မဲ့ ကြိုတင်ရှိနေရမဲ့ အခြေအနေကို ပရီကွန်ဒီရှင်လို ခေါ်ပြီး လုပ်ဆောင်ပြီး ရှိရမဲ့ အခြေအနေကို ပိုစ်ကွန်ဒီရှင်လို ခေါ်ပါတယ်။ သတ်မှတ်ထားတဲ့ ပရီကွန်ဒီရှင်နဲ့ ကိုက်ညီမှသာ ဖန်ရှင်တစ်ခုဟာ သူလုပ်ဆောင်ပေးရမဲ့ ကိစ္စကို မှန်ကန်အောင် ဆောင်ရွက်ပေးနိုင်မှာပါ။ ဖန်ရှင် အသုံးပြုတဲ့ အခါမှာရော တည်ဆောက်တဲ့ အခါမှာပါ ပရီကွန်ဒီရှင် ပိုစ်ကွန်ဒီရှင်တွေ အပေါ် အခြေခံပြီး တိတိကျကျဖြည့်တော်းစွာ ပစာနကျပါတယ်။

ပုံ ၃.၁ (က) မှ (ခ) အနေအထားသို့ ကားရဲလ်က တိုင်ထိပ်အရောက် တက်သားရပါမယ်။ တိုင်အကွား အဝေး၊ အမြှင့် အမျိုးမျိုးနဲ့ အလားတူ ကဗ္ဗာတွေမှာလည်း အလုပ်လုပ်ရပါမယ်။ (နံရုံကို တိုင်ဟု ယူဆပါ)။ တိုင်အောက်ခြေကိုသွားတာနဲ့ တိုင်ထိပ်တက်တာ အလုပ်နှစ်ခု ပါဝင်တယ်လို့ မြင်နိုင်တယ်။ ဒီအတွက် ဖန်



(က) မတိုင်စီ အခြေအနေ



(ခ) ပြီးနောက် အခြေအနေ

ပုံ ၃.၁ တိုင်ထိပ်သို့

ရှင်နှစ်ခု သတ်မှတ်ပေးပါမယ်။

```
def go_to_pole():
    while front_is_clear():
        move()

def ascend_pole():
    while right_is_blocked():
        move()
```

အထက်ပါ ဖန်ရှင်နှစ်ခုနဲ့ go\_to\_top ကို ဆက်လက် သတ်မှတ်ပါမယ်

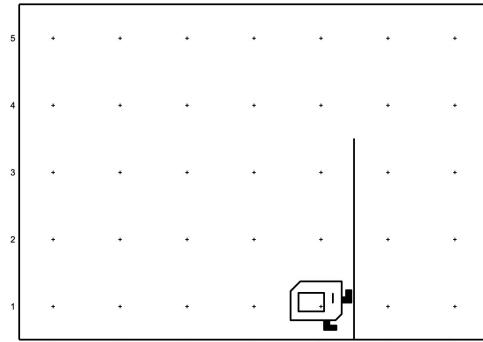
```
def go_to_top():
    go_to_pole()
    turn_left()
    ascend_pole()
    turn_right()
```

go\_to\_pole အပြီးမှာ ကားရဲလ်ဟာ တိုင်ခြေမှာ အရှေ့ဘက်မျက်နှာမှုပြီး ရှိနေမှာပါ။ ascend\_pole က တိုင်ခြေမှာ ကားရဲလ် အပေါ်ဘက်ကို မျက်နှာမှုတဲ့ အနေအထားကနေ စရပါမယ်။ go\_to\_pole ပြီး

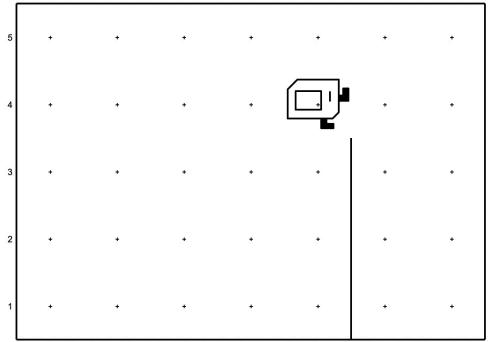
ရင် ascend\_pole အတွက် အသင့်အနေအထားဖြစ်အောင် turn\_left လုပ်ပေးရပါမယ်။

ဖန်ရှင် စတင်မလုပ်ဆောင်မဲ့ ကြိုတင်ရှိနေရမဲ့ အခြေအနေကို ပရီကွန်ဒီရှင်လို့ ပြောခဲ့ပါတယ်။ ဖန်ရှင် သတ်မှတ်တဲ့အခါ ပရီကွန်ဒီရှင်ကို တိတိကျကျ စဉ်းစားဖို့ လိုပါတယ်။ ဖန်ရှင်အသုံးပြုတဲ့အခါမှာလည်း သတ်မှတ်ထားတဲ့ ပရီကွန်ဒီရှင် အတိုင်းကိုက်ညီဖို့ လိုတယ်။ ascend\_pole ပရီကွန်ဒီရှင်းဟာ တိုင်ခြား ကားရဲလ် အပေါ်ဘက်ကို မျက်နှာမူတဲ့ အနေအထား ဖြစ်ပါတယ်။ ပုံ ၃.၂ (က) ကို ကြည့်ပါ။

ဖန်ရှင် လုပ်ဆောင်အပြီးမှာ ရှိနေရမဲ့ အခြေအနေကို ပိုစ်ကွန်ဒီရှင်လို့ ဖော်ပြခဲ့ပါတယ်။ ဖန်ရှင်တစ်ခု ဟာ ပရီကွန်ဒီရှင်နဲ့ ကိုက်ညီတဲ့ အနေအထားကနေ စတင်ရင် ပိုစ်ကွန်ဒီရှင်နဲ့ ကိုက်ညီအောင်လုပ်ဆောင်ပေးပြီး အဆုံးသတ်ရမှာပါ။ ပုံ ၃.၂ (ခ) မှာ ascend\_pole ပိုစ်ကွန်ဒီရှင်ကို တွေ့နိုင်ပါတယ်။



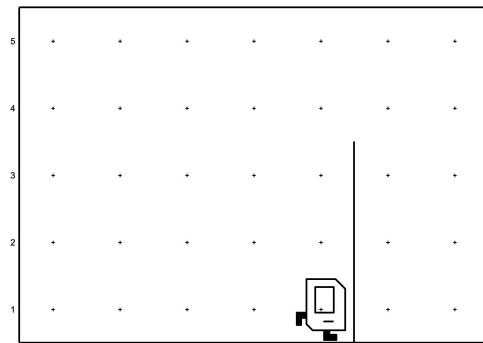
(က)



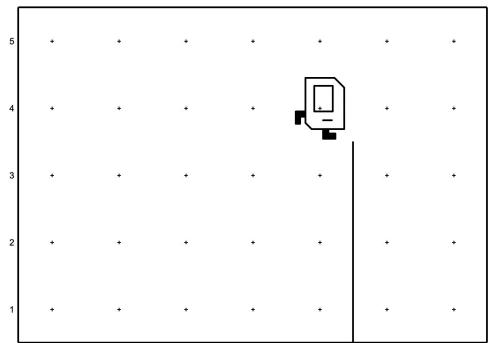
(ခ)

ပုံ ၃.၂ ascend\_pole ပရီကွန်ဒီရှင်းနှင့် ပိုစ်ကွန်ဒီရှင်း

ဖန်ရှင် ပရီကွန်ဒီရှင် ပိုစ်ကွန်ဒီရှင်ကို သင့်တော်သလို သတ်မှတ်နိုင်ပါတယ်။ ပုံ (၃.၃) တွင် ascend\_pole အတွက် အခြား ရွေးချယ်နှင့် ပရီနဲ့ ပိုစ်ကွန်ဒီရှင်ကို ကြည့်ပါ။



(က)



(ခ)

ပုံ ၃.၃ ascend\_pole ပရီနဲ့ ပိုစ်ကွန်ဒီရှင်

အထက်ပါ ပရီနဲ့ ပိုစ်ကွန်ဒီရှင်အရ ascend\_pole ကို အခဲလို

```
def ascend_pole():
    turn_left()
    while right_is_blocked():
```

```

move()
turn_right()

```

သတ်မှတ်ရမှာပါ။ go\_to\_top ကလည်း ဒီလို ဖြစ်သွားမယ်

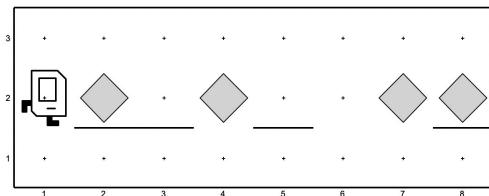
```

def go_to_top():
    go_to_pole()
    ascend_pole()

```

## ၃.၃ ဖန်ရှင်များဖြင့် abstraction များ တည်ဆောက်ခြင်း

အခန်း (၂) စာမျက်နှာ (၂၆) မှ လမ်းပြင်တဲ့ ပရိုဂါရမ်မှာ ကွန်နာတစ်ခုဟာ ဘိပါလည်းမရှိ၊ အောက်ဘက် ကပ်လျက် နံရုံလည်းမရှိရင် ဘိပါတစ်ခု ချထားပေးရပါတယ်။ ဒီကိုစွဲ ဆောင်ရွက်ပေးဖို့အတွက် ဖန်ရှင်တစ်ခု သတ်မှတ်နိုင်ပါတယ်။



ပုံ ၃၄

```

def repair_corner():
    if right_is_clear():
        if no beepers_present():
            put_beep()

```

အကြေအနေနှစ်ခုစလုံး မှန်တော့မှ ဘိပါချေပေးဖို့ nested if သုံးထားတာက နားလည်ရ အတန်အသုံး ခေါ်ခဲ့နိုင်ပါတယ်။ ဒါပေမဲ့ repair\_corner ကို အသုံးပြုတဲ့အခါ ဘယ်လိုရေးထားလဲ စဉ်းစားစရာ မလိုပါဘူး။ ဘိပါလည်းမရှိ၊ ညာဘက် နံရုံလည်းမရှိတဲ့ ကွန်နာမှာ ဘိပါချေချင်ရင် repair\_corner ဖန်ရှင်နဲ့ လုပ်လိုရတယ်ဆိုတာ သိထားရင် သုံးလိုပြု။

ဖန်ရှင် သတ်မှတ်တဲ့အခါ ဆောင်ရွက်ပေးစေခြင်း ကိစ္စကို ‘ဘယ်လို လုပ်ရမှာလ’ အသေးစိတ် စဉ်းစားရမှာဖြစ်ပေမဲ့ အသုံးပြုတဲ့အခါမှာတော့ ဒီလိုအသေးစိတ်တွေကို ထပ်ပြီး စဉ်းစားဖို့ မလိုတော့ပါဘူး။ ဖန်ရှင် ‘ဘာလုပ်ပေးတာလ’ ကပဲ အရေးကြီးတယ်။ ‘ဘယ်လို’ တည်ဆောက်ထားလဲ သိစရာမလိုဘဲ ‘ဘ’ လုပ်ပေးလဲ သိရှုနဲ့ အသုံးပြုလိုရစေတာကို abstraction လုပ်တယ်လိုခေါ်ပါတယ်။ Abstraction လုပ်ခြင်းအတွက် ဖန်ရှင်တွေဟာ အဓိက အကျဆုံး အခြေခံ အုတ်ချုပ်တွေပါပဲ။

Abstraction လုပ်ထားလိုက်ခြင်းအားဖြင့် ရွှေ့လုပ်ထွေးထွေးတွေ ထပ်ခါထပ်ခါ ခေါင်းရှုပ်ခံ စဉ်းစား နေဖို့ မလိုအပ်တော့ဘဲ ကိစ္စတစ်ခုကို အလွယ်တကူ လုပ်ဆောင်လို့ရှုံးတယ်။ ကုဒ်တွေဖတ်ရတာလည်း ပို့ရှင်းပြီး နားလည်ရ လွယ်ကူ့သွားတယ်။ ဒါကြောင့် ပရိုဂါရမ်တွေ တည်ဆောက်ရမှာ abstraction လုပ်ခြင်းဟာ အရေးပါပါတယ်။ အရေးကြီးဆုံးလို ဆိုရင်လည်း မမှားဘူး။ ကြီးမားရှုပ်ထွေးတဲ့ ဆော်ပဲတွေကို လေ့လာကြည့်ရင် abstraction ပေါင်းများစွာနဲ့ တစ်ဆင့်ပြီးတစ်ဆင့် တစ်လွှာပြီးတစ်လွှာ တည်ဆောက်ထားတယ်ဆိုတာ တွေ့ရမှာပါ။

`repair_corner` ဟာ အနိမ့်ဆုံးအလွှာက အခြေခံ abstraction တစ်ခု ဖြစ်တယ် ဆုံးပါ။ ငှုံးကို အခြေခံပြီး တစ်ဆင့်ပိုမြင့်တဲ့ အလွှာအတွက် abstraction တွေကို တည်ဆောက်ယူနိုင်ပါတယ်။ ဥပမာ

```
def repair_street():
    while front_is_clear():
        repair_corner()
        repair_corner()
```

ဒီအလွှာထက် နောက်ထပ် တစ်ဆင့်ပိုမြင့်တဲ့ abstraction တွေကိုလည်း ဆက်လက် တည်ဆောက်ယူနိုင်ပါတယ်။ ဥပမာ `repair_world` ကို တည်ဆောက်ရာမှာ `repair_street` ကို အခြေခံအစိတ်အပိုင်းဖြစ် အသုံးပြန်ပါတယ်။ ဒီလို abstraction အလွှာတွေ အဆင့်ဆင့်နဲ့ ပရိုဂရမ် တည်ဆောက်နည်းတွေကို ဆက်လက်လေ့လာကြရပါမယ်။

## ၃.၄ Bottom-up Programming

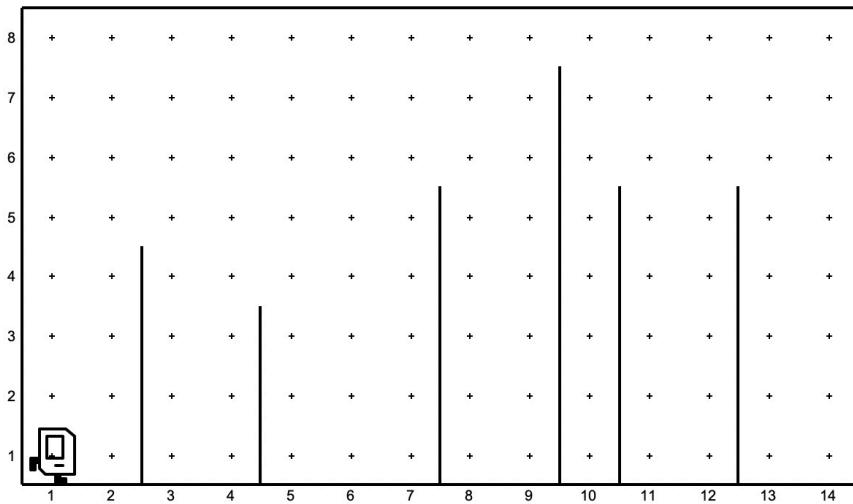
ကြီးကျယ်ခမ်းနားတဲ့ အဆောက်အအုံကြီးတွေ၊ လျှို့ဝှက်ဆန်းကြယ်တဲ့ သဘာဝဖြစ်စဉ်တွေ၊ အုံဖွယ်သိပ္ပါန်းနည်းပညာ ဆန်းသစ်တိတိစွဲမှုတွေ စတဲ့အခုန်တွေအားလုံးဟာ ရှိုးရှင်းတဲ့ အစိတ်အပိုင်းလေးတွေနဲ့ ဖွဲ့စည်းတွေကိုလည်း ထားတာပါပဲ။ ဒါကြောင့် အရွယ်အစား ကြီးမှား ရှုပ်တွေးတဲ့ ပရိုဂရမ်တွေကိုလည်း ရှိုးရှင်းတဲ့ အစိတ်အပိုင်းလေးတွေနဲ့ အခြေပြု ဖွဲ့စည်းတည်ဆောက်သင့်တယ်လို့ ယူဆမယ်ဆိုရင် ကျိုးကြောင်းဆီလျှော်တယ်ပဲ ဆိုရမှာပါ။

ခက်ခဲရှုပ်တွေးတဲ့ ကိစ္စတစ်ခုကို ဖြေရှင်းတဲ့အခါ အပိုင်းတွေခဲ့ပြီး တစ်ပိုင်းချင်းကို သီးခြားဖြေရှင်းလေ့ရှုတယ်။ အပိုင်းတွေခဲ့လိုက်ခြင်းအားဖြင့် တစ်ပိုင်းစီဟာ မူလဖြေရှင်းရမဲ့ ကိစ္စလောက် မခက်ခဲတော့ ပါဘူး။ အရွယ်အစားအားဖြင့်လည်း နိုင်တက် ငယ်သွားမယ်။ အပိုင်း တစ်ပိုင်းချင်းစီကို ဖြေရှင်းတဲ့အခါ မူလည်း အလားတူပဲ လုပ်ဆောင်ရှုနိုင်တယ်။ အပိုင်းတစ်ပိုင်းကို သူထက်သေးငယ်တဲ့ အပိုင်းတွေအဖြစ် ထပ်မံ့ထပ်တိန်းပိုမြင်ပါတယ်။ ဒီတိုင်း တစ်ဆင့်ပြီးတစ်ဆင့် လုပ်သွားမယ်ဆိုရင် နောက်ဆုံးမှာ အလွယ်တာကူ ဖြေရှင်းလို့ရတဲ့ အစိတ်အပိုင်းလေးတွေ ဖြစ်သွားရမှာပါပဲ။ *Bottom-up programming* ဆိုတာကတော့ ပရိုဂရမ်ရေ့ပြီး ကိစ္စတစ်ခုကို ဖြေရှင်းရာမှာ ရှိုးရှင်းသထက်ရှိုးရှင်း၊ သေးငယ်သထက်လေးငယ်တဲ့ အပိုင်းလေးတွေဖြစ်လာအောင် အဆင့်ဆင့်ပိုင်းခဲ့ပြီး အောက်ခြေအရှုံးရှင်းဆုံးအလွှာကနေ အပေါ်ကိုတစ်ဆင့်ချင်းချင်း တက် ဖြေရှင်းတဲ့နည်း ဖြစ်တယ်။ လေ့လာကြည့်ကြရအောင်။

ပုံ (၃.၅) ကဗျာမှာ ကားရဲလ် တန်းကော်ပြီးပြုပါမယ်။ ထောင်လိုက်နံရုံတွေကို တန်းတွေလို့ယူဆပါ။ တန်းတွေဟာ အနိမ့်အမြင် အမျိုးမျိုးဖြစ်နိုင်ပါတယ်။ ကဗျာရဲ့ အပေါ်ဘက် နံရုံကို ထိတဲ့အထိတော့ မြင်လိုမရပါဘူး။ ဒါမှ တန်းကိုကော်ပြီး အခြားဘက်ကို သွားလို့ရမှာပါ။ (1, 1) ကွန်နာကနေ တာစထွက်မှာဖြစ်ပြီး (14, 1) မှာ ပန်းဝင်ရမှာပါ။

တန်းကော်ပြီးတဲ့ ကိစ္စမှာ ပါဝင်တဲ့ အပိုင်းတစ်ခုက တန်းတစ်ခုကော်တာ။ တန်းတစ်ခု ကော်တာကို ထပ်ခဲ့ကြည့်ရင် အပေါ်တာက်တာနဲ့ အောက်ဆင်းတာ နှစ်ပိုင်းရှိုမယ်။ အခုလို နိုင်မှုလကိစ္စတစ်ခုကနေ တစ်ဆင့်တာက်တာတစ်ဆင့် ပိုးသေးငယ်တဲ့ အပိုင်းလေးတွေဖြစ်လာအောင် ခွဲထုတ်တာကို *problem decomposition* လုပ်တယ်လို့ ခေါ်ပါတယ်။

Bottom-up programming နည်းနဲ့ ပရိုဂရမ်ရေးတဲ့အခါ problem decomposition အရှင်းဆုံး လုပ်ရတယ်။ ပြီးရင် အောက်ဆုံးအလွှာမှ အသေးဆုံးအပိုင်းလေးတွေကနေ စတင်ဖြေရှင်းတယ်။ ပြီးတဲ့အခါ အဲဒီ အသေးဆုံး အပိုင်းလေးတွေကို အခြေခံအစိတ်အပိုင်းအဖြစ် အသုံးပြုပြီး အောက်ဆုံးအလွှာထက် တစ်ဆင့်မြှင့်တဲ့ အပေါ်အလွှာက အပိုင်းတွေကို ဆက်လက်ဖြေရှင်းတယ်။ ဒီအလွှာက အပိုင်းတွေကို အခြေခံအစိတ်အပိုင်းအဖြစ် အသုံးပြုပြီး နောက်ထပ်တစ်ဆင့် ပုံမြင့်တဲ့အလွှာက အပိုင်းတွေကို ဆက်လက်



ပို ၃၅

ဖြေရှင်းမှာဖြစ်တယ်။ ဒီလိုမျိုး အောက်ကနေ အပေါ်ကို တစ်လွှာပြီးတစ်လွှာ တက်သွားပြီး ဖြေရှင်းတဲ့ နည်းစနစ်ဟာ bottom-up programming ပါပဲ။

တန်းကော်ပြုပွဲမှာ Problem Decomposition လုပ်ထားတာကို တစ်လွှာချင်း ခွဲကြည့်မယ်ဆို ရင် အခုလိုတွေရှိမှာ ဖြစ်ပါတယ်။

- တန်းကော်ပြုပြုင်ပွဲဝင်ခြင်း
  - ↳ တန်းတစ်ခုကော်ခြင်း
  - ↳ အပေါ်တက်ခြင်း
  - ↳ အောက်ဆင်းခြင်း

Bottom-up နည်းအရ အောက်ဆုံးအလွှာ အပေါ်တက်၊ အောက်ဆင်း ကိစ္စကို ပထမဆုံး ဖြေရှင်းပါမယ်။ အပိုင်းတစ်ခုအတွက် ဖုန်ရှင်းတစ်ခု သတ်မှတ်ရပါမယ်။

```
def ascend():
    """
    တန်းနဲ့လွှတ်တဲ့အထိ အပေါ်ဘူး တက်သည်
    Precondition: တန်းဘယ်ဘက် ခြေရှင်းမှာ အရောက်မျက်နှာမူပြီး ရှိနေ
    Postcondition: တန်းထိပ်အပေါ် ဘယ်ဘက်ကွန်နာမှာ အရောက်မျက်နှာမူပြီး ရှိနေ
    """

    turn_left()
    while right_is_blocked():
        move()
        turn_right()

def descend():
    """
    တန်းအောက်သို့ ပြန်ဆင်းသည်
    Precondition: တန်းထိပ်အပေါ် ညာဘက်ကွန်နာမှာ အရောက်မျက်နှာမူပြီး ရှိနေ
    Postcondition: တန်းညာဘက် ခြေရှင်းမှာ အရောက်မျက်နှာမူပြီး ရှိနေ
```

```
turn_right()  
while front_is_clear():  
    move()  
turn_left()
```

အထက်ပါ ဖုန်းရှင်နှစ်ခုကို အသုံးပြုပြီး အပေါ်တစ်ဆင့် ပိုမြင့်တဲ့ အလွှာက တန်းတစ်ခုကော်တာကို ဆက်လက်ဖြေရှင်းရပါမယ်။ ascend နဲ့ descend ပရီနဲ့ ပိုစ် ကွန်ဒါရှင်တွေအရ move လုပ်ပေးဖို့လိုပါတယ်။

```
def jump_over_hurdle(
```

## တန်းတစ်ခုကို ကျော်သည်

Precondition: တန်းဘယ်ဘက် ခြေရင်းမှာ အရော့ဘက်မျက်နှာမူပြီး ရှိနေ

*Postcondition:* တန်းညာဘက် ခြေရင်းမှာ အရွှေ့ဘက်မျက်နှာမူပြီး ရှိနေ

111

ascenso

`move()`

ဒီဖန်ရှင်နဲ့ နေက်တစ်ဆင့်ကို ဆက်လက်တည်ဆောက်ရပါမယ်။ ရှေ့မျှရှင်းနေလျှင် ရှေ့တိုးမယ်၊ မရှုံးလှုံး တန်းကိုကော်ရမယ်။ ပန်းဝင်ဖိအတူက်ဆံလျှင် ဒီအလုပ်ကို ဆယ်သုံးခါလုပ်ပေးရှုပါပဲ။

```
def compete_hurdle_race():
```

ကန်းကော်မြို့ပိုင်း၊ ပန်းကုန်သာသံကြီး ပျော်ပိုင်သာသံ

*Postcondition:* (2,2) ස්ථානයෙහි පෙනෙන මුදල පෙනෙන මුදල නිසැරගැනීමේ අවස්ථාව ඇති නොවූ

103

```
for i in range(13):
```

**if** front is clear():

`move()`

also:

jump\_over\_hurdle()

ပရိုဂျမ် အစအဆုံးကို လေ့လာကြည့်ပါ။ ဖန်ရှင်တစ်ခုချင်း ပရိုနဲ့ ပို့စ် ကွန်ဒီရှင်တွေကို တိတိကျကျ မြင်အောင် ရရှိစိုက်ကြည့်ဖို့လည်း အရေးကြီးတယ်။ ဖန်ရှင် docstring မှာ ငါးငါးဖန်ရှင် ပရိုနဲ့ ပို့စ် ကွန်ဒီရှင် ကို တိတိကျကျ၊ ဖော်ပိထားတာဟာ အလေ့အထကောင်း တစ်ခုပါ။ ဖန်ရှင်တိုင်းမှာ ပါသံတယ်။

# File: hurdle\_jumping.py

```
from stanfordkarel import *
```

```
def main():
```

## **"Hurdle Jumping Program"**

compete\_hurdle\_race()

```

def ascend():
    """
    တန်းနဲ့လွှတ်တဲ့အထိ အပေါ်သို့ တက်သည်
    Precondition: တန်းဘယ်ဘက် ခြေရင်းမှာ အရောက်မျက်နှာမူပြီး ရှိနေ
    Postcondition: တန်းထိပ်အပေါ် ဘယ်ဘက်ကွန်နာမှာ အရောက်မျက်နှာမူပြီး ရှိနေ
    """

    turn_left()
    while right_is_blocked():
        move()
    turn_right()

def descend():
    """
    တန်းအောက်သို့ ပြန်ဆင်းသည်
    Precondition: တန်းထိပ်အပေါ် ညာဘက်ကွန်နာမှာ အရောက်မျက်နှာမူပြီး ရှိနေ
    Postcondition: တန်းညာဘက် ခြေရင်းမှာ အရောက်မျက်နှာမူပြီး ရှိနေ
    """

    turn_right()
    while front_is_clear():
        move()
    turn_left()

def jump_over_hurdle():
    """
    တန်းတစ်ခုကို ကျော်သည်
    Precondition: တန်းဘယ်ဘက် ခြေရင်းမှာ အရောက်မျက်နှာမူပြီး ရှိနေ
    Postcondition: တန်းညာဘက် ခြေရင်းမှာ အရောက်မျက်နှာမူပြီး ရှိနေ
    """

    ascend()
    move()
    descend()

def compete_hurdle_race():
    """
    တန်းကျော်ပြီးပြုင့်ပဲ ပန်းဝင်သည်ထိ ယဉ်ပြိုင်သည်
    Precondition: (၁,၁) ကွန်နာတွင် အရောက်မျက်နှာမူနေ
    Postcondition: (၁၄,၁) ကွန်နာတွင် အရောက်မျက်နှာမူနေ
    """

    for i in range(13):
        if front_is_clear():
            move()
        else:

```

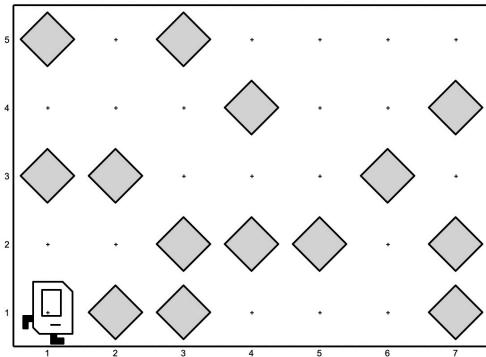
```
jump_over_hurdle()
```

```
def turn_right():
    for i in range(3):
        turn_left()

if __name__ == "__main__":
    run_karel_program("hurdle_jumping")
```

### အမှိုက်သိမ်းတဲ့ ကားရဲလ်

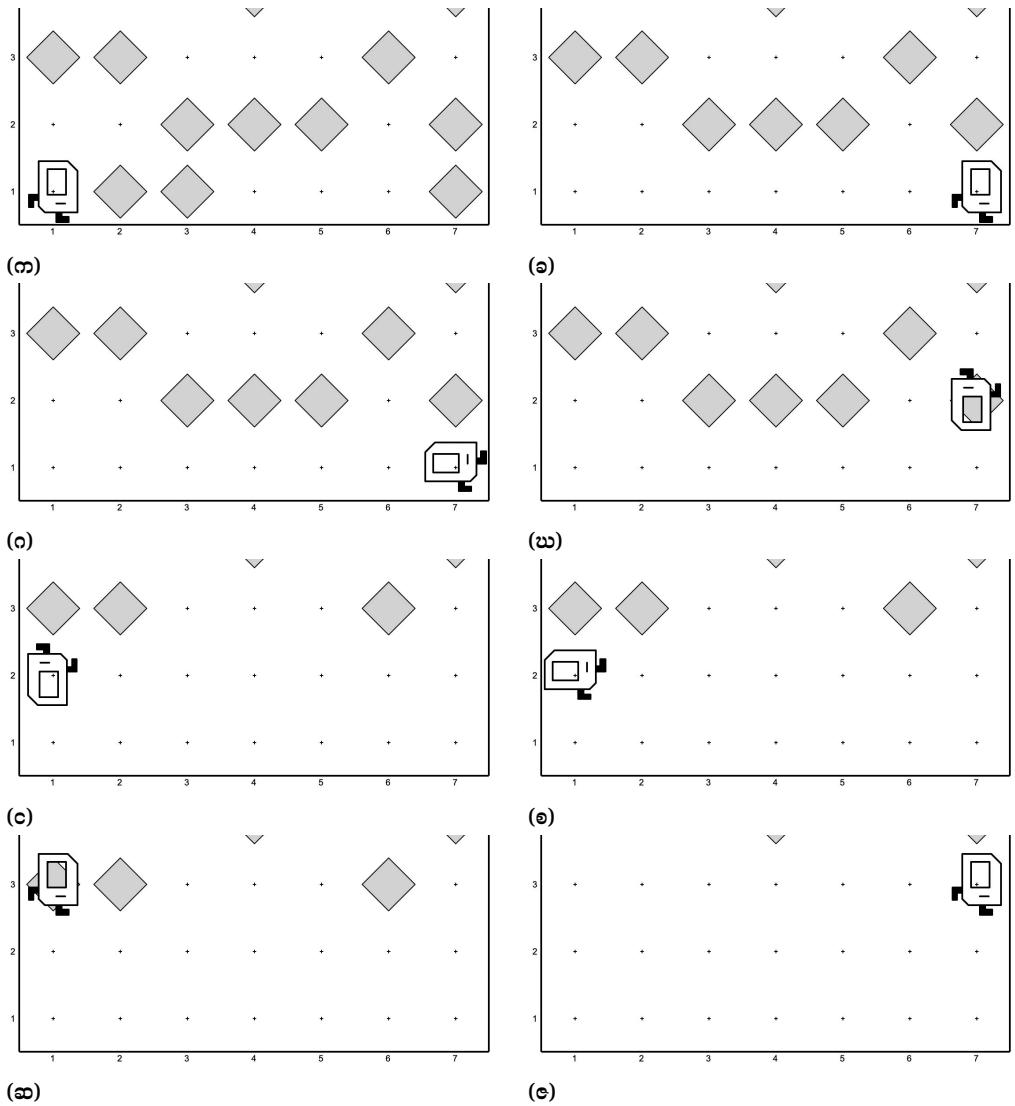
အခုံတစ်ခုမှာ ကားရဲလ်က အမှိုက်တွေရှင်းပေးရပါမယ်။ ပုံ (၃.၆) နူးနာ ကဗ္ဗာမှာ ကြံရာကျပန်း (random) ပြန့်ကျေနေတဲ့ ဘိပါတောကို အမှိုက်လို့ ယူဆပါ။ ကဗ္ဗာအရွယ်အစား အပျိုးမျိုးအတွက် အမှိုက်ရှင်းပေးတဲ့ ပရိုကရမ်တစ်ခု ရေးပေးရမှာပါ။



ပုံ ၃.၆

ဒီကိစ္စကို ဖော်ရှင်းဖို့က နည်းလမ်းတစ်ခုတည်း ရှိတာ မဟုတ်ပါဘူး။ နည်းလမ်းတစ်ခုက ဒီလိုပါ။ (၁) လမ်းကနေ စပီး ရှင်းတယ် 『စာမျက်နှာ ၄၀ ပုံ (၃.၇) (က) နှင့် (ခ) တွင် ကြည့်ပါ။』 ပြီးရင် မြောက်ဘက်လှည့်ထားမယ် ပုံ (၃.၇) (ဂ)။ ရှင်းနေသေးတယ်ဆိုရင် ဒုတိယလမ်းကို တက်၊ အနောက်ဘက် မျက်နှာမှူးထားမယ် ပုံ (၃.၇) (ယ)။ ဒုတိယလမ်းကို ဆက်ရှင်းပြီးတော့လည်း မြောက်ဘက်ကိုပဲ မျက်နှာမှူးထားပါမယ် ပုံ (၃.၇) (င)၊ (စ)။ ရှင်းနေသေးရင် တတိယလမ်းကို ကူး၊ အရှေ့ဘက်မျက်နှာ မူထားမယ် ပုံ (၃.၇) (ဆ)။ ဒီအတိုင်း တစ်လမ်းပြီးတစ်လမ်း ရှင်းသွားမှာဖြစ်တယ်။ လမ်းတစ်လမ်းရှင်းပြီး မြောက်ဘက်မျက်နှာမှူးလို့ ပိတ်နေရင် နောက်ထပ် လမ်းမရှိတော့ဘူး။ လမ်းအားလုံး ရှင်းပြီးသွားပြီ။ စာမျက်နှာ ၄၁ ပုံ (၃.၈) (က)၊ (ခ)၊ (ဂ) တို့ကို ကြည့်ပါ။ Problem decomposition လုပ်ကြည့်ရင် အောက်ပါ အတိုင်း တော်းစွဲပါတယ်။

- ကဗ္ဗာတစ်ခုလုံး အမှိုက်တွေရှင်း (clean world)
  - ▶ လမ်းတစ်လမ်း အမှိုက်ရှင်း (clean street)
    - ▶ ကွန်နာတစ်ခု အမှိုက်ရှင်း (clean corner)
    - ▶ မြောက်ဘက်လှည့် (turn north)
    - ▶ လမ်းပြောင်း/နောက်တစ်လမ်းကူး (change street)

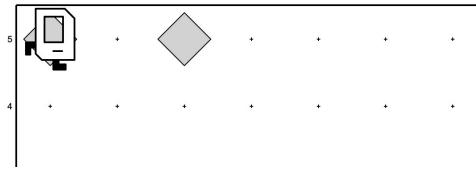


ပုံ ၃၇

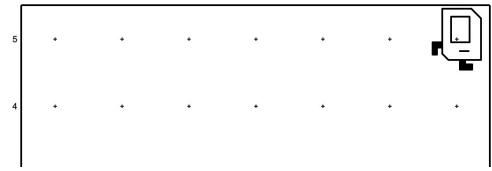
လမ်းတစ်လမ်း ရှင်းတဲ့ ကိစ္စကို ထပ်ခဲ့ကြည့်ရင် ကွန်နာတစ်ခုရှင်းတာကို တွေ့ရမှာပါ။ Bottom-up နည်းအရ အောက်ဆုံးအလွှာက ဂွဲန်နာတစ်ခု အမိုက်ရှင်းတဲ့ ကိစ္စအတွက် ဖန်ရှင်ကို ပထမဆုံး သတ်မှတ် ရပါမယ်။

```
def clean_corner():
    """
    ဂွဲန်နာတစ်ခုမှာ ဘိပါရှင်းပေးသည်
    Precondition: ဂွဲန်နာတစ်ခုမှာ ကားရဲ့လ် ရှိနေမယ်။ အများဆုံး ဘိပါတစ်ခု ရှိနိုင်တယ်။
    Postcondition: ဘိပါ မရှိတဲ့ ဂွဲန်နာဖြစ်ရမယ်
    """

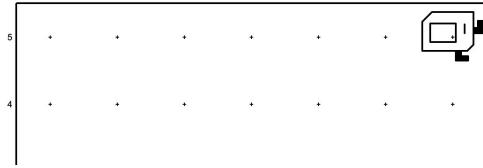
    if beepers_present():
        pick_beeper()
```



(က)



(ခ)



(ဂ)

### ပုံ ၃.၈

`clean_corner` နဲ့ လမ်းတစ်လမ်းရှင်းတဲ့ ဖန်ရှင် သတ်မှတ်ပါမယ်။

```
def clean_street():
    """
    လမ်းတလျောက် ဘပါအားလုံးရှင်းပေးသည်
    Precondition: လမ်းအစွန်းတစ်ဖက်မှာ အခြားဘက်စွန်းကို မျက်နှာမှု၏ ရီမယ်
    Postcondition: လမ်းပေါ်ဘပါအားလုံး ရှင်းပြီး အခြားဘက်စွန်း ရောက်နေမယ်
    """

    while front_is_clear():
        clean_corner()
        move()
        clean_corner()
```

မြောက်ဘက်လှည့်တဲ့ ဖန်ရှင်

```
def turn_north():
    while not_facing_north():
        turn_left()
```

while loop နဲ့ မြောက်ဘက်ကို မျက်နှာမှ မနေသော် ဘယ်ဘက်လှည့်ခိုင်းထားတာ သတိထားကြည့်ပါ။  
မြောက်ဘက် မျက်နှာမှုနေတဲ့ အနေအထားမှာ ရပ်သွားမှာ ဖြစ်တယ်။

လမ်းတစ်လမ်းရှင်းပြီး နောက်တစ်လမ်းကူးတဲ့ ဖန်ရှင်

```
def change_street():
    """
    လမ်းတစ်လမ်း၏ အစွန်းတစ်ဖက်မှာ အပေါ်လမ်းသို့ ကူးသည်
    Precondition: လမ်းအစွန်းတစ်ဖက်မှာ ကပ်လျက် နံရုံကို မျက်နှာမှု၏ ရီမယ်
    Postcondition: အပေါ်တစ်လမ်းကူးပြီး အခြားဘက်စွန်းကို မျက်နှာမှု၏ ရီမယ်
    """

    move()
    if right_is_blocked():
        turn_left()
```

```

    else:
        turn_right()

```

တစ်လမ်းချင်း အမိုက်အကုန် ရှင်းတဲ့ ဖန်ရှင်းကို သတ်မှတ်လို့ ရပါပြီ

```

def clean_world():
    """
    လမ်းအားလုံးမှ ဘိပါတွေ ရှင်းပေး
    Precondition: (၁,၁) ဤနှစ်နာမာ အကျဉ်းသမဂ္ဂ မျက်နှာမူ၍ ရှိနေမယ်
    Postcondition: လမ်းအားလုံး အမိုက်ရှင်းပြီး ဖြစ်မယ်
    """

    clean_street()
    turn_north()
    while front_is_clear():
        change_street()
        clean_street()
        turn_north()

```

ပရိုဂရမ် အစအဆုံး ဆက်လက်လေ့လာကြည့်ပါ။ (ပရိုနဲ့ ပိုစ် ကွန်ဒီရှင်တွေကို ချုန်ထားတယ်။ နေရာ သက်သာအောင်ပါ။ အပြည့်အစုံ ရေးတဲ့အခါ မလိုတော့တာ မဟုတ်ဘူး။)

```

# File: clean_world.py
from stanfordkarel import *

def main():
    """Karel clean the world"""
    clean_world()

```

```

def clean_world():
    clean_street()
    turn_north()
    while front_is_clear():
        change_street()
        clean_street()
        turn_north()

```

```

def clean_corner():
    if beepers_present():
        pick_beeper()

```

```

def clean_street():
    while front_is_clear():
        clean_corner()

```

```

        move()
        clean_corner()

def turn_north():
    while not_facing_north():
        turn_left()

def change_street():
    move()
    if right_is_blocked():
        turn_left()
    else:
        turn_right()

def turn_right():
    for i in range(3):
        turn_left()

if __name__ == "__main__":
    run_karel_program("clean_world")

```

## ၃.၅ Top-down Programming

*Top-down programming* ဟာ အပေါ်ဆုံးအလွှာမှ စ၍ တည်ဆောက်တဲ့နည်း ဖြစ်ပါတယ်။ Bottom-up မှာလို အောက်မှ အထက် မတက်ဘဲ အထက်မှအောက် တစ်လွှာပြီးတစ်လွှာ အဆင့်ဆင့် တည်ဆောက် သွားမှာပါ။ နည်းလမ်းနှစ်ခုလုံးမှာ problem decomposition လုပ်ရမှာဖြစ်ပေမဲ့ လုပ်ငန်းစဉ် ကွားမြှုပ်နည်းများမှာလို ချက်ရှိတယ်။ Top-down နည်းအတွက် problem decomposition လုပ်တဲ့အခါ bottom-up မှာလို အောက်ဆုံးအလွှာထိ တစ်ခါတည်း ခွဲခြမ်းစိတ်ဖြာစရာမလိုဘူး။ တစ်ဆင့်ချင်းပဲ ခွဲရပါတယ်။ ပြီးခဲ့တဲ့ ဥပမာ အမှိုက်သိမ်းတဲ့ ကိစ္စကို ခွဲခြမ်းစိတ်ဖြာမယ်ဆိုရင် အခုလို

- ကဗျာတစ်ခုလုံး အမှိုက်တွေရှင်း (clean world)
- ▶ လမ်းတစ်လမ်း အမှိုက်ရှင်း (clean street)
- ▶ ဖြောက်ဘက်လှည့် (turn north)
- ▶ လမ်းပြောင်း/နောက်တစ်လမ်းကူး (change street)

ဖြစ်မှာပါ။ စာမျက်နှာ (၃၉) က ခွဲခြမ်းစိတ်ဖြာတော့ တူဗ္ဗယ်ဆိုပေမဲ့ သတိပြုရမှာက လမ်းတစ်လမ်း ရှင်းတဲ့ ကိစ္စကို လောလေဆယ် ထပ်ပြီး အသေးစိတ် မခွဲသေးဘူး။ ဒီအဆင့်မှာပဲ ပုံပေါ်ထားတယ်။

ပြီးတဲ့အခါ အပေါ်ဆုံး အဆင့်ကို စပြီးဖြောင်းတယ်။ Top-down နည်းရဲ့ အဓိက ထူးခြားချက်က လက်ရှိဖြောင်းမဲ့ ကိစ္စရဲ့ အောက်အလွှာကို ‘ဖြောင်းနိုင်ပြီး’ ဖြစ်တယ်လို့ မှတ်ယူရတာပါ။ တစ်နည်းအားဖြင့် clean\_world ဖန်ရှင်း သတ်မှတ်တဲ့အခါ clean\_street, turn\_north, change\_street ဖန်ရှင်းတွေ ရှိထားပြီးဖြစ်တယ်လို့ ယူဆရ

တာဖြစ်တယ်။

```
def clean_world():
    clean_street()
    turn_north()
    while front_is_clear():
        change_street()
        clean_street()
        turn_north()
```

ဒီနေရာမှာ ပရဲနဲ့ ပိုစ် ကွန်ဒါရှင်တွေ တိတိကျကျ သတ်မှတ်ထားဖို့ အလုန်အရေးကြီးပါတယ်။ `clean_street`, `turn_north`, `change_street` ဖန်ရှင်တစ်ချင်းခဲ့ ပရဲနဲ့ ပိုစ် ကွန်ဒါရှင်တွေ တိတိကျကျ ရှိထား မှပဲ `clean_world` ကို မှန်ကန်အောင် ရေးဖို့ ဖြစ်နိုင်မှာပါ။ ဥပမာအားဖြင့် `clean_street` ပိုစ်ကွန်ဒါရှင်ကသာ မြောက်ဘက်လှည့် အနေအထားနဲ့ အဆုံးသတ်မယ်ဆိုရင် `clean_world` ကို အခုလို ရေးပါ လိမ့်မယ်။

```
def clean_world():
    clean_street()
    while front_is_clear():
        change_street()
        clean_street()
```

အပေါ်ဆုံးအလွှာ ဖြေရှင်းပြီးသွားရင် အောက်အလွှာကို တစ်ဆင့် ဆင်းပြီး ဖြေရှင်းပါတယ်။ အောက်ပါ ကိစ္စရပ်

- လမ်းတစ်လမ်း အမှိုက်ရှင်း (clean street)
- မြောက်ဘက်လှည့် (turn north)
- လမ်းပြောင်း/နောက်တစ်လမ်းကူး (change street)

သုံးခုပါဝင်တယ်။ အပေါ်ဆုံးကို ပထမအလွှာလို့ ယူဆရင် ဒါသည် ဒုတိယ အလွှာပါ။ တစ်ချင်း လိုအပ်သလို ထပ်မံခွဲခြမ်းစိတ်ဖြာ ရပါမယ်။ ဒါပေမဲ့ နောက်ထပ်တစ်ဆင့်ပဲ ခွဲခြမ်းစိတ်ဖြာ ရမှာပါ။ ဆိုလိုတာက ဒုတိယအလွှာကို ဖြေရှင်းတဲ့အခါ တတိယအလွှာထိပဲ ခွဲဖို့ စဉ်းစားတယ်။ တတိယ အလွှာကို ဘယ်လို ခွဲမလဲ ဆက်မစဉ်းစားသေးဘူး။ လမ်းတစ်လမ်း ရှင်းတဲ့ကိစ္စရှိ ဖြေရှင်းတဲ့အခါ အခုလို

- လမ်းတစ်လမ်း အမှိုက်ရှင်း (clean street)
  - ▶ ကွန်နာတစ်ခု အမှိုက်ရှင်း (clean corner)

ဒီကုန်ပိုစ် (decompose) လုပ်နိုင်တယ်။ ကွန်နာတစ်ခု ရှင်းတာက လွယ်တဲ့အတွက် ထပ်ခွဲစရာတော့ မလိုဘူး၊ အကယ်၍ ထပ်ခွဲဖို့ လိုတဲ့ ကိစ္စဖြစ်ခဲ့ရင်လည်း top-down နည်းအရ လောလောဆယ် အနေနဲ့ ဒီအဆင့်မှာပဲ မခွဲသေးဘဲ ရပ်ထားရပါမယ်။ `clean_street` ဖန်ရှင်းအတွက် `clean_corner` ရှိထားပြီး ဖြစ်တယ်လို့ မှတ်ယူရမှာပါ။

```
def clean_street():
    while front_is_clear():
        clean_corner()
        clean_corner()
```

ဒုတိယအလွှာမှာ ပါဝင်တဲ့ မြောက်ဘက်လှည့်တာနဲ့ လမ်းကူးတာကို ဆက်လက်ဖြေရှင်းပါမယ်။ နှစ်

ခုလုံး အတန်အသင့် ရိုးရှင်းတဲ့အတွက် ထပ်မွဲတော့ဘူး။ (လိုအပ်ရင်တော့ အောက်တစ်ဆင့် ထပ်ပြီး ဒါ ကွန်ပိုစ် လုပ်ရမှာပါ)။

```
def turn_north():
    while not_facing_north():
        turn_left()

def change_street():
    move()
    if right_is_blocked():
        turn_left()
    else:
        turn_right()
```

ဒုတိယတစ်ဆင့် ဖြေရှင်းပြီးသွားပါပြီ။

ညာဘက်လူညွှတ်တာကို လမ်းပြောင်းတဲ့ကိစ္စရဲ့ အခဲ့လို ယူဆကောင်း ယူဆနိုင်ပါတယ်။ ဒါပေမဲ့ အရမ်း အခြေခံကျတဲ့အတွက် နဂိုပါပြီး turn\_left နဲ့ အဆင့်တူလို ယူဆရင် ပိုပြီး ကျိုးကြောင်းဆီလျော် မယ်လို ယူဆတယ်။ ဒီလိုဖန်ရှင်မျိုးတွေဟာ ဘုံ (common) သုံး ဖြစ်လေ့ ရှိတယ်။ အခြားဖန်ရှင်တွေ (နှစ်ခုခုံအထက်) ကနေ ခေါ်သုံးရလေ့ ရှိတယ်။ ကားရဲလ်ပရိုကရမ် အားလုံးလိုလုံးမှာလည်း လိုအပ်လေ့ ရှိတယ်။ ဒီသဘောအရ turn\_north ကိုလည်း turn\_left, turn\_right တို့လို အခြေခံ အကျ ဆုံး ဘုံဖန်ရှင်အနေနဲ့ ယူဆမယ်ဆုံးရှင်လည်း မမှားပါဘူး။

(ယူဆချက်ကို ဖော်ပြတာသာဖြစ်ပါတယ်။ ဘယ်လိုမှ မှန်တယ်၊ ဘယ်လိုဆုံးရှင်ဖြင့် မှားတယ် မဆိုလို ပါ။ မိမိကိုယ်တိုင် စဉ်းစားဆင်ခြင် သုံးသပ်ပြီးမှ ဖြစ်သင့်တယ်ထင်တာကို ဆုံးဖြတ်လိုပါတယ်)။

အောက်တစ်လွှာ ဆက်ဆင်းရပါမယ်။ ဒုတိယ အဆင့်ကို ဒီကွန်ပိုစ် လုပ်လိုက်တာတွေဟာ တတိယ အလွှာမှာ ပါဝင်တယ်။ ကွန်နာတစ်ခု ရှင်းတဲ့ကိစ္စပဲ ရှိပါတယ်။ ရိုးရှင်းတဲ့အတွက် နောက်တစ်ဆင့် ထပ်မွဲ တော့ဘဲ တစ်ခါတည်း ဖန်ရှင်သတ်မှတ်ပါမယ်။

```
def clean_corner():
    if beepers_present():
        pick_beeper()
```

အပေါ်ဆုံးကနေ တစ်ဆင့်ပြီးတစ်ဆင့် ဖြေရှင်းလာတာ တတိယအလွှာမှာ ထပ်မွဲတော့တဲ့အတွက် ဒီမှာ ပဲ ပြီးသွားပြီဖြစ်တယ်။ အကယ်၍ ထပ်ခဲ့ထားတာ ရှိတယ်ဆုံးရင် အောက်တစ်ဆင့် ထပ်ဆင်းပြီး ပြီးခဲ့တဲ့ တစ်ဆင့်ချင်းမှာ လုပ်ခဲ့သလိုပဲ ဆက်သွားရမှာပါ။ ပရိုကရမ် run မယ်ဆုံးရင် ကျိုးနေသေးတဲ့ turn\_right ။ main နဲ့ အန်ထရီပိုင့် ဖြည့်ရုံပါပဲ

```
def main():
    clean_world()

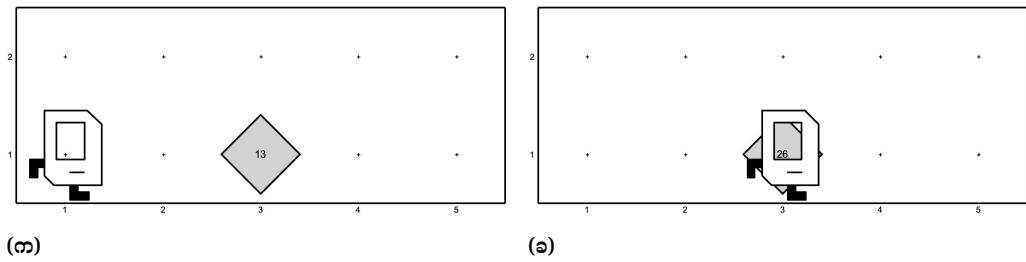
if __name__ == "__main__":
    run_karel_program("clean_world")
```

(turn\_right ဖန်ရှင်ကို ထပ်မဖော်ပြတော့ပါ)

## ခုတိယ top-down ဥပမာ (Double Beeper Pile)

အမှန်တကယ် မရှိသေးတဲ့ ဖန်ရှင်တွေကို ရှိထားပြီးဖြစ်တယ်လို့ ယူဆထားရတာဟာ top-down နည်းရဲ့ အခိုက် သော့ချက်ဖြစ်သလို ပရိုဂရမ်းမင်း စလေ့လာသူတွေအတွက် နားလည်ရ အခက်ခံး သဘောတရား တစ်ခုဆိုရင်လည်း မမှားဘူး။ အောက်တစ်ဆင့်အလွှာ ဖန်ရှင်တွေ ‘ဘာလုပ်ပေးလဲ’၊ ပရိုနဲ့ ပိုစ် ကွန်ဒီရှင် တွေ ဘယ်လိုဖြစ်သင့်လဲ တိတိကျကျ စိတ်ကူးပုံဖော်တားပြီး လက်ရှိဖန်ရှင်ကို ဘယ်လိုပေးမလဲ စဉ်းစားရ တာ ဦးနောက်ခြောက်စရာ ဖြစ်နေတာပါ။ ဥပမာ နောက်တစ်ခုလောက် ထပ်ကြည့်ရင် ပိုပြီး သဘောပေါက် သွားမယ် ထင်ပါတယ်။

အခုပုရိုဂရမ်ဗုံး ကားရဲ့က ကွန်နာတစ်ခုမှာ စုပ်ထားတဲ့ ဘိပါတွေကို နှစ်ဆဖြစ်အောင် လုပ်ပေးရ ပါမယ်။ ပုံ (၃-၉) တွင်ကြည့်ပါ။ နိုင် ဆယ့်သိုးကနေ နှစ်ဆယ့်ခြောက်ခု ဖြစ်သွားပါတယ်။ ကားရဲ့က ကိန်းကောင်းတွေအကြောင်း နားမလည်သလို ရော့က်တာလည်း မလုပ်တတ်ပါဘူး။ ဒါကြောင့် ဘိပါနှစ် ဆွားဖို့ အခြားနည်းလမ်းရှာရပါမယ်။



ပုံ ၃-၉

ဘိပါပုံ (အစုအပိုက် ဆိုလို) ထဲကနေ ဘိပါတစ်ခုကောက်လိုက်၊ ရှေ့ကွန်နာမှာ နှစ်ခုချထားလိုက် ဆက်တိုက် လုပ်မယ်ဆိုရင် နိုင်ဘိပါတွေ ကုန်သွားတဲ့အခါ နှစ်ဆရိုတဲ့ ဘိပါပုံတစ်ခု (ရှေ့ကွန်နာမှာ) ရှုံးမှာပါ။ တစ်ခါတည်း ဘိပါအားလုံး ကောက်လို့ မရပါဘူး။ ကားရဲ့က ဘယ်နှစ်ခု ကောက်ထားလဲ မမှတ်မိတဲ့ အတွက်ကြောင့်ပါ။ တစ်ခုကောက်လိုက် ရှေ့ကွန်နာမှာ နှစ်ခုပြန်ချထားလိုက် လုပ်ရပါမယ်။

ရှေ့ကွန်နာမှာ နိုင်ဘိပါပုံရဲ့ နှစ်ဆဖြစ်အောင် လုပ်လို့ရတဲ့နည်းတော့ စဉ်းစားလို့ရပြီ။ ပြီးတဲ့အခါ အဲဒါ ဘိပါတွေကို နိုင်ဘိပါပုံနေရာကို ရွှေ့ရုံပါဝဲ။ ဒီကွန်ပိုစ်လုပ်ကြည့်ရင်

- ဘိပါပုံသို့ သွား (go to beeper pile)
- ဘိပါပုံကို (မူလနေရာတွင်) နှစ်ဆလုပ် (double beeper pile)
  - ▶ ဘိပါပုံကို ရှေ့ကွန်နာမှာ နှစ်ဆလုပ်
  - ▶ နိုင်ကွန်နာသို့ ဘိပါပုံ ပြန်ရွှေ့

ဘိပါပုံဆိုကို သွားတာက လွယ်တယ်

```
def go_to_beeper_pile():
    while no_beeper_present():
        move()
```

နိုင်နေရာမှာ ဘိပါပုံ နှစ်ဆလုပ်တဲ့ ကိစ္စကို နောက်တစ်ဆင့် ထပ်ခွဲထားတယ်။ ရှေ့ကွန်နာမှာ နှစ်ဆပုံတာနဲ့ နိုင်နေရာ ပြန်ရွှေ့တာ ကိစ္စနှစ်ခု ပါဝင်တယ်။ ဒီအတွက် ဖန်ရှင်နှစ်ခု ရှိတယ်လို့ မှတ်ယူရပါမယ်။ ငါးတို့ လုပ်ဆောင်ချက်က ဘာလဲ ငါးတို့ရဲ့ ပရိုနဲ့ ပိုစ် ကွန်ဒီရှင်တွေ ဘယ်လိုဖြစ်သင့်လဲ တိတိကျကျ စဉ်းစားထားရမှာပါ။ ဒီအတွက်ကို စိတ်ထဲမှာပဲ လုပ်လို့ရသလို အောက်ပါအတိုင်း ဗလာဖန်ရှင် (empty

*function)* သတ်မှတ်ပြီး docstring နဲ့ တိတိကျကျ ချရေးထားတာဟာလည်း ပရီဂရမ်မာ အများစုံ လုပ်လေ့လုပ်ထို့တဲ့ အလေ့အထတစ်ခုပါ။

```
def double_beeper_pile_next():
    """
    ဘိပါပိုကို ရွှေ့ကွန်နာတွင် နှစ်ဆဖြစ်အောင် ရွှေ့ပေးသည်
    Precondition: ဘိပါပိုပေါ်တွင် အရှေ့ဘက် မျက်နှာမှု၍ ရှိနေမယ်
    Postcondition: နှစ်ဆဘိပါပိုပေါ်တွင် အရှေ့ဘက် မျက်နှာမှု၍
    """
    pass
```

```
def move_beeper_pile_next():
    """
    ဘိပါပိုကို ရွှေ့ကွန်နာသို့ ရွှေ့ပေးသည်
    Precondition: ဘိပါပိုပေါ်တွင် အစွာက်ဘက်မျက်နှာမှု ရှိနေမယ်
    Postcondition: ရှေ့ကို ရွှေ့ပြီးဘိပါပိုပေါ်တွင် အနောက်ဘက်မျက်နှာမှု ရှိနေမယ်
    """
    pass
```

Python မှာ ဗလာဖန်ရှင်ဆိုပေမဲ့ စတိတ်မန့်တစ်ခုတော့ ပါရမယ်။ မဟုတ်ရင် ဆင်းတက်စ် အမှားဖြစ်မှုပါ။ ဒါကြောင့် pass စတိတ်မန့်ကို ယာယိအနေနဲ့ သုံးလေ့ရှိတယ်။ ဒ်မီစတိတ်မန့် (dummy statement) လို့ ခေါ်ပါတယ်။ ဖန်ရှင်နှစ်ခုရဲ့ ပရီနဲ့ ပိုစ် ကွန်ဒီရှင်တွေကို ပုံ (၂.၁၀) နဲ့ (၂.၁၁) မှာ တွေ့နိုင်ပါတယ်။ နှစ်ခုလုံး မျက်နှာမှုတဲ့ဘက် မပြောင်းတာကို ကရှုပြပါ။ ဘိပါပိုက မူလနေရာမဟုတ်ဘဲ ရှေ့ကိုရောက်သွားတာကိုလည်း ကရှုပြပါ။

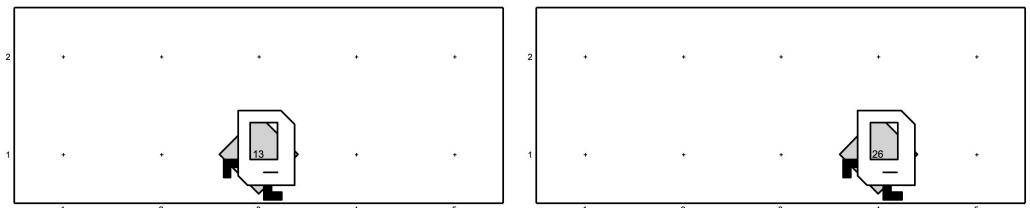
မူလနေရာမှာပဲ ဘိပါနှစ်ဆဖြစ်အောင် လုပ်တဲ့ ဖန်ရှင်က ဒီလိုဖြစ်ပါမယ်

```
def double_beeper_pile():
    """
    ဘိပါပိုကို မူလနေရာတွင်ပင် နှစ်ဆတိုးပေးသည်
    Precondition: ဘိပါပိုပေါ်တွင် အရှေ့ဘက်မျက်နှာမှုလျက် ရှိနေ
    Postcondition: နှစ်ဆတိုးပြီး ဘိပါပိုတွင် မူလအတိုင်း အရှေ့ဘက်မျက်နှာမှုလျက် ရှိနေ
    """
    double_beeper_pile_next()
    turn_left()
    turn_left()
    move_beeper_pile_next()
    turn_left()
    turn_left()
```

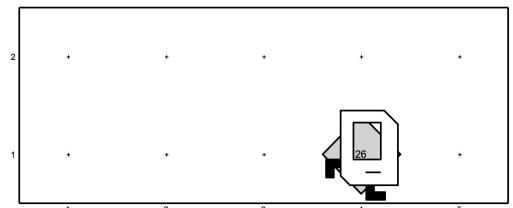
ပထမဖန်ရှင်ပြီး ဒုတိယဖန်ရှင်အတွက် အဆင်သင့်ဖြစ်အောင် ဘယ်ဘက်နှစ်ခါ လှည့်ရပါမယ်။ နောက်ဆုံးမှာလည်း အရှေ့ဘက် ပြန်လှည့်ပေးရမယ်။ မဟုတ်ရင် အခုဖန်ရှင်ရဲ့ သတ်မှတ်ထားတဲ့ ပိုစ်ကွန်ဒီရှင်နဲ့ မကိုက်ညီဘဲ အနောက်ဘက်လှည့် အနေအထားဖြစ်နေမှာ။

အထက်ပါဖန်ရှင်မှာ ဘယ် နှစ်ခါလှည့်ကို နှစ်ကြိမ်လုပ်ထားပါတယ်။ ဆန်ကျင်ဘက် အရပ်ကို လှည့်ဖို့အတွက် ရည်ရွယ်တာပါ။ ဒီအတွက် turn\_around ဖန်ရှင် ရှိသင့်ပါတယ်။ ရှိမယ်ဆိုရင်

```
def double_beeper_pile():
```

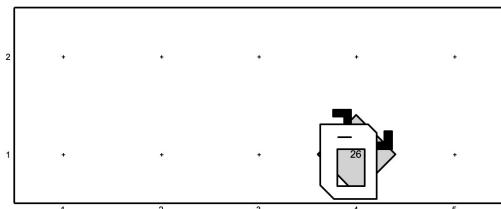


(က)

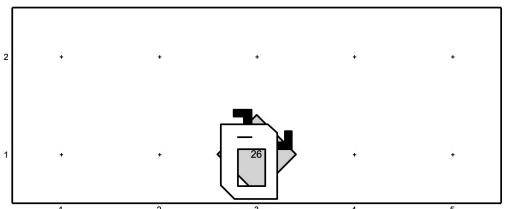


(ခ)

ပုံ ၃.၀၀



(က)



(ခ)

ပုံ ၃.၀၁

„ „ „

ဘိပါပိုကို မူလနေရာတွင်ပင် နှစ်ဆတိုးပေးသည်

*Precondition:* ဘိပါပိုပေါ်တွင် အရောက်မျက်နှာမှုလျက် ရှိနေ

*Postcondition:* နှစ်ဆတိုးပြီး ဘိပါပိုတွင် မူလအတိုင်း အရောက်မျက်နှာမှုလျက် ရှိနေ

„ „ „

```
double_beeper_pile_next()
turn_around()
move_beeper_pile_next()
turn_around()
```

ဒီနေရာမှာ မှတ်သားသင့်တာတစ်ခုက ဒီကွန်ပိုစ် လုပ်တဲ့အခါ တစ်ခါတည်းနဲ့ ပြီးပြည့်စုံတဲ့ ရလဒ် ဖြစ် ချင့်မှ ဖြစ်မှုပါ။ တစ်ခါတစ်ရုံ ပရိုဂရမ် ရေးနေရင်း စိတ်ကူးသစ် သို့မဟုတ် ပိုကောင်းတဲ့နည်းလမ်း ခေါင်းထဲ ပေါ်လာတတ်ပါတယ် တဖြည်းဖြည်း သဘောပေါက်လာတတ်တယ်။ ဒီအခါမှာ ဒီကွန်ပိုစ် လုပ် တာကို လိုအပ်သလို အလိုက်သင့် ပြောင်းပေးနိုင်ပါတယ်။ အထက်က ဥပမာမှာ turn\_around လိုအပ် မယ်ဆိုတာ ကြိုးမသိခဲ့ပါဘူး။

double\_beeper\_pile\_next နဲ့ move\_beeper\_pile\_next အတွက် ဆက်လက်စဉ်းစားပါ မယ်။ အတန်အသင့် လွယ်ကူမယ် ယူဆတဲ့အတွက် နောက်တစ်ဆင့် ထပ်မံ့ထော့ဘူး။

```
def double_beeper_pile_next():
    „ „ „
```

ဘိပါပိုကို ရေးကွန်နာတွင် နှစ်ဆဖြစ်အောင် ရွှေပေးသည်

*Precondition:* ဘိပါပိုပေါ်တွင် အရောက် မျက်နှာမှု၍ ရှိနေမယ်

*Postcondition:* နှစ်ဆဘိပါပိုပေါ်တွင် အရောက် မျက်နှာမှု၍

„ „ „

```

while beepers_present():
    pick_beeper()
    move()
    put_beeper()
    put_beeper()
    turn_around()
    move()
    turn_around()

    move()

def move_beeper_pile_next():
    """
    ဘိပိုကို ရွှေ့ကွန်နာသို့ ရွှေ့ပေးသည်
    Precondition: ဘိပိုပေါ်တွင် အနောက်ဘက်မျက်နှာမှ ရှိနေမယ်
    Postcondition: ရွှေ့ကို ရွှေ့ပြီးဘိပိုပေါ်တွင် အနောက်ဘက်မျက်နှာမှ ရှိနေမယ်
    """
    while beepers_present():
        pick_beeper()
        move()
        put_beeper()
        turn_around()
        move()
        turn_around()

    move()

အခုနောက်ခံး: turn_around, main နဲ့ အနုထရိပိုင့် ဖြည့်စိုး ကျွန်ပါတယ်။ ပရိဂရမ် အစအဆုံး  

လေ့လာကြည့်ပါ။

# File: double_beeper_pile.py
from stanfordkarel import *

def main():
    """Karel doubles the number of beepers in a beeper pile"""
    go_to_beeper_pile()
    double_beeper_pile()

def double_beeper_pile():
    double_beeper_pile_next()
    turn_around()
    move_beeper_pile_next()
    turn_around()

```

```
def go_to_beeper_pile():
    while no_beeper_present():
        move()

def double_beeper_pile_next():
    while beepers_present():
        pick_beeper()
        move()
        put_beeper()
        put_beeper()
        turn_around()
        move()
        turn_around()

    move()

def move_beeper_pile_next():
    while beepers_present():
        pick_beeper()
        move()
        put_beeper()
        turn_around()
        move()
        turn_around()

    move()

def turn_around():
    turn_left()
    turn_left()

if __name__ == "__main__":
    run_karel_program("double_beeper_pile")
```

# အခန်း ၄

## ကားရဲလိနှင့် ရီကားဆစ်ဖိ ဖန်ရှင်များ

ဖန်ရှင်တစ်ခုကနေ ‘အခြား’ ဖန်ရှင်တွေ ခေါ်သုံးတာကို အခန်း (၃) မှာ တွေ့ခဲ့ပြီးပါပြီ။ ဒါပေမဲ့ ဖန်ရှင်တစ်ခုက ငါးကိုယ်ငါး ပြန်ခေါ်ထားတာကိုတော့ မဖြေဖူးသေးပါဘူး။ ဖန်ရှင်တွေဟာ ဆော်ဖို့ အဆောက်အအီး တည်ဆောက်ရာမှာ မရှိမဖြစ်တဲ့ အခြေခံအုတ်ချပ်တွေလို ဆိုရမှာပါ။ ငါးကိုယ်တိုင်ကို ပြန်လည်အသုံးပြု၍ ဖန်ရှင်အုတ်ချပ်တစ်ခု ဖန်တီးလို ရနိုင်ပါမလား။ ဒီမေးခွန်းဟာ ထူးဆန်းကောင်း ထူးဆန်းနေပါလိမ့်မယ်။ အခြေခံကျပြီး စိတ်ဝင်စားစရာကောင်းတဲ့ ဖို့လော်ဆော်ဖိ မေးခွန်းလည်း ဖြစ်တယ်။

ဖန်ရှင် သတ်မှတ်ချက်ထဲမှာ ငါးဖန်ရှင်ကိုယ်တိုင်ကို ပြန်ခေါ်လို ရပါတယ်။ ရီကားဆစ်ဖိ ဖန်ရှင် (recursive function) လို ခေါ်တယ်။ ရီကားဆစ်ဖိ ဖန်ရှင်တွေဟာ ဘီဂင်နာ ပရိုဂရမ်မာအတွက် နားလည်ဖို့ ခက်ခဲတဲ့ သဘောတရားအဖြစ် ယူဆကြတာကြောင့် စာအုပ် အတော်များများမှာ နောက်ကျပြီး ဖော်ပြလေ့ရှိတယ်။ တကယ်က လူများစု ထင်း/ပြောသလို နားမလည်နိုင်လောက်အောင် ရှုပ်ထွေး ခက်ခဲတဲ့ သဘောတရား မဟုတ်ပါဘူး။ သာမန်လူအားလုံး နားလည်နိုင်ပါတယ်။ ဒါကြောင့် စောစောစီးစီး အခုပဲ မိတ်ဆက်ပေးလိုက်ပါတယ်။ အကယ်၍ နားမလည်ခဲရင်လည်း ပြဿနာမရှိပါဘူး။ အကြမ်းဖျော်းလောက် ဖတ်ကြည့်ပြီး နောက်လာမဲ့ အခန်းတွေကို ကျော်သွားနိုင်ပါတယ်။ နောင်တစ်ချိန်ကျမှ ပြန်လာဖတ်ပေါ့။

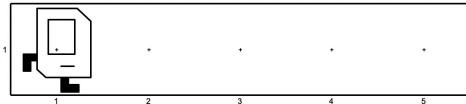
### ၄.၁ ရီကားဆစ်ဖိ ဖန်ရှင် ဘယ်လို အလုပ်လုပ်လဲ

ရီကားဆစ်ဖိ ဖန်ရှင် ဥပမာတစ်ခုကို လေ့လာကြည့်ပါမယ်။ အောက်ဖော်ပြပါ ဖန်ရှင်ဟာ ငါးကိုယ်တိုင်ကို ငါး ပြန်ခေါ်ထားတာ ကရုပ်ကြည့်ပါ။ recursive call လို ကွန်းမန်ရေးထားတဲ့ လိုင်းမှာပါ။

```
def make_beeper_row():
    if front_is_clear():
        put_beeper()
        move()
        make_beeper_row() # recursive call
    else:
        put_beeper()
```

ဒီဖန်ရှင်ကို ခေါ်လိုက်ရင် ဘာဆက်ဖြစ်မလဲဆိုတာ စိတ်ဝင်စားစရာပါ။ ဖန်ရှင်မစတင်မိ အနေအထား ကို ပုံ (၄.၁) မှာ ကြည့်ပါ။ ဖန်ရှင်ကို ကန်းစီး စခေါ်လိုက်တာမို့လို့ initial call လို ရည်ညွှန်းပါမယ်။

```
# initial call
make_beeper_row()
```



### ပုံ ၄.၁

ဖန်ရှင် စတင် လုပ်ဆောင်ပါမယ်။ ရှေ့မှာရှင်းနေတဲ့ အတွက် if ဘလောက်ကို လုပ်မှုပါ

```
put_beeper()
move()
make_beeper_row() # recursive call
```

ဘိပါချု ရှေ့တိုး ပုံ (၄.၂) (က) ကြည့်ပါ။ ပြီးရင် သူကိုယ်သူ ပြန်ခေါ်ထားတယ်။ ဒါဟာ ပထမဆုံး တစ်ကြိမ်ပါ။ ဖန်ရှင်ခေါ်ရင် ဖြစ်မြေအတိုင်းပဲ ဖန်ရှင်နဲ့ သက်ဆိုင်တဲ့ ဘလောက်ကို ဆောက်ရှုက်တာပေါ့။ ဒီတော့ make\_beeper\_row ဖန်ရှင်ဘလောက်ကိုပဲ တစ်ခါထပ်လုပ်မှုပါ။ ရှေ့မှာ ရှင်းနေတဲ့အတွက် if အပိုင်းကို လုပ်တယ်

```
put_beeper()
move()
make_beeper_row() # recursive call
```

ဘိပါချု ရှေ့တိုး ပုံ (၄.၂) (ခ) ပြီး သူကိုယ်သူ ပြန်ခေါ်ထားတဲ့ကိစ္စ တစ်ခါထပ်ဖြစ်ပြန်တယ်။ ဒါနဲ့ဆို နှစ်ကြိမ်။ ဖန်ရှင်ဘလောက် အလုပ် ပြန်လုပ်မယ်။ ရှေ့မှာရှင်းတယ်၊ if ကိုပဲ ထပ်လုပ်

```
put_beeper()
move()
make_beeper_row() # recursive call
```

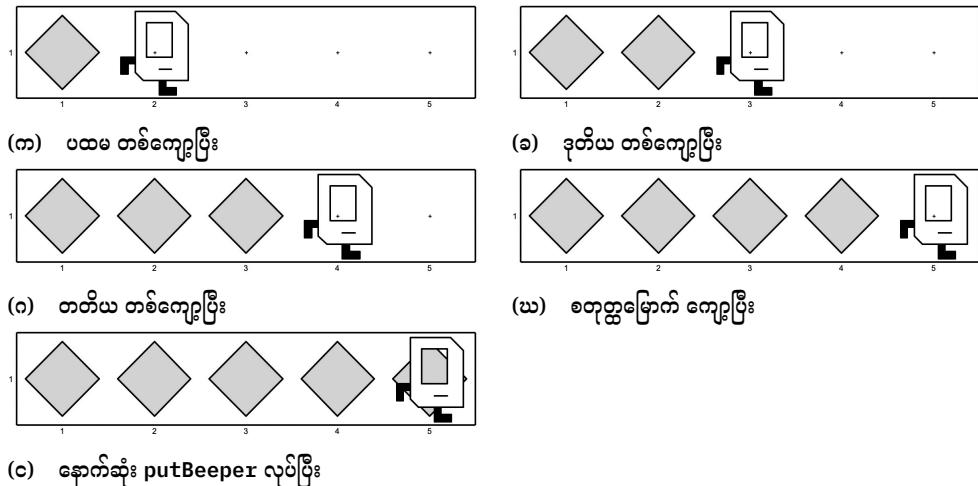
ပုံ (၄.၂) (ဂ) နေရာရောက်ပြီး သူကိုယ်သူ ထပ်ခေါ်ထားပြန်တယ်။ သုံးကြိမ်ရှိပြီ။ ဒီတစ်ခါလည်း if အပိုင်းပဲ ထပ်လုပ်

```
put_beeper()
move()
make_beeper_row() # recursive call
```

ရှေ့တိုးပြီးရင် နံရုပ်တ်နေပြီ ပုံ (၄.၂) (ဃ)။ သူကိုယ်သူ ခေါ်တယ်။ ရှေ့မှာ ပိတ်နေတဲ့အတွက် else အပိုင်းကို လုပ်မှုပါ။

```
put_beeper()
```

သူကိုယ်သူ ပြန်ခေါ်တဲ့ကစွာ ထပ်မဖြစ်တော့ဘူး။ ဒီမှာပဲ ပြီးဆုံးသွားတယ်။ ရိုကားဆစ်ဖဲ့ ဖန်ရှင် အလုပ် လုပ်ပဲ အခြားသဘောတရားက ဒါပါပဲ။ loop တွေ မသုံးဘဲ ပြန်ကျော်နေတဲ့ သဘောကို ရိုကားဆစ်ဖဲ့ ဖန်ရှင်မှာ တွေ့ရပါတယ်။ ဖန်ရှင်က သူကိုယ်သူ (သို့ ကိုယ့်ကိုကိုယ်) ပြန်ခေါ်တာကို recursive call လို ခေါ်ပါတယ်။ ဒါကို အကြောင်းရှင်းအောင် recursive function တွေမှာ recursive call အနည်းဆုံး တစ်ခု ပါ ရမှာ ဖြစ်ပါတယ်။



ပုံ ၄၂

```
def make_beeper_row():
    if front_is_clear():
        put_beeper()
        move()
        make_beeper_row()
    else:
        put_beeper()

make_beeper_row()
```

ရီကားဆစ်ဖောက် ဖြစ်တာကို စိတ်ကူးပံ့ဖော်ကြည့်ဖို့ ပြထားတာပါ။ အပြင်ဆုံး မြှားအနက်က ကန် ပြုး ဖန်ရှင်ကောလ် စတင်တာဖြစ်ပေါ်တာကို ဖော်ပြထာ်။ ရီကားဆစ်ဖောက် မဟုတ်သေးဘူး။ မြှား အပြားက ရီကားဆစ်ဖောက်လောက်တွင် ဖန်ရှင်အစ ပြန်ရောက်သွားတာကို ပြထာ်။ ပထမနဲ့ ဒုတိယ ရီကားဆစ်ဖောက် နှစ်ခုအတွက်ပြထားတာပါ။ ရေးက ဥပမာအတွက် မြှားအပြား လေးခါ ရီရှုံးမှာပါ (ရီကား ဆစ်ဖောက် လေးကြိမ်အတွက်)။ မြှားအပြား နောက်ထပ် နှစ်ခုရီတယ် မှတ်ယူပါ (ပုံမှာထပ်ထည့်ရင် ကြပ်ညပ်ပြီး ကြည့်ရှုရှုပ်လို့ မဆွဲပြတာ)။ လေးကြိမ်မြောက်မှာ ရီကားဆစ်ဖောက် ထပ်မဖြစ်တော့ဘူး။ (if အပိုင်းကို မလုပ်တော့ဘူးအတွက်)။ ဘိဝါချုပြီး ဖန်ရှင်ကောလ် စခဲ့တဲ့နေရာကို ပြန်ရောက် သွားမယ် (မြှားအနဲ့)။ ဘယ်လို့ ပြန်ရောက်သွားတာလဲ ဆက်ကြည့်ရအောင်။

### နောက်ဆုံး ရီကားဆစ်ဖောက်မှ ပြန်လုပ်ခြင်း

နောက်ဆုံး ရီကားဆစ်ဖောက်ကနေ မူလ ဖန်ရှင်ခေါ်ခဲ့တဲ့ နေရာကို ဘယ်လိုပြန်ရောက်သွားတာလဲ။ ဒီကိစ္စနားလည်ဖို့ ဖန်ရှင် return ပြန်ခြင်းအကြောင်း အရင်ကြည့်ရှုပါမယ်။ ဖန်ရှင်ကောလ် လုပ်ဆောင် တဲ့အခါ အဲဒီဖန်ရှင်နဲ့ သက်ဆိုင်တဲ့ ဘလောက်ဆီကို ခုန်ကျော် ရောက်ရှိသွားမှာပါ။ ဖန်ရှင်ဘလောက်ကို လုပ်ဆောင်ပြီး ခေါ်ခဲ့တဲ့ နေရာကို ပြန်လည်ရောက်ရှိသွားမှာ ဖြစ်တယ်။ ဒီဖြစ်စဉ်ကို ဖန်ရှင် return ပြန်တယ်လို့ ပြောပါတယ်။

```

def main():
    turn_right()
    move()
    turn_right()
    move()

def turn_right():
    turn_left()
    turn_left()
    turn_left()

```

ပထမ turn\_right ကောင် လုပ်ဆောင်တဲ့အခါ ကောင်လုပ်တဲ့ နေရာကနေ turn\_right ဖန်ရှင်တဲ့ ကို jump လုပ်ပြီး ရောက်သွားတယ်။ မြှားအနက်နဲ့ ပြထားတယ်။ ဖန်ရှင်ဘလောက် လုပ်ဆောင်ပြီးတဲ့ အခါ ခေါ်ခဲ့တဲ့နေရာ main ဖန်ရှင်တဲ့ ပြန်ရောက်သွားတယ် (မြှားအနီး)။ ဒုတိယ turn\_right လည်း ထိုနည်းတူစွာပဲ ဖြစ်ပါတယ်။

```

def main():
    turn_right()
    move()
    turn_right()
    move()

def turn_right():
    turn_left()
    turn_left()
    turn_left()

```

နှစ်ဆင့် သုံးဆင့် ဖန်ရှင်ကောင်တွေမှာလည်း ဒီသဘောတရား အတိုင်းပါပဲ။ အောက်ဖော်ပြပါ ပုဂ္ဂိုလ်ကုန်ကို ကြည့်ပါ။ main ဖန်ရှင်တဲ့ကနေ do\_tricks ဆိုကို ရောက်သွားမယ်။ do\_tricks ထဲက နေ put\_two ထဲကို ရောက်သွားမယ်။

```

def main():
    do_tricks()
    move()

def do_tricks():
    move()
    put_two()
    turn_left()
    move()

def put_two():
    put_beeper()
    put_beeper()

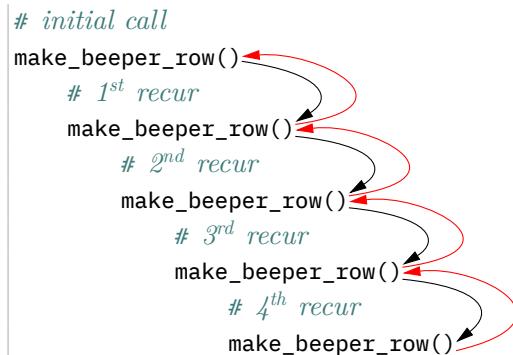
```

put\_two ပြီးသွားတဲ့အခါ do\_tricks ထဲ ပြန်ရောက်သွားမယ်။ turn\_left, move ဆက်လုပ်

ပြီး do\_tricks ခေါ်ခဲ့တဲ့နေရာ main ထဲ ပြန်ရောက်သွားမယ်။ နောက်ဆုံး main ထဲက move ကို ဆက်လုပ်ပါတယ်။

ဖန်ရှင် အဆင့်ဆင့် ခေါ်ထားတဲ့အခါ နောက်ဆုံးခေါ်တဲ့ ဖန်ရှင်က အရင်ဆုံး return ပြန်ပါတယ်။ main ကနေ do\_tricks ကိုခေါ်၊ do\_tricks ကနေ put\_two ကိုခေါ်ထားရင် put\_two ကနေ do\_tricks ဆိုကို အရင် return ပြန်တယ်။ ပြီးတော့မှ do\_tricks ကနေ main ကို ပြန်ရောက်မှာပါ။ ဒီသဘောအရ put\_two return မပြန်မချင်း do\_tricks ဖန်ရှင်မပြီးသေးဘူး။ put\_two ကနေ ပြန်လာပြီးမှ ကျော်တဲ့ turn\_left, move ဆက်လုပ်တယ်။ ပြီးတော့မှ do\_tricks ဖန်ရှင်က return ပြန်ပါတယ်။

ရိုကားဆစ်စုံ ဖန်ရှင်ကောလ်တွေ ဘယ်လို့ return ပြန်လဲ။ ရွှေ့ကလို့ မြှားတွေနဲ့ ဆွဲပြလို့ ရပေမဲ့ ကြည့်ရတာ ရှုပ်ရှက်ခတ်နေမှာပါ။ အခုလို့ မြင်ကြည့်ရင် ပိုရှင်းပါတယ်။



မြှားအနုက်တွေက ဖန်ရှင်ကောလ် တစ်ဆင့်ပြီးတစ်ဆင့် ဖြစ်တာကို ပြတာပါ။ အထက်မှအောက် အစီအစဉ်အတိုင်း ဖြစ်ပါတယ်။ လေးကြိမ်မြှာက်မှာ နောက်ထပ် ရိုကားဆစ်စုံကောလ် ထပ်မဖြစ်တော့ဘဲ နောက်ဆုံး ရိုကားဆစ်စုံကောလ်က အရင်ဆုံး return စပြန်ပါတယ် (အောက်ဆုံး မြှားအနီး၊ ပြထား)။ ဒီအောက် တတိယ ရိုကားဆစ်စုံကောလ်ကို ပြန်ရောက်သွားမှာပါ။ ဒီအတိုင်း တစ်ဆင့်ပြီးတစ်ဆင့် အထက်ကို return ပြန်ပြီး နောက်ဆုံးမှာ ပထမဆုံး ခေါ်ခဲ့တဲ့နေရာ ပြန်ရောက်သွားမှာပါ။ (အခုပြထားတာကို တကယ့် Python ကုဒ် အနေနဲ့ မယူဆရပါ။ ဖြစ်စဉ် နားလည်အောင် ပြခြင်းသာဖြစ်ပါတယ်)။ နဲ့ပြင်ထားတဲ့ make\_beeper\_row ဗားရှင်းမှာ return ပြန်တဲ့ ဖြစ်စဉ်ကို ကြည့်ရအောင်။

| make\_beeper\_row()

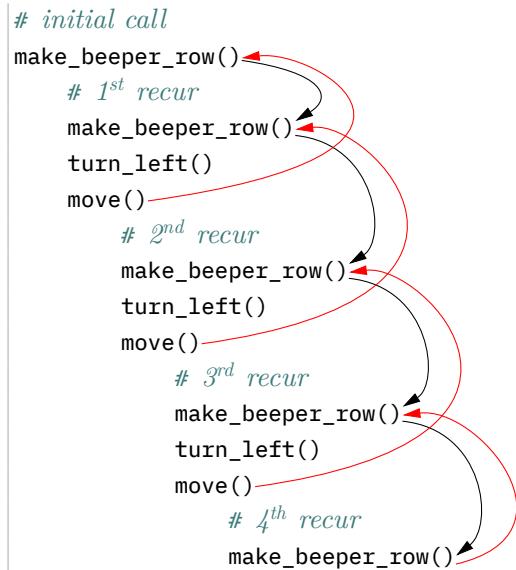
```

def make_beeper_row():
    if front_is_clear():
        put_beeper()
        move()
        make_beeper_row()
        turn_left()
        move()
    else:
        put_beeper()

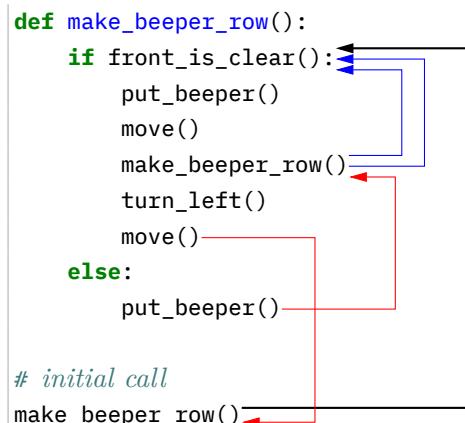
```

if အပိုင်း ရိုကားဆစ်စုံကောလ်ပြီး turn\_left နဲ့ move ထပ်ဖြည့်ထားတာ။ ရိုကားဆစ်စုံကောလ် return ပြန်လာပြီးမှ ဒီနှစ်ခု ဆက်လုပ်မှာပါ။ နောက်ဆုံး ရိုကားဆစ်စုံကောလ်က return စဖြစ်တယ်။ ဒီတော့မှ တတိယအောက် turn\_left နဲ့ move ကို လုပ်အောင်မှာပါ။ ပြီးမှ တတိယကောလ် return ပြန်

တယ်။ ဒီအတိုင်း အထက်ကို တက်သွားပြီး ကျဉ်နေသေးတဲ့ စတိတ်မန်တွေကို လုပ်ဆောင်ပါတယ်။ ပထမ ဆုံးကောလ်အောက် ကျဉ်နေတောက္ခာ နောက်ဆုံးကျမှ ပြီးမှာပါ။



ရိုကားဆစ်ဖောက်လ် နှစ်ခါပဲ ဖြစ်မယ်ဆိုရင် အောက်ပါအတိုင်း မြင်ကြည့်လို ရပါတယ်။ မြှေးအဖြားနှစ်ခုက ရိုကားဆစ်ဖောက်လ်ဖြစ်တာ။ ဒုတိယကောလ်က ဘိပါချပြီး (else အပိုင်း) အရင် return မယ်။ အထက်ကို ညွှန်တဲ့ မြှေးအနီကို ကြည့်ပါ။ ဘယ်လူည့်၊ ရှေ့တိုးပြီး ပထမ ကောလ်က နောက်မှ initial call လုပ်ခဲ့တဲ့ဆို ပြန်ရောက်တာ။ အောက်ကိုညွှန်တဲ့ မြှေးအနီကို ကြည့်ပါ။



ရိုကားဆစ်ဖောက်အကြောင်း လေ့လာတဲ့အခါ ဘိုင်နာအများစုံ ကြောက်လုပ်ညက် နားလည်ဖို့ အတွက် အခက်အခဲဆုံးတစ်ခုက return ပြန်တဲ့ သဘောတရားပါပဲ။ များများစုံးစား၊ များများလေ့ကျင့်ရင် ဒီအခက်အခဲ ကျော်ဖြတ်နိုင်မှာပါ။ if...else အပြီးမှာ put\_beeper လေးတစ်ခုပဲ ထပ်ဖြည့်လိုက်ရင် ဘယ်လိုဖြစ်မလဲ။

```

# File: make_beeper_row2.py
def make_beeper_row():
    if front_is_clear():

```

```

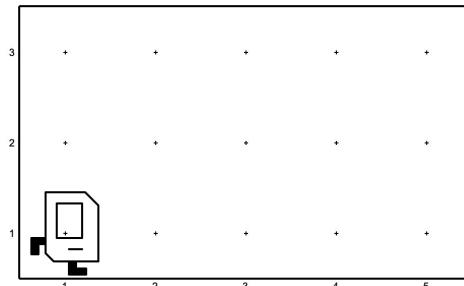
put_beeper()
move()
make_beeper_row()
turn_left()
move()

else:
    put_beeper()
    put_beeper()

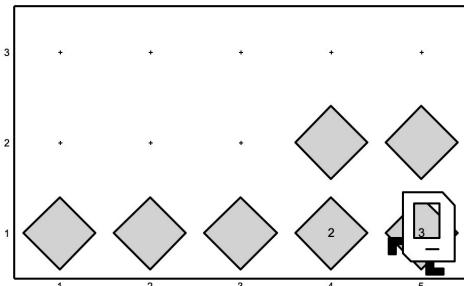
# initial call
make_beeper_row()

```

ပုံ (၄.၃) (က) နဲ့ (ခ) က မတိုင်မိနဲ့ ပြီးနောက် အခြေအနေပါ။ နောက်ဆုံးမှာ ဘိပါလေးခုကို ပါတ်လည် ဘယ်လိုချွေသူးလဲ စဉ်းစားကြည့်ပါ။ ရှေ့မှုဖော်ပြခဲ့သလို ဖန်ရှင်ကောလ်တွေ တစ်ဆင့်ပြီးတစ်ဆင့် ဖြစ်ပုံနဲ့  
return ဖြစ်ပုံကို မြှေးဆွဲကြည့်ပါ။



(က)



(ခ)

ပုံ ၄.၃

## ၄.၂ ရီကားဆစ်ဖိနည်းပြင့် ပုံးဖော်ရှင်းခြင်း

ရှေ့စက်ရှင်မှာ လေ့လာခဲ့တာက ရီကားဆစ်ဖိနှင့် ဘယ်လို အလုပ်လုပ်လဲပဲ ရှိပါသေးတယ်။ တစ်နည်း အားဖြင့် မကြန်စ်မေ့ (mechanism) ကို လေ့လာတာပါ။ အခုံတစ်ခါ ရီကားဆစ်ဖိနှင့်တွေနဲ့ ပုံးဖော်ရှင်းမှု တွေ ဘယ်လိုပြောရှင်းမလဲ ဆက်လက်လေ့လာပါမယ်။ ‘ရီကားဆစ်ဖိနှင့် စဉ်းစားခြင်း’ (thinking recursively) သို့မဟုတ် ‘ရီကားဆစ်ဖိနည်းပြင့် ပုံးဖော်ရှင်းခြင်း’ (solving problems recursively) ကို လေ့လာမှုပါ။

### ရီကားဆစ်ဖိနည်းပြင့် စဉ်းစားခြင်း ဥပမာ (၁)

ကွန်းနှုန်းရှိတဲ့ ဘိပါတွေအားလုံး ကောက်မယ်ဆိုပါစို့။ ဘိပါတွေခုံး အထက်ရှိနိုင်တယ်။ ဘိပါမရှိတာလည်း ဖြစ်နိုင်တယ် ယူဆပါ။ ဒီအတွက် ရီကားဆစ်ဖိနှင့် သတ်မှတ်ပါမယ်။

```

def pick_all beepers():
    ... # to do soon

```

ဖြော်ရှင်းမဲ့ကိစ္စတစ်ခုကို ဂင်းကိုယ်တိုင်နဲ့ ပုံပန်းသဏ္ဌာန်တူပြီး အရွယ်အစားအားဖြင့် တစ်ဆင့်ထက်

တစ်ဆင့် သေးငယ်တဲ့ ကိစ္စတွေအဖြစ် ခွဲခြမ်းကြည့်ပါတယ်။ ဥပမာ ဘိပါဒါးခုရှိတဲ့ ကိစ္စကို ဘိပါလေးခု သံဃာ၊ နှစ်ခု နဲ့ တစ်ခု ရှိတဲ့ ကိစ္စတွေအဖြစ် ခွဲပြီး မြင်ကြည့်ရမှာပါ။ ဒီကိစ္စမှာ ဘိပါအရေအတွက်ဟာ အရွယ်အစားပဲ။ လေးခုရှိတဲ့ ကိစ္စဟာ ငါးခုရှိတဲ့ကိစ္စထက် အရွယ်အားဖြင့် တစ်ဆင့်ငယ်တာပေါ့။ ဘိပါမရှိ တာလည်း ဖြစ်နိုင်တော့ သူည်ဘိပါဟာ အငယ်ဆုံးဖြစ်တယ်လို့ ယူဆနိုင်တယ်။

`pick_all_beeper` ဖန်ရှင်ဟာ လက်ရှိဖြေရှင်းမဲ့ အရွယ်အစားထက် တစ်ဆင့်ငယ်တဲ့ ကိစ္စကို ဖြေရှင်းနိုင်ပြီးသားလို့ မှတ်ယူရပါမယ်။ လက်ရှိဖြေရှင်းမဲ့ ကိစ္စက ဘိပါဒါးခု ကောက်ရမယ်ဆုံးရင် ဘိပါလေးခု ကောက်နိုင်ပြီးသားလို့ ယူဆရမှာပါ။  $n$  ဘိပါရှိတယ် ဆုံးရင်  $n - 1$  ဘိပါကို ကောက်နိုင်ပြီးသား ယူဆရမယ်။ Top-down နည်းမှာလည်း ဒီလိုပဲ မရှိသေးဘဲ မလုပ်နိုင်သေးဘဲ ရှိတယ် လုပ်နိုင်တယ် မှတ်ယူပြီး စဉ်းစားဖြေရှင်းတာကို ပြန်အုတ်ရမှာပါ။ ရိုကားဆစ်စ် စဉ်းစားတဲ့အခါမှာ လက်ရှိသတ်မှတ်နေတဲ့ ဖန်ရှင်ကိုယ်တိုင်ကို ရှိပြီးဖြစ်တယ်လို့ သဘောထားရတာ။

ပြီးတဲ့အခါ လက်ရှိကိစ္စကနေ သူထက်တစ်ဆင့်ငယ်တဲ့ ကိစ္စဖြစ်သွားအောင် ဘာလုပ်ရမလဲ စဉ်းစားရတယ်။ ဘိပါဒါးခုကနေ လေးခုဖြစ်အောင် ဘိပါတစ်ခု ကောက်ရမှာပေါ့။ ယျော့ယျော့ပြောရင်  $n$  ဘိပါရှိရှင် ဆုံးရင်  $n - 1$  ဘိပါဖြစ်သွားအောင် ဘာလုပ်ရမလဲ စဉ်းစားတာ။

လက်ရှိဖန်ရှင်ဟာ  $n - 1$  ဘိပါကို ကောက်နိုင်ပြီးသားလို့ ယူဆထားတယ်။  $n$  ဘိပါရှိရှင် ဘိပါတစ်ခု ကောက်လိုက်ရင်  $n - 1$  ဘိပါဖြစ်သွားမယ်။ ကျွန်းနေတဲ့  $n - 1$  ဘိပါကောက်ဖို့ လက်ရှိဖန်ရှင်ကိုပဲ ပြန်ခေါ်လိုက်မှာပေါ့။

```
# only partially done
def pick_all_beeper():
    ...
    # to pick (n) beepers
    pick_beeper()
    pick_all_beeper() # assuming it can pick (n - 1) beepers
    ...
```

ရိုကားဆစ်စ် စဉ်းစားတတ်ဖို့ ဒီအဆင့်က အခရာအကျဆုံးပဲ။ တစ်ဆင့်ငယ်တဲ့ ကိစ္စ  $n - 1$  ကို ဖြေရှင်းနိုင်ပြီးသားလို့ မှတ်ယူပြီး လက်ရှိကန္တ  $n$  ကို ဘယ်လို့ ဖြေရှင်းမလဲ စဉ်းစားသွားတာ။

ဖန်ရှင်သတ်မှတ်ချက်က မပြီးသေးပါဘူး။ အသေးငယ်ဆုံး ကိစ္စကို ခွင့်ချက်အနေနဲ့ စဉ်းစားရမယ်။ အသေးငယ်ဆုံးကိစ္စက ဘိပါမရှိတာ (သူည်) ဖြစ်တယ်။ ဘိပါမရှိရင် ဘာမှလုပ်စရာမလိုဘူး။  $n \neq n - 1$  ဘိပါအတွက် အထက်ပါအတိုင်း စဉ်းစားတဲ့အခါ  $n \neq 0$  လို့ ယူဆရမှာပါ။ ဒါကြောင့် ဘိပါရှိမှုပဲ လုပ်အောင် အခုလုပ်ဖြစ်ရပါမယ်

```
# finished
def pick_all_beeper():
    if beepers_present():
        pick_beeper()
        pick_all_beeper()
```

ရိုကားဆစ်ဖန်ရှင် မှန်မမှန် စစ်ဆေးခြင်း

ရိုကားဆစ်ဖန်ရှင် တက်စ် (test) လုပ်ရင် အသေးဆုံးကိစ္စကနေ စရာတယ်။ ပြီးခဲတဲ့ ဖန်ရှင်အတွက် အသေးဆုံးက သူည်ပါ။ ဘိပါမရှိရင် ဖန်ရှင်က မှန်ရဲလား အရင်ဆုံး စစ်ကြည့်ပါမယ်။

```
# assume no beeper
pick_all_beeper()
```

if ဘလေက် မလုပ်ဆောင်ဘဲ return ဖြစ်သွားမှာပါ (ရီကားဆစ်စောင် မဖြစ်လိုက်ဘူး)။ သူည့်ဘို့  
ပါအတွက် ဖြစ်သတ္တုအတိုင်း ဖြစ်ပါတယ်။ ဘိပါတစ်ခုပဲ ရှိရင်ရော ဘယ်လိုဖြစ်မလဲ။ ဘိပါကောက်တယ်  
သူညာဘို့ဖြစ်သွားပြီး ရီကားဆစ်စောင် ဖြစ်မယ်။ တစ်ကြိမ်ပဲ ဖြစ်မယ်။ if ဘလေက် အလုပ်မလုပ်  
ဘဲ return ပြန်မယ်။

```
# initial call
pick_all_beeper()
# 1st recur
pick_all_beeper()
```

သုံးခုရှိရင် ရီကားဆစ်စောင် နှစ်ခါဖြစ်ပြီးမှ return ပြန်မှာပါ။

```
# initial call
pick_all_beeper()
# 1st recur
pick_all_beeper()
# 2nd recur
pick_all_beeper()
```

မျှော်လင့်ထားသလို အလုပ်လုပ်နေပါတယ်။ အထက်ပါအတိုင်း စစ်ကြည့်သွားရင် ဘိပါ သုံး၊ လေး၊  
ငါး၊ ... ခု တွေအတွက်လည်း တောက်လျောက်မှန်နေမှာပါ။ ဘာကြောင့် ပြောနိုင်ရတာလဲ။ အခုလို စဉ်းစား  
ကြည့်ခိုင်ပါတယ်။ ရှေ့မှာ စစ်ကြည့်တာ  $n = 0, n = 1$  အတွက် မှန်တာ သေချာပြီ။  $n = 2$  အတွက် စစ်  
မယ်ဆိုပါစို့

```
# start with n = 2
if beepers_present():
    pick_beepers() # after this n = 1
    pick_all_beeper() # works correctly for n = 1
```

$n = 2$  အတွက်မှန်ရင်  $n = 3$  အတွက်လည်း ဆက်ပြီး မှန်နေမှာပါ

```
# start with n = 3
if beepers_present():
    pick_beepers() # after this n = 2
    pick_all_beeper() # already works for n = 2
```

$n = 3$  အတွက်မှန်ရင်  $n = 4$  အတွက်လည်း မှန်ပြီပေါ့

```
# start with n = 4
if beepers_present():
    pick_beepers() # after this n = 3
    pick_all_beeper() # already works for n = 3
```

ဒီတိုင်းဆက်သွားရင် သူည့်အပါအဝင် မည့်သည့် အပေါင်းကိန်းပြည့်  $n$  အတွက်မဆို (non-negative  
integer) မှန်တယ်ဆိုတာ မြင်နိုင်ပါတယ်။

## ရိကားဆုံး စဉ်းစားခြင်း ဥပမာ (၂)

ဒုတိယ ဥပမာအနေနဲ့ အခန်း (၃) က အမှိုက်ရှင်းတဲ့ ဥပမာကို ရိကားဆုံးဖြစ်နည်းနဲ့ ဖြေရှင်းကြည့်ရအောင်။ လမ်းတစ်လမ်းရှင်းဖို့ ဘယ်လိုစဉ်းစားမလဲ။

ဖြေရှင်းမဲ့ ကိစ္စကို ငြင်းကိုယ်တိုင်နဲ့ သဏ္ဌာန်တူပြီး အရွယ်အစားအားဖြင့် တစ်ဆင့်ထက်တစ်ဆင့် သေး ယောက်တဲ့ ကိစ္စတွေအဖြစ် ခွဲခြမ်းကြည့်ရမှာပါ။

လမ်းတစ်ခုရဲ့ အရှည်ကို အရွယ်အစားလို့ ယူဆနိုင်တယ်။ တစ်နည်းအားဖြင့် လမ်းတစ်လျှောက် ကွန်နာအရောတွက်ဟာ အခုဖြေရှင်းမဲ့ ကိစ္စရဲ့ အရွယ်အစားပဲ။ ကွန်နာတစ်ခုတော့ အနည်းဆုံး ရှိရမယ်။ ဒါ ကြောင့် အသေးဆုံးက  $n = 1$  ဖြစ်တယ်။

clean\_street ဖန်ရှင်ဟာ လက်ရှိဖြေရှင်းမဲ့ အရွယ်အစားထက် တစ်ဆင့်ယောက်တဲ့ ကိစ္စကို ဖြေရှင်း နိုင်ပြီးသားလို့ မှတ်ယူရပါမယ်။

အခုကိစ္စအတွက် ကွန်နာဝါးခဲ့ ရှင်းမယ်ဆိုရင် clean\_street ဖန်ရှင်ဟာ ကွန်နာလေးခုနဲ့ လမ်းကို ရှင်းနိုင်တယ်လို့ ယူဆရမယ်။  $n$  ကွန်နာရှိတဲ့ လမ်းအတွက်  $n - 1$  ကွန်နာကို ရှင်းနိုင်ပြီးသားလို့ ယူဆရမှာ ဖြစ်ပါတယ်။

လက်ရှိအရွယ်အစားကို သုတေသနတစ်ဆင့်ယောက်တဲ့ ကိစ္စဖြစ်သွားအောင် ဘာလုပ်ရမလဲ စဉ်းစားရတယ်။ ကွန်နာဝါးခဲ့ လမ်းကို ရှင်းတဲ့ အခုအခု ကွန်နာ လေးခုပဲ ရှင်းစရာကျန်အောင် ဘာလုပ်မလဲ။ ယဉ်ဘုယျပြောရင်  $n$  ကွန်နာဆိုရင်  $n - 1$  ကွန်နာ ရှင်းဖို့လို့တော့အောင် ဘာလုပ်မလဲ။

လမ်းတစ်လမ်းရှင်းတဲ့ အခု လမ်းအစ သို့မဟုတ် လမ်းအဆုံးမှာ အခြားဘက်စွန်းကို မျက်နှာမှုထားတယ်။ ရှုံးတစ်ကွန်နာ ရွှေလိုက်ရင် ရှင်းစရာ ကွန်နာတစ်ခု လျော့သွားမှာပါ။

တစ်ဆင့်ယောက်တဲ့ ကိစ္စ  $n - 1$  ကို ဖြေရှင်းနိုင်ပြီးသားလို့ မှတ်ယူပြီး လက်ရှိကစ္စ  $n$  ကို ဘယ်လို့ ဖြေရှင်းမလဲ စဉ်းစားရပါမယ်။

ကွန်နာဝါးခဲ့ရှင်းမယ်ဆိုရင် လေးခုကို ရှင်းနိုင်ပြီးသား မှတ်ယူထားရမယ်။ ဘိပါရှိရင်ကောက်၊ ရှုံးတစ်ကွန်နာ ရွှေထားလိုက်ရင် ရှင်းစရာ လေးခုပဲကျန်မယ်။ ဒီလေးခုကို လက်ရှိသတ်မှတ်နေတဲ့ ဖန်ရှင်းနဲ့ ရှင်းလိုက်ရုံးပေါ့။

```
def clean_street():
    ...
    if beepers_present():
        pick_beeper()
    move()
    clean_street() # assuming already works for n - 1
    ...

```

အသေးငယ်ဆုံး ကိစ္စကို ချင်းချက်အနေနဲ့ စဉ်းစားရမယ်။ ကွန်နာတစ်ခုဟာ အသေးငယ်ဆုံးကိစ္စ ဖြစ်တယ်။ ဒီကိစ္စကို ဘယ်လို့ ဖြေရှင်းမလဲ။

ကွန်နာတစ်ခုပဲ ရှိတယ်ဆိုရင် ဘိပါရှိရင် ကောက်လိုက်ရုံးပေါ့။  $n \neq n - 1$  ကွန်နာ အတွက် အထက်ပါ

အတိုင်း စဉ်းစားတဲ့အခါ  $n > 1$  လို့ ယူဆရမှာပါ။ ရှေ့မှာရှင်းနေရင်  $n > 1$  မို့လို့၊ မရှင်းတော့ဘူးဆိုရင်  $n = 1$  ဖြစ်နေပြီ။

```
def clean_street():
    if front_is_clear():           # ရှေ့မှာ ကွန်နာတွေ ရှိနေသေးရင်
        if beepers_present():
            pick_beeper()
            move()
            clean_street()
    else:                         # နောက်ဆုံးကွန်နာဆိုရင်
        if beepers_present():
            pick_beeper()
```

အသေးဆုံးကနေစပြီး ဖန်ရှင် အလုပ်လုပ်တာ မှန်/မမှန် စိစစ်ပါ။ ကွန်နာတစ်ခုပဲ ရှိတဲ့လမ်းဆိုရင် စစချင်းပဲ ရှေ့မှာ ပိတ်နေမှာပါ။ else အပိုင်း အလုပ်လုပ်မယ်။ ဘိပါရှိရင် ကောက်တယ်။ ကွန်နာတစ်ခု ရှိတယ်ဆိုရင် စစချင်း ရှေ့မှာ နံရုံမရှိဘူး။ if အပိုင်း အလုပ်လုပ်မယ် ဘိပါရှိရင် ကောက်တယ် ရှေ့တိုးတယ် (နံရုံပိတ်သွားပြီ)။ ရိုကားဆိုစ်ကောလ် ဖြစ်တယ်။ else အပိုင်းလုပ်ပြီးတာနဲ့ return စဖြစ်တယ်။

ဒီအတိုင်းဆက်စစ်သွားရင် တစ်ခုထက်ပိုတဲ့ ကွန်နာတွေအတွက်လည်း မှန်အောင် အလုပ်လုပ်နေမယ်ဆိုတာ သက်သေပြန်ပါတယ်။ စာရေးသူ အတွေးအကြံအရ အသေးဆုံးနဲ့ သူထက်ကြီးတာ နှစ်ခုသုံးခဲ့လေက်ထိ မှန်တယ်ဆိုရင် နောက်ဟာတွေအတွက် မှားစရာ အကြောင်းမရှိတော့ဘူး။

ရိုကားဆိုစ်နည်းနဲ့ လမ်းတစ်လမ်း ရှင်းလို့ရပါပြီ။ ကားရဲ့လုပ်ကမ္ဘာတစ်ခုလုံး ရှင်းဖို့ ရိုကားဆိုစ်ဖူး ဆက်ပြီး စဉ်းစားပါမယ်။

- လမ်းအရေအတွက်ဟာ အရွယ်အစားလို့ ယူဆနိုင်တယ်။ အသေးဆုံးက လမ်းတစ်လမ်းပါ။
- ငါးလမ်းရှိရင် ကျွန်တဲ့လေးလမ်းကို ရှင်းနိုင်ပြီးသား မှတ်ယူရမယ်။
- (၁) လမ်းရှင်းပြီး အဆုံးမှာ အပေါ်လမ်း ကူးလိုက်ရင် လေးလမ်းပဲကျွန်မယ် (လမ်းအရေအတွက်  $n$  ရှိရာကနေ  $n - 1$  ဖြစ်အောင် ဘာလုပ်မလဲ စဉ်းစားတာ)။
- တစ်ဆင့်ယောက်တဲ့ ကိစ္စ  $n - 1$  ကို ဖြဖော်ပိုင်ပြီးသားလို့ မှတ်ယူပြီး လက်ရှိကစွဲ  $n$  ကို ဘယ်လို့ ဖြဖော်ရှင်းမလဲ စဉ်းစားရပါမယ်။

```
def clean_world():
    ...
    clean_street()
    turn_north()
    change_street()
    clean_world()
    ...
```

တစ်လမ်းရှင်းပြီး နောက်တစ်လမ်းကို ကူးလိုက်ရင် ဆက်ရှင်းဖို့ လမ်း အရေအတွက်  $n - 1$  ကျွန်မယ် (ငါးလမ်းရှိတာကို တစ်လမ်းရှင်းပြီး အပေါ်လမ်းကူးလိုက်ရင် ရှင်းစရာ လမ်း လေးခုပဲ ကျွန်မယ်)။ ကျွန်တဲ့  $n - 1$  လမ်းကို ရှင်းဖို့ လက်ရှိ clean\_world ဖန်ရှင်းကိုပဲ ပြန်ခေါ်တယ်။

- အသေးငယ်ဆုံး ကိစ္စကို ချင်းချက်အနေနဲ့ စဉ်းစားရမယ်။ လမ်းတစ်လမ်းပဲရှိတာက အသေးငယ်ဆုံး။ လမ်းတစ်လမ်းရှင်းပြီး မြောက်ဘက်လှည့်အပြီး ပိတ်နေပြီးဆိုရင်တော့ နောက်ထပ် ဆက်ပြီး ရှင်းစရာ လမ်းမရှိတော့ဘူး။ အပေါ်အဆင့်က လမ်းကူးတာနဲ့ ကျွန်တဲ့လမ်းတွေကို ရှင်းတဲ့ကိစ္စကို

မြောက်ဘက်လှည့်ပြီးတဲ့အခါ ရှေ့မှာရှင်းနေမှ လုပ်ရမှာပါ။ ဒီတော့ အခုလို

```
def clean_world():
    clean_street()
    turn_north()
    if front_is_clear():
        change_street()
        clean_world()
```

ဖွစ်ရမယ်။

တစ်လမ်း၊ နှစ်လမ်း၊ သုံးလမ်း အသေးဆုံး ကိစ္စတွေ မှန်/မမှန် စိစစ်ကြည့်ပါ။ ပရိုဂရမ် အစအဆုံး ဖော်ပြ ပေးထားပါတယ်။ လေ့လာကြည့်ပါ။

```
# File: clean_world_recur1.py
from stanfordkarel import *

def main():
    clean_world()

def clean_world():
    clean_street()
    turn_north()
    if front_is_clear():
        change_street()
        clean_world()

def clean_street():
    if front_is_clear():          # ရှေ့မှာ ကွန်နာတွေ ရှိနေသေးရင်
        if beepers_present():
            pick_beeper()
        move()
        clean_street()
    else:                      # နောက်ဆုံးကွန်နာဆိုရင်
        if beepers_present():
            pick_beeper()

def change_street():
    move()
    if right_is_blocked():
        turn_left()
    else:
        turn_right()

def turn_right():
    turn_left()
    turn_left()
```

```
turn_left()

def turn_north():
    while not_facing_north():
        turn_left()

if __name__ == "__main__":
    run_karel_program("clean_world")

# File: checkerboard_recur.py
from stanfordkarel import *

def main():
    mk_checkerboard()

def mk_checkerboard():
    mk_checker_row()
    turn_north()
    if front_is_clear():
        if beepers_present():
            switch_row()
            mk_checker_row2()
        else:
            switch_row()
            mk_checker_row()
    turn_north()
    if front_is_clear():
        switch_row()
        mk_checkerboard()

def mk_checker_row():
    put_beeper()
    if front_is_clear():
        move()
    if front_is_clear():
        move()
        mk_checker_row()

def mk_checker_row2():
    if front_is_clear():
        move()
        put_beeper()
    if front_is_clear():
        move()
        mk_checker_row2()
```

```
def switch_row():
    move()
    if right_is_blocked():
        turn_left()
    else:
        turn_right()

def turn_right():
    turn_left()
    turn_left()
    turn_left()

def turn_north():
    while not_facing_north():
        turn_left()

if __name__ == "__main__":
    run_karel_program("4x5")
```

# အခန်း ၅

## ဒေတာများနှင့် ဖန်ရှင်များ

ရှုပိုင်း ကားရဲလ်အခန်း လေးခုမှာ လေ့လာခဲ့ကြတဲ့ အကြောင်းအရာတွေဟာ ပရီဂရမ်မင်း ဘာသာရပ်ရဲ့ အခြေခံအကျခုံး ပင်မထောက်တိုင် သဘောတရားတွေလို့ ဆိုရမှာပါ။ ဒီသဘောတရားတွေ မကျေည်က ဘဲ ပရီဂရမ်ရေးလို့ မရပါဘူး။ ကွန်ဒီဇိုင်နယ်တွေဖြစ်တဲ့ if, if...else ပြန်ကျော်ခြင်းအတွက် for နဲ့ while loop၊ ဖန်ရှင်တွေ၊ top-down, bottom-up ပရီဂရမ်မင်း၊ ရီကားရှင်းနဲ့ ရီကားဆစ်စ် စဉ်းစား ခြင်း စတာတွေနဲ့ ပရီဂရမ်မင်း ပူးကြော်တွေ ဖြေရှင်းနိုင်ရင် ပရီဂရမ်မာလောကား ပထမ တစ်ထစ် တက်လှမ်း အောင်မြှင့်ပြီ ပြောနိုင်ပါတယ်။ ဒီသဘောတရားတွေကို ဘိုင်နာတွေ အရိုးရှင်းဆုံးနည်းနဲ့ နားလည်အောင် လေ့ကျင့်လို့ရအောင် ကားရဲလ်က ထောက်ကာပေးတာပါ။ စက်ရုပ်လေး ကားရဲလ်ကို နှုတ်ဆက်ခဲ့ပြီး အခု ဆက်လက်လေ့လာကြမှာကတော့ ဒေတာ၊ အိပ်စံပရက်ရှင်းနဲ့ ဗော်ရောဘဲလို့ အကြောင်းပါ။

ကွန်ပျိုးတာတွေဟာ အချက်အလက် (ဒေတာ) အမျိုးမျိုးကို ကိုင်တွယ်ဆောင်ရွက်ပေးနိုင်တယ်။ ကိုန်း ကဏ္ဍားတွေအပြင် စာသား၊ ရုပ်သံ စတာတွေကိုပါ လက်ခံ အလုပ်လုပ်ပေးနိုင်တယ်။ ဒီလိုလုပ်ဆောင်နိုင်စွမ်း ဟာ ကွန်ပျိုးတာတွေကို နယ်ပယ်ပေါင်းစုံမှာ တွင်တွင်ကျယ်ကျယ် အသုံးချလာရခြင်းရဲ့ အဓိက အကြောင်း အရင်း ဆိုရင်လည်း မမှားဘူး။

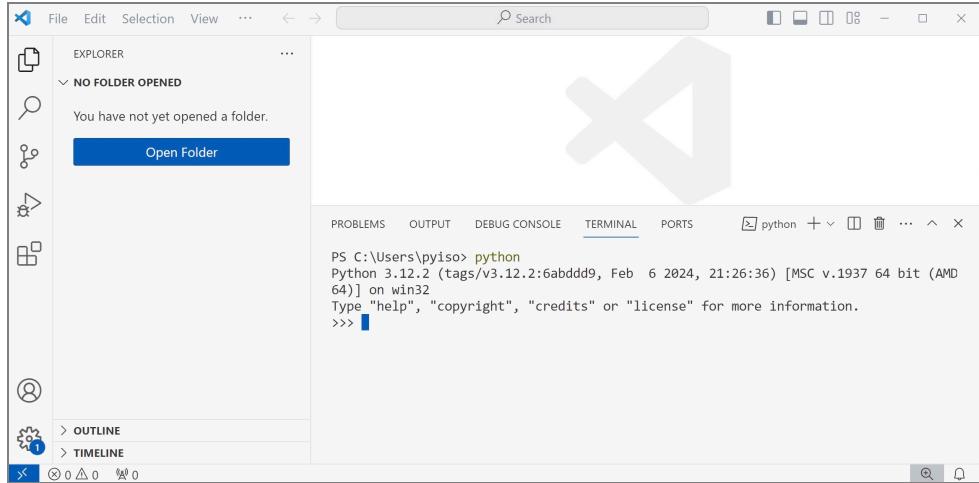
ဒေတာ အမျိုးအစား အများအပြားရှိပေမဲ့ အခြေခံအကျခုံးက ကိုန်းဂဏန်းတွေပါ။ ကွန်ပျိုးတာတွေ ကို စတင်တိထွင်ဖို့ ကြိုးစားလာကြတဲ့ အဓိက အကြောင်းအရင်းကလည်း ဂဏန်းသချာ အတွက်အချက် တွေကို လုပ်ဆောင်ရာမှာ လူတွေကို အကောအညီ ပေးဖို့အတွက်ပဲလို့ ဆိုရိုင်ပါတယ်။ ဒါအပြင် စာသား၊ ရုပ်သံ စတဲ့ အခြားဒေတာ အမျိုးအစားတွေကို ကွန်ပျိုးတာထဲမှာ ဖော်ပြုသိမ်းဆည်းထားဖို့အတွက် ကိုန်းဂဏန်းတွေကိုပဲ အသုံးပြုထားတယ်ဆိုတာ နောက်ပိုင်းမှာ နားလည်သိမြင် လာပါလိမ့်မယ်။ ဒါကြောင့်လည်း ယနေ့ခေတ် ကွန်ပျိုးတာတွေကို အစိုက်အစီး ကွန်ပျိုးတာတွေပေါ့။

### ၅.၁ ကိန်းဂဏန်းများ

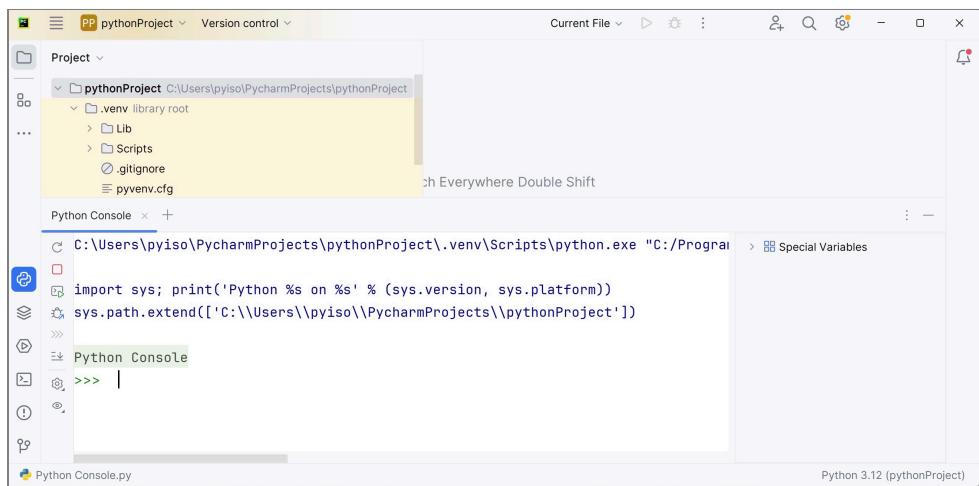
Python အင်တာပရက်တာနဲ့ Python ကုဒ်တွေကို run လိုရတဲ့နည်း နှစ်ခုရှိပါတယ်။ တစ်နည်းက ကုဒ်တွေကို .py ဖိုင်နဲ့သိမ်းပြီး python ကွန်မန်းနဲ့ run တာပါ။ ဒါက ကားရဲလ် ပရီဂရမ်တွေမှာ သုံးခဲ့တဲ့နည်း၊ နောက်တစ်နည်းကတော့ ကုဒ်တစ်ကြောင်းချင်း ရိုက်ထည့်ပြီး run တဲ့နည်းပါ။ ဒီနည်းလမ်းက interactive mode နဲ့ အင်တာပရက်တာကို အသုံးပြုတာပါ။ တစ်နည်းအားဖြင့် ကိုယ်က ကုဒ်တစ်ကြောင်းချင်း ရိုက်ထည့်ပေးပြီး အင်တာပရက်တာကလည်း အဲဒီတစ်ကြောင်းချင်းကို run ပြီး ရလဒ်ကို ပြပေးပါတယ်။ အခုအခန်းအတွက် interactive mode နဲ့ သုံးပါမယ်။ ကုဒ်တစ်ကြောင်းချင်း စမ်းသပ်

## ကြည့်ရတာ လွယ်တဲ့အတွက်ကြောင့်ပါ။

ဝင်းဒီ: Command Prompt သိမဟုတ် မက်ခံအိအက်စ် Terminal မှ python ကွန်မန်း run ပြီး interactive mode ကို ဝင်နိုင်ပါတယ်။ VS Code မှာပဲ သုံးချင်လည်း ရတယ်။ View မိန္ဒာမှု Terminal ကိုဖွံ့ဖြိုး (Ctrl + ` ရှေ့ကတ်ကိုနဲ့ ဖွံ့ဖြိုးလည်းရတယ်) ပြီး python ကွန်းမန်း run ရုံး။ PyCharm မှာဆိုရင် Python Console အိုင်ကွန်နိုင်ပြီး ဝင်ရပါမယ်။ ပုံ (၅.၁)၊ (၅.၂) တွင်ကြည့်ပါ။



ပုံ ၅.၁ VS Code Python Console



ပုံ ၅.၂ PyCharm Python Console

`2 + 5` ထည့်ပြီး Enter ကိုနိုင်ပါ။ အင်တာပရက်တာက ရလဒ် 7 နဲ့ တံ့ပြန် လုပ်ဆောင်ပေးပါလိမ့်မယ်။ ဒီလို့ အင်တာပရက်တာက လုပ်ဆောင်ပေးတာကို *evaluate* လုပ်တယ်လို့ ပြောတယ်။

```
>>> 2 + 5
```

7

အောက်ပါအတိုင်း တစ်ခုပြီးတစ်ခု ဆက်လက်စမ်းသပ်ကြည့်ပါ။

```
>>> 2 + 2
4
>>> 3 * 3
9
>>> 4 - 2
2
>>> 5/2
2.5
```

$3 \times 3 \text{ ကို } 3 * 3 \text{ လို ရိုက်ထည့်ပေးရပြီး } 5 \div 2 \text{ အတွက် } 5 / 2 \text{ လို ရေးရတာကို သတိပြုမိမှာ ပါ။ Programming language အများစုံမှာ * (asterisk) အမြဲက်သက်တာဖြစ် အသုံးပြုပြီး / (forward slash) ကို အစားသက်တာအနေနဲ့ အသုံးပြုလေ့ရှိတယ်။$

### Values and Types

တန်ဖိုးတိုင်းဟာ တိုက်ပ် (type) တစ်မျိုးမျိုးမှာ ပါဝင်ပါတယ်။ -3, 0, 2 စတဲ့ တန်ဖိုးတွေဟာ int (integer ရဲ့ အတိုက်ပ်) တိုက်ပ်ဖြစ်ပြီး -3.0, 0.1, 3.3333 စတေတွေက float တိုက်ပ် ဖြစ်ပါတယ်။ ဒဿ်မကိုန်းတွေကို ဂွန်ပျူးတာနဲ့ ဖော်ပြန့် floating point လိုခေါ်တဲ့ နည်းစနစ်ကို အသုံးပြုတယ်။ ဒဿ်မကိုန်း အတွက်အချက်တွေကိုလည်း ဒီနည်းစနစ်ကို အခြေခံပြီး ဂွန်ပျူးတာက လုပ်ဆောင်တာပါ။ ဒါကြောင့် floating point ဟာ ဒဿ်မကိုန်းတွေကို ဖော်ပြန့်နဲ့ ဒဿ်မကိုန်း အတွက်အချက်တော့ လုပ်ဆောင်ဖို့ တိတွင်ထားတဲ့ နည်းစနစ်တစ်ခုလို ဆိုနိုင်ပါတယ်။ ဒီစနစ်ကို အခြေခံထားတဲ့ ဒဿ်မကိုန်းတွေကို programming language တွေမှာ float တိုက်ပ်လို့ ခေါ်တာပါ။

float တိုက်ပ်ဟာ လိုသလောက် တိကျလိုမာရတဲ့ သဘောရှိတယ်။ အောက်ပါအတိုင်း စမ်းကြည့်ရင် 0.3 နဲ့ 1.0 ရသင့်တာ ဖြစ်ပေမဲ့ အတိအကျ အဖြေဖွေတွက်ပါဘူး။

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1;
0.9999999999999999
```

ကွဲခြားချက်က မဆိုစလောက် သေးငယ်တယ် ဆုံးပေမဲ့ ဒီအချက်ကို ပရှုပြုမိဖို့ လိုပါတယ်။ Floating point စနစ်ဟာ လုံးဝကြီးတိကျဖို့ မလိုအပ်တဲ့ (တနည်းအားဖြင့် မဆိုစလောက် သေးငယ်တဲ့ ကွာဟာချက်ကို လက်ခံနိုင်တဲ့) ကိန်းကဏ္ဍးအတွက်အချက် ကိစ္စတွေအတွက် ရည်ရွယ်တာပါ။ သိပုံနဲ့ နည်းပညာဆိုင်ရာ တိုင်းတာ တွက်ချက်မှုတွေအတွက် အသုံးပြုလေ့ရှိတယ်။ ဒဿ်မကိုန်းတွေ လုံးဝအတိအကျ ဖြစ်ဖို့ လိုအပ်တဲ့ ကိစ္စမျိုးတွေ (ဥပမာအားဖြင့် ငွေကြေးကိစ္စ အတွက်အချက်) မှာ အသုံးမပြုသင့်ပါဘူး။ ဆယ်ပြားကို 0.1 နဲ့ဖော်ပြရင် ဆယ်ပြားစေ ဆယ်စွေားတော့ တစ်ကျပ် ဖြစ်ကို ဖြစ်သင့်ပြီး 0.9999999999999999 မဖြစ်သင့်ဘူး။ ဒီလိုကိစ္စမျိုးတွေအတွက် Python မှာ Decimal ကို အသုံးပြုနိုင်ပါတယ်။ လောလောဆယ် ကိန်းကဏ္ဍးတွေနဲ့ ပါတ်သက်ပြီး စကတည်းက သိထားသင့်တာတချို့ကို ကြိုးကြော်ထားတာပါ။ Decimal တိုက်ပ် အကြောင်း မကြေခင် လေ့လာမှာပါ။

```
>>> from decimal import *
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1')
Decimal('0.3')
```

int တိုက်ပ် ဖော်ပြနိုင်တဲ့ အကြီးဆုံး အပေါင်းကိန်းပြည့် သို့မဟုတ် အငယ်ဆုံး အနှုတ်ကိန်းပြည့် တန်ဖိုးကို ကန့်သတ်ထားတာ မရှိဘူး။ သိဒ္ဓရီအရ ကြိုးလို့/ငယ်လို့ ရပါတယ်။ လက်တွေ

မှာတော့ အသုပြုတဲ့ ကွန်ပျူးတာစနစ်ပေါ် မူတည်ပြီး ကန့်သတ်ချက်ရှိမှာပါ။ ဒီစာရေးနေတဲ့ ကွန်ပျူးတာမှာ အောက်ပါကိန်းပြည့်တွေကို အသာလေး တွက်ချက်ပေးနိုင်ပါတယ်။ ဒို့ထက်အများကြီးကြီးတဲ့ဟာတော့ ကိုလည်း ကိုင်တယ်တဲ့က်ချက်ရှိနိုင်အုံမှာပါ။ တော်ရုံကိစ္စတော့အတွက် ကွန်ပျူးတာစနစ်တစ်ခုခဲ့ လက်တွေ့ကန့်သတ်ချက်ကို ကျော်လိန်သွားဖို့ မလွယ်ပါဘူး။

```
>>> 90000000000000000000000000000000 + 5000000000  
9000000000000000000000000000000050000000000  
>>> -90000000000000000000000000000000 + 1  
-899999999999999999999999999999999999999
```

float กเต็ม int နဲ့ မတူဘဲ အကြိုးဆုံး/အငယ်ဆုံး ကန်သတ်ချက်ရှိပါတယ်။ ဒီကန်သတ်ချက် ကလည်း ကွန်ပူးတာစနစ်ပေါ် မူတည်ပါတယ်။ တစ်ကိုယ်ရောင်း ကွန်ပူးတာ အများစုအတွက်  $1 \times 10^{400}$  (၁ နောက် သုည အလုံး ၄၀၀) သို့မဟုတ်  $-1 \times 10^{400}$  (အနှစ် ၁ နောက် သုည အလုံး ၄၀၀) ဟာ ကန်သတ်ချက်ကော်လှန်ပါတယ်။

```
>>> 1e400  
inf  
>>> -1e400  
-inf  
>>> 1e-400  
0.0  
>>> -1e-400  
-0.0
```

အသုတေသနများကို e အမှတ်အသားနဲ့ရေးထားတာပါ။ အကွဲရာ e နောက် လိုက်တဲ့ ကိန်းပြည့်ကဏ္ဍးကို 10 ထပ်ကိန်းလို့ မှတ်ယူရပါတယ်။ e3 က  $1 \times 10^{-3}$ , e-3 က  $1 \times 10^{-3}$  ပါ။ အကွဲရာ E အကြီးနဲ့လည်း ရေးနိုင်တယ်။ ထပ်ကိန်း [ကြီးလွန်းတဲ့အတွက် ကန့်သတ်ချက် ကျဉ်လွန်ရင် inf (အနှစ်ကိန်းခံဗျင် -inf ) ရပါမယ်။ Infinity ကို ဆိုလိုတာပါ။ (၁) မပြည့်တဲ့ ဟမာဏ သေးငယ်လွန်းတဲ့ ကဏ္ဍးတွေကိုလည်း အနီးစပ်ဆုံး သူညာအနေနဲ့ ယူပါတယ်။ အနီးစပ်ဆုံးယူတဲ့အခါ အပေါင်း/အနှစ်ကိုတော့ ခွဲခြားပေးပါတယ်။ e အမှတ်အသားနဲ့ရေးထားတဲ့ နောက်ထပ် ပုံမှန်လေး နစ်ခုပါ

```
7.34767309e22    # mass of the moon in kg  
9.1093837015e-31 # mass of an electron in kg
```

ଗଣକ୍ସା: ଅଧିକାରୀଙ୍କ ବୁଝାର୍ଥିତ ପରିମାଣ 100,500 (ଟଙ୍କାର୍ଥିକ୍ସନ୍‌ଡିଲ୍‌ର୍) । 1,500,000 (ଟଙ୍କାର୍ଥିକ୍ସନ୍‌ଡିଲ୍‌ର୍କିମ୍ବା) ଯଦୁପ୍ରତିକର୍ତ୍ତା ବୁଝାର୍ଥିତ ହେଉଥିଲା ଏବଂ Python ମୁଣ୍ଡରେ କେବଳାକାରୀଙ୍କ ପରିମାଣ ହେଉଥିଲା ।

```
>>> 1_500_000 + 100_500  
1600500  
>>> 200_000.33 + 3_800_000.22  
4000000.55000000093
```

## တိုက်ပဲမတူသည့် ကိန်းဂဏန်း အိပ်စံပရက်ရှင်များ

အိပ်စံပရက်ရှင်တွေမှာ ပါဝင်တဲ့ တိုက်ပဲ တစ်မျိုးတည်း ဖြစ်ရမယ် မရှိပါ။ တိုက်ပဲမတူတဲ့ ကိန်းဂဏန်းတွေ အိပ်စံပရက်ရှင်တစ်ခုမှာ ရောပြီး ပါဝင်နိုင်ပါတယ်။

```
>>> 5 - 2.0
3.0
>>> 5 - 2
3
>>> 3 * 2.0
6.0
>>> 3 * 2
6
```

`int` နဲ့ `float` ရောနေရင် အိပ်စံပရက်ရှင် ရလဒ်သည် `float` တိုက်ပဲ ဖြစ်မှာပါ။ အစား (division) မှာတော့ အင်တီဂျာအချင်းချင်း စားတဲ့အခါမှာလည်း ရလဒ်က `float` ဖြစ်ပါမယ်။

```
>>> 9/3
3.0
>>> 9.12/3.3
2.7636363636363637
>>> 88/3
29.333333333333332
>>> 1/3
0.3333333333333333
>>>
```

## အင်တီဂျာ ဒီပီးရှင်း၊ မော်ဒျူလို နှင့် ထပ်ကိန်းတင်ခြင်း

အကယ်၍ ဒေသမကိန်းမထွက်ဘဲ ကိန်းပြည့် လိုချင်ရင် // ကို သုံးရပါမယ်။ ဒီအခါ အကြောင်းကိုဖယ်ပြီး စားလဒ်ကိုပဲ ကိန်းပြည့်အနေနဲ့ ရမှာပါ။

```
>>> 9//3
3
>>> 12//5
2
>>> 3//5
0
```

သချာမှာ ဒီလိုမျိုး အစားကို အင်တီဂျာ ဒီပီးရှင်း (integer division) လိုခေါ်ပါတယ်။ အကြောင်းရှာမယ် ဆိုရင် % အော်ပရိတ်တာ ရှိပါတယ်။ % ကို မော်ဒျူလို (modulo) အော်ပရိတ်တာလို့ ခေါ်တယ်။ remainder အော်ပရိတ်တာလို့လည်း ခေါ်တယ်။

```
>>> 7 % 5
2
>>> 100 % 10
0
```

အင်တိဂျာ ဒီပီရှင်းနဲ့ မော်ဒူလိုကို အနှစ်ကိန်းတွေနဲ့ သုံးမယ်ဆိုရင် သတိပြုပါ။ စားလဒ် အနှစ် ကိန်း ဖြစ်ရင် // က ပိုင်ယ်တဲ့ အနှစ်ကိန်းကို အနီးစပ်ဆုံး ယူမှာပါ။ တစ်နည်းအားဖြင့် round down လုပ်တာ ဖြစ်တယ်။

```
>>> -12 // -10
1
>>> -12 // 10
-2
>>> 12 // -10
-2
>>> -31 // 10
-4
>>> -35 // 10
-4
>>> -38 // 10
-4
```

-2 နဲ့ -4 ထွက်တာ သတိပြုပါ။ အဖြေအတိအကျက် -1.2 ကို အနီးစပ်ဆုံး သုတက်ပိုင်ယ်တဲ့ -2 ကို အနီးစပ်ဆုံး ယူတယ်။ -3.1, -3.5, -3.8 တို့ကိုလည်း အနီးစပ်ဆုံး -4 ယူတာပါ။

မော်ဒူလို အော်ပရိတ်တာ % သုံးတဲ့အခါ ရလဒ်ဟာ စားကိန်းနဲ့ sign အပေါ် မူတည်တယ်။ (အပေါင် အနှစ်ကို ဆိုလိုတာပါ)။

```
>>> -17 % 10
3
>>> 17 % -10
-3
```

မော်ဒူလိုနဲ့ အင်တိဂျာ ဒီပီရှင်း အော်ပရိတ်တာ နှစ်ခုက အောက်ပါညီမျှခြင်းအရ ဆက်စပ်နေတာပါ။ စားကိန်း  $B \neq 0$  ဖြစ်ပါတယ်။

$$B * (A//B) + A\%B = A$$

ဒါကြောင့်  $B = 10, A = -17$  ဖြစ်လျှင်

$$\begin{aligned} B * (A//B) + A\%B &= A \\ 10 * (-17//10) + -17\%10 &= -17 \\ -20 + -17\%10 &= -17 \\ -17\%10 &= -17 + 20 \\ -17\%10 &= 3 \end{aligned}$$

အကယ်၍  $B = -10, A = 17$  ဖြစ်လျှင်

$$\begin{aligned} B * (A//B) + A\%B &= A \\ -10 * (17// - 10) + 17\% - 10 &= 17 \\ 20 + 17\% - 10 &= 17 \\ 17\% - 10 &= 17 - 20 \\ 17\% - 10 &= -3 \end{aligned}$$

အထက်ပါ ညီမျှခြင်းဟာ ကိန်းပြည့်တွေအတွက်ပဲ မှန်တာပါ။ // နဲ့ % ကို အသေမကိန်းတွေနဲ့လည်း သုံးလို့ရပေမဲ့ ရလဒ်တွေက အထက်ပါ ညီမျှခြင်းကို ပြေလည်စေမှာ မဟုတ်ပါဘူး။ float တိုက်ပ်ဟာ

```
>>> 9.9 // 3.3
3.0
>>> 9.9 % 3.3
8.881784197001252e-16
>>> 9.9 / 3.3
3.0000000000000004
>>> 3.5 / 0.1
35.0
>>> 3.5 // 0.1
34.0
>>> 3.5 % 0.1
0.09999999999999981
```

ထပ်ကိန်းတင် (exponentiation) ဖို့ အတွက် အော်ပရိတ်တာက \*\* ပါ။  $2^4$  နဲ့  $(3.3)^3$  ကို အခါ လို တွက်ပါတယ်။

```
>>> 2 ** 4
16
>>> 3.3 ** 3
35.937
```

## သချုပ်ရှင်များ

ကိန်းကဏ္ဍားတွေအကြောင်း လေ့လာလက်စနဲ့ math လိုက်ဘရီ သချုပ်ရှင်တချို့ကိုလည်း တစ်ခါတည်း ဆက်ကြည့်လိုက်ရအောင်။ အဓိကက သချုပ်ရှင်ဆိုတာထက် ဖန်ရှင် အခြေခံအသုံးပြုပုံကို စပီးလေ့လာ မှာပါ။ math လိုက်ဘရီက Python မှာ တစ်ခါတည်း ထည့်ထားပေးပြီးသား (built-in) လိုက်ဘရီပါ။ အင်စတောလ်လုပ်စရာ မလိုဘဲ အင်ပို့လုပ်ပြီး သုံးလို့ရတယ်။

```
>>> from math import *
```

အင်ပို့လုပ်ပြီးရင် math လိုက်ဘရီဖန်ရှင်တွေကို သုံးလို့ရပါပြီ။ ကိန်းတစ်ခုခဲ့ နှစ်ထပ်ကိန်းရင်းကို sqrt, သုံးထပ်ကိန်းရင်းကို cbrt ဖန်ရှင်နဲ့ ရှာနိုင်ပါတယ်။

```
>>> cbrt(27)
3.0
>>> sqrt(81)
9.0
```

သချုပ်ရှင်အားလုံးဟာ input တန်ဖိုးတစ်ခု သို့မဟုတ် တစ်ခုထက်ပို၍ လက်ခံပြီး output တန်ဖိုး တစ်ခု ပြန်ထုတ်ပေးပါတယ်။ 27 နဲ့ 81 ဟာ input ဖြစ်ပြီး 3.0 နဲ့ 9.0 က output ဖြစ်တယ်။

```
>>> gcd(2406, 654)
```

```
>>> gcd(2406, 654, 354)
6
>>> gcd(2406)
2406
```

အကြီးဆုံးဘုံးဆွဲကိန်းကို `gcd` ဖန်ရှင်နဲ့ ရှာတာပါ။ အင်ဟိဂျာ `input` တစ်ခုနဲ့အထက် လက်ခံတဲ့ ဖန်ရှင်ဖြစ်တယ်။ `input` ကဏ္ဍားအားလုံးကို စားလို့ပြတ်တဲ့ အကြီးဆုံးကိန်းကို ရှာပေးတယ်။ ကိန်းပြည့်မဟုတ်တာ ထည့်ရင် အယူရာဖြစ်ပါတယ်။

```
>>> gcd(2.4, 4.8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
```

လေ့ကရှစ်သမ်း ထံရှိခို့မေတ္တရီ ဖန်ရှင်တွေလည်းပါတယ်။  $\log_{10}(x)$ ,  $\sin(x)$ ,  $\cos(x)$  တို့ကို ဥပမာပြထားတာ ကြည့်ပါ။

```
>>> log10(1000)
3.0
>>> sin(pi/2) # pi/2 radians = 90 degrees
1.0
>>> sin(pi/4) ** 2 + cos(pi/4) ** 2
1.0
```

## ၅.J ‘တိုက်ပ်’ ဆိုတာ ဘာလဲ

Programming language အားလုံးမှာ တိုက်ပ် သိမဟုတ် ဒေတာတိုက်ပ် သဘောတရား ပါရှိပါတယ်။ `int` နဲ့ `float` တိုက်ပ်မှာ ပါဝင်တဲ့ ကိန်းပြည့်တွေနဲ့ ဒုသေမကိန်းတွေကို မိတ်ဆက်ပြီးတဲ့အခါ ‘တိုက်ပ်’ ဆိုတာဘာလဲ တိတိကျကျ ရှင်းပြလို့ရပါပြီ။ တိုက်ပ်တစ်ခုဟာ

- တန်ဖိုးတွေပါဝင်တဲ့ အစ (set) တစ်ခု နဲ့
- ငှုံးတန်ဖိုးများအပေါ်တွင် အသုံးချိန်းတဲ့ အော်ပရေးရှင်းတွေ ပါဝင်တဲ့ အစုတစ်ခု ဖြစ်ပါတယ်။ ဥပမာ `int` တိုက်ပ်ကို ကြည့်ရင် ကိန်းပြည့်တွေ ပါဝင်တဲ့ အစုနဲ့ ကိန်းပြည့်တွေအပေါ်မှာ လုပ်ဆောင်လို့ရတဲ့ အော်ပရေးရှင်းတွေ ပါဝင်တဲ့ အစ

```
{..., -3, -2, -1, 0, 1, 2, 3, ...}
{+, -, *, /, //, %, **, ...}
```

ဖြစ်ပါတယ်။ `float` တိုက်ပ်ကတော့ ကိန်းစစ် (real numbers) တွေပါဝင်တဲ့ အစုနဲ့ ငှုံးတို့အပေါ်မှာ လုပ်ဆောင်လို့ရတဲ့ အော်ပရေးရှင်းတွေ ပါဝင်တဲ့ အစုတို့ ဖြစ်ပါတယ်။ `int` နဲ့ `float` တိုက်ပ်မှာ အော်ပရေးရှင်းတွေ တူတူဖြစ်နေတာ တွေ့ရမှာပါ။ ဒီလို့အမြဲဖြစ်မယ်လို့ မယူဆရပါဘူး။ တိုက်ပ် မတူတဲ့အခါ အသုံးချလို့ ရနိုင်တဲ့ အော်ပရေးရှင်းတွေ ကွားမြားနိုင်ပါတယ်။ ဥပမာ `str` တိုက်ပ် အော်ပရေးရှင်းတွေက `int` တို့ `float` တို့နဲ့ မတူပါဘူး။ `str` က `string` ရဲ့ အတိုကောက်ဖြစ်ပြီး စာသားတွေအတွက် အသုံးပြုပါတယ်။ မကြာခင် လေ့လာကြမှာပါ။

အပေါက် အော်ပရေးရှင်း အစုမှာ အစက်သုံးစက် ... ကို သတိပြုပါ။ ဆိုလိုတာက အခြား အော်ပရေးရှင်းတွေ ဒီအစုမှာ ပါဝင်ပါသေးတယ်။ `int` နဲ့ `float` တွေအတွက် ဖန်ရှင်တွေကိုလည်း ဒီအစုမှာ

ပါဝင်တယ်လို့ ယူဆရမှာပါ။

```
>>> from math import *
>>> sqrt(2.0)
1.4142135623730951
>>> abs(-5)
5
```

ဥပမာအနေနဲ့ `sqrt` နဲ့ `abs` ဖန်ရှင် အသုံးချုပ်ပါ။ နှစ်ထပ်ကိန်းရှင်းနဲ့ ပကဗ္ဗာတန်ဖိုး ရှာပေးပါတယ်။ လိုအပ်ရင် ကိုယ်ပိုင်ဖန်ရှင်တွေ သတ်မှတ်ပြီး တိုက်ပ်တစ်ခုရဲ့ အော်ပရေးရှင်းတွေကို ဖြည့်စွက်တိုးချဲ့နိုင်ပါတယ်။

အော်ပရေးရှင်းနဲ့ အော်ပရိတ်တာ ရောထွေးစရာ ရှိပါတယ်။ `+, -, *, /, //, %, **` စတဲ့ သက်ကတွေကို အော်ပရိတ်တာလို့ ခေါ်ပါတယ်။ အော်ပရေးရှင်း လုပ်ဆောင်ဖို့အတွက် အသုံးပြုတဲ့ သက်ကတွေကို အော်ပရိတ်တာလို့ ခေါ်တော့ပါ။ ဥပမာ “\* သက်ကတာဘာ အမြှောက်အော်ပရေးရှင်း လုပ်ဆောင် ဖို့ သတ်မှတ်ထားတဲ့ အော်ပရိတ်တာ” လို့ ပြောတယ်။ အမြှောက် ‘အော်ပရေးရှင်း’ ကျတော့ မြှောက်တဲ့အလုပ် ဆောက်ရွက်တာကို ဆိုလိုတာ။

## ၅.၃ ဖော်ရောဲလ်များ

ဖော်ရောဲလ်ဆိုတာ တန်ဖိုးတစ်ခုကို ကိုယ်စားပြုတဲ့ နံမည်ပါပဲ။ နံမည်နဲ့ ငြင်းကိုယ်စားပြုတဲ့ တန်ဖိုး တွဲဖက်ပေးဖို့ အဆိုင်းမန်း (*assignment*) စတိတ်မန်းကို သုံးရပါတယ်။

```
>>> age = 12
>>> weight = 35.5
```

`age` နဲ့ `weight` ဟာ ဖော်ရောဲလ်တွေ ဖြစ်ပါတယ်။ ညီမြှော်ပြင်းသက်တာ (=) ကတော့ အဆိုင်းမန်း အော်ပရိတ်တာပါ။ ဖော်ရောဲလ်နဲ့ တန်ဖိုး တွဲဖက်ပေးတဲ့ အော်ပရိတ်တာ ဖြစ်တယ်။ ဖော်ရောဲလ်နံမည်ကို variable *identifier* လို့လည်း ခေါ်ပါတယ်။ Identifier က နည်းပညာ အခေါ်အဝေါ်ပေါ့။ Variable name က သာမန်လူ နားလည်တဲ့ နည်းနဲ့ ပြောတာပါ။ ဖော်ရောဲလ် တစ်ခုချင်း ထည့်ကြည့်ရင် ငြင်းကိုယ်စားပြုတဲ့ တန်ဖိုးကို ပြန်ထုတ်ပေးတာ တွေ့ရမှာပါ။

```
>>> age
12
>>> weight
35.5
```

အိပ်စ်ပရက်ရှင်တွေက ဖော်ရောဲလ်တွေနဲ့ ဖြစ်နိုင်ပါတယ်။ အိပ်စ်ပရက်ရှင် တွေက်ချက်ရင် ဖော်ရောဲလ် တန်ဖိုးနဲ့ အစားထိုး တွေက်ချက်တယ်လို့ ယူဆရမှာပါ။ ဥပမာ

```
>>> age + 1
13
>>> weight / 2
17.75
```

ဖော်ရောဲလ်တစ်ခုကို အိပ်စ်ပရက်ရှင်ရလဒ်နဲ့ အဆိုင်းမန်း လုပ်လိုပါတယ်။ `rect_area` ကို အောက်

တွင် ကြည့်ပါ။ အလျား အနဲ့ မြောက်လဒ်ကို အဆိုင်းမနဲ့ လုပ်ထားတာ တွေ့ရပါမယ်။

```
>>> rect_width = 22.5
>>> rect_length = 10
>>> rect_area = rect_width * rect_length
>>> rect_area
225.0
```

### အဆိုင်းမနဲ့ ခတိတ်မနဲ့

ဖေရီရောဘလ်တစ်ခုဟာ အချိန်တစ်ချိန်မှာ တန်ဖိုးတစ်ခုကိုပဲ ကိုယ်စားပြုနိုင်တယ်။ ဒါပေမဲ့ အချိန်ကာလပေါ် မူတည်ပြီး တန်ဖိုးပြောင်းနိုင်တယ်။ (တစ်ချိန်တည်းမှာ တန်ဖိုးနှစ်ခု မဖြစ်နိုင်ဘူး)။ ဥပမာ x တန်ဖိုးဟာ ပထမ 10 ပါ။ ဒုတိယ အဆိုင်းမနဲ့လုပ်ပြီးတဲ့ အချိန်မှာ အဲဒီ x ကပဲ 1000 ဖြစ်နေမှုပါ။

```
>>> x = 10
>>> x
10
>>> x = 1000
>>> x
1000
```

### ၅.၄ စာသားများ

စာသား (text) ဟာ အသုံးအများဆုံး ဆက်သွယ်ဆောင်ရွက်ရေး ကြားခံနယ်တစ်ခုပါ။ ဝက်ဘ်ဆိုက် စာမျက်နှာ၊ အီးပေးလုံး၊ အီးဘွတ်ခံနဲ့ အီးလက်ထရွန်းနှစ် စာရွက်စာတမ်း (e-documents) စတာတွေမှာ ရုပ်သံတွေ အသုံးပြုလာကြပေမဲ့ စာသား အဓိကဖြစ်နေဆဲပါပဲ။ ဆိုရှယ်မိဒီယာ၊ ဂိမ်းနဲ့ အမြားအပ်ပဲတွေ ဟာလည်း စာသားနဲ့ မက်းနိုင်ကြပါဘူး။ ဒါကြောင့် ပရီဂရမ်းမင်းအတွက် စာသားဟာ ဘယ်လောက်ထိ အရေးပါကြောင်း အများကြီးပြောစရာ လိုအယ်မထင်ပါဘူး။

ပရီဂရမ်းမင်းမှာ စာသားကို string လိုပေါ်ပြီး ကာရက်တာ (character) တွေနဲ့ စီတန်ဖွဲ့စည်းထားတယ်။ ကာရက်တာဆိုတာ အခြေခံ သတင်းအချက်အလက် ယူနစ်တစ်ခုပါပဲ။ အကွ္ခာ၊ ဂဏန်း (digit)၊ သက်တ သို့မဟုတ် ကွန်ထရိုးလုံကုဒ် တစ်ခုခဲ့ ဖြစ်နိုင်ပါတယ်။ ဥပမာ A, B, C, \$, @, #, 1, 3, \_ စသည်ဖြစ်၏ Double quotes ("") တစ်စုံကြား ညှပ်ရေးထားတဲ့ ကာရက်တာတွေ အသီအတန်းလိုက်ကို စာသားအနေနဲ့ ယူဆတယ်။ Python မှာ စာသားရဲ့ တိုက်ပ်ဟာ str ဖြစ်တယ်။ string ကို အတိုကောက် ယူထားတာပါ။

```
>>> "Hello, World!"
'Hello, World!'
```

သို့မဟုတ် " အစား single quotes('') တစ်စုံလည်း သုံးနိုင်ပါတယ်။

```
>>> 'Hello, World!'
'Hello, World!'
```

စာသားတစ်ခုမှာ ပါဝင်တဲ့ ကာရက်တာ အရေအတွက်ကို len ဖန်ရှင်နဲ့ စစ်ကြည့်နိုင်ပါတယ်။ ကာရက်တာ တစ်လုံးမှ မပါတဲ့ "" (သို့ '') ကို empty string လို့ ခေါ်ပါတယ်။

```

>>> len("Hello, World!")
13
>>> long_sentence = "This is a long sentence nobody wants to read."
>>> len(long_sentence)
45
>>> len("")
0
>>> len(" ") # contain a single space
1

```

str တိုက်ပဲရဲ အခြေခံကျတဲ့ အော်ပရေးရှင်းတစ်ခုက စာသားတစ်ခုနဲ့ တစ်ခု ဆက်တာပါ။ + အော်ပရိတ်တာနဲ့ စာသားတွေကို ဆက်နိုင်ပါတယ်။

```

>>> "Yangon " + "and " + "Mandalay"
'Yangon and Mandalay'

```

စာသားအချင်းချင်းပဲ ဆက်လိုက်ပါတယ်။ စာသားနဲ့ ကိန်းဂဏန်း ဆက်လိုမာပါဘူး။ အောက်ပါအတိုင်း စမ်းကြည့်တဲ့အခါ str နဲ့ float ဆက်လိုမာဘူးလို့ အယ်ရှာမက်ဆွဲချုပ် ကျလာမှာပါ။

```

>>> from math import *
>>> pi
3.141592653589793
>>> "The value of π is " + pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "float") to str

```

str ဖန်ရှင်က ကိန်းဂဏန်းတစ်ခုကနေ စာသားကို ထုတ်ပေးပါတယ်။ မူးရှင်းကိန်းဂဏန်းကို စာသားဖြစ်အောင် ပြောင်းလိုက်တာ မဟုတ်ပါဘူး။ ကိန်းဂဏန်း တန်ဖိုးကနေ ငြင်းကိုဖော်ပြတဲ့ စာသားကို ဖန်ရှင်က ပြန်ထုတ်ပေးတာပါ။

```

>>> str(pi)
'3.141592653589793'

```

ထွက်လာတဲ့ တန်ဖိုးဟာ စာသားဖြစ်တဲ့အတွက် single quote ပါနေတာ သတိပြုပါ။ pi တန်ဖိုးနဲ့ စာသားအုပ်လို့ ဆက်ရပါမယ်။

```

>>> "The value of π is " + str(pi)
'The value of π is 3.141592653589793'

```

str ဖန်ရှင်နဲ့ စာသားရအောင် အရင်လုပ်ပြီးမှ ဆက်ထားတာပါ။ စာသားကနေ ကိန်းဂဏန်း လိုချင်ရင် int နဲ့ float ဖန်ရှင် သုံးနိုင်ပါတယ်။

```

>>> int('1024')
1024
>>> int('1024') * 2
2048
>>> float('2.4') * 3

```

| 7.1999999999999999

ကဏ္နားပြောင်းလို့မရတဲ့ စာသားဖြစ်နေရင် အယ်ရှာဖြစ်မှာပါ။

```
>>> int('1a24')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1a24'
>>> int('12.3')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.3'
```

'12.3' မှာ ဒသာမ ပါနေတာကြောင့် int ပြောင်းလို့ မရတဲ့အတွက် အယ်ရှာတက်တာပါ။  
စာသားကို \* အော်ပရီတ်တာနဲ့ ပွဲးယူလို့ရတယ်။

```
>>> 'hello' * 3
'hellohellohello'
```

'hello' သုံးခါ ဆက်လိုက်တာပါ။ string နဲ့ အကြမ်အရေအတွက် ကိန်းပြည့် ဖြစ်ရပါမယ်။ သူည် ထို့မဟုတ် အနှုတ်ကိန်း ဖြစ်နေရင် empty string ရမှာပါ။

```
>>> 'World' * -3
''
>>> 'Hello' * 0
''
```

စာသားနဲ့ ကဏ္နား ဖလှယ်လို့ရပါတယ်

```
>>> 3 * 'Hello'
'HelloHelloHello'
```

### string ပေါ်ရောဲလ်များ

ပေါ်ရောဲလ်တွေကို string တန်ဖိုးတွေအတွက်လည်း အသုံးပြုနိုင်တယ်။ အောက်ပါ အိပ်စံပရ်ရှင်တွေ ကို နားလည်နိုင်မလား ကြိုးစားကြည့်ပါ။ ပေါ်ရောဲလ် တစ်ခုချင်းကို သူ့ရဲ့တန်ဖိုးနဲ့ အစားထိုးပြီး +, \*, \* အော်ပရီတ်တာတွေ အလုပ်လုပ်ပုံး ဆက်စပ်စဉ်းစားရင် ဘာကြောင့် အခုလုံး အဖြစ်ကိုလဲ ခန်းမှန်းနိုင်မှာပါ။ သိပ်ခက်ခက်ခဲ့ခဲ့ မဟုတ်ပါဘူး။

```
>>> name = 'Kathy'
>>> first_part = 'Hello'
>>> second_part = 'How are you doing?'
>>> first_part + ', ' + name + '. ' + second_part
'Hello, Kathy. How are you doing?'
>>> (first_part + ', ') * 3 + name
'Hello, Hello, Hello, Kathy'
```

## Escape Character and Escape Sequence

String တစ်ခုရေးတဲ့အခါ ပုံမှန်အားဖြင့် လိုချင်တဲ့ စာသားအတိုင်း ကိုဘုံးကနေ ကာရက်တာ တစ်လုံးချင်း ရိုက်ရုံပါပဲ။ စပယ်ရှယ် ကာရက်တာ တချို့ကိုတော့ ကိုဘုံးကနေ တိုက်ရိုက် ရိုက်ထည့်လို့ မရဘဲ သီးခြားနည်းလမ်းတစ်ခုနဲ့ ရေးပေးရပါတယ်။ ဥပမာ စာသားထဲမှာ tab ကာရက်တာအတွက် \t နဲ့ newline အတွက် \n ရေးရမှာပါ။ ကိုဘုံးကနေ tab ကိုး enter/return ကိုး နိုပ်ပြီး တိုက်ရိုက်ထည့်လို့မရပါဘူး။ သီးခြားအဓိပ္ပာယ် တစ်ခုအတွက် \ နဲ့စတဲ့ ကာရက်တာအတွက်လိုက် ကို escape sequence လိုခေါ်ပြီး \ ကိုတော့ escape character လို့ ခေါ်ပါတယ်။

```
>>> two_lines = "Line 100\nLine 101"
>>> two_lines
'Line 100\nLine 101'
>>> tabs_eg = "Line 1\t\t1,000,000\nLine 1000\t10,000"
>>> tabs_eg
'Line 1\t\t1,000,000\nLine 1000\t10,000'
```

Escape sequence တစ်ကို **bold** ဖော်နဲ့ ပြထားပါတယ်။ Python ကွန်ဆိုးလုံး \t နဲ့ \n ကို အရှိအတိုင်း ပြနေပါတယ်။ ဒါပေမဲ့ အခုလို့ စမ်းကြည့်ရင် သိသာပါလိမ့်မယ်။

```
>>> print(two_lines)
Line 100
Line 101
>>> print(tabs_eg)
Line 1      1,000,000
Line 1000    10,000
```

Double quotes တစ်စုံနဲ့ စာသားထဲမှာ " ပါနေရင် \ " လိုရေးရပါမယ်။ Single quote တစ်စုံနဲ့ စာသားထဲမှာ ' ပါနေရင်လည်း \ ' လိုရေးရပါမယ်။

```
>>> 'I'll tell you the truth'
"I'll tell you the truth"
>>>
>>> 'I'll tell you the truth'
File "<stdin>", line 1
  'I'll tell you the truth'
  ^
SyntaxError: invalid syntax

>>> "He said, \"I am very tired\""
'He said, "I am very tired"'
>>>
>>> "He said, "I am very tired""
File "<stdin>", line 1
  "He said, "I am very tired"
  ^
SyntaxError: invalid syntax
```

Double quotes နဲ့ စာသားထဲက single quote သိမဟုတ် single quote နဲ့ စာသားထဲက

double quotes ဆိုရင်တော့ \ မလိုပါဘူး။

```
>>> "I'll tell you the truth"
"I'll tell you the truth"
>>> 'He said, "I am tired"'
'He said, "I am tired"'
```

နှစ်မျိုးလုံး ပါနေရင်တော့ တစ်မျိုးက \ ပါရပါမယ်။ ကျေန်တဲ့တစ်မျိုးကတော့ ပါရင်လည်းရာ မပါလည်း  
ပြသာနာမရှိဘူး။ အောက်ပါတို့ကို ဂရိုစိုက် လေ့လာကြည့်ပါ။

```
>>> 'He asked, "Don\'t you like?"'
'He asked, "Don\'t you like?"'
>>> "He asked, \"Don't you like?\""
'He asked, "Don\'t you like?"'
>>> "He asked, \"Don\\'t you like?\""
'He asked, "Don\\'t you like?"'
>>> 'He asked, \"Don\\'t you like?\"'
'He asked, "Don\\'t you like?"'
```

Escape Sequence	အဓိပ္ပာယ်
\'	single quote
\"	double quote
\\\	backslash
\t	tab
\n	newline
\r	carriage return

## တေတဲ့ ၅.၁ Python Escape Sequences

### ၅.၅ အိပ်စံပရက်ရှင်များ

တိုက်ပါတာဘာလဲ အကြမ်းဖျဉ်း ရှင်းပြဲ ပြီးပါပြီ။ ကိန်းဂဏန်းနဲ့ စာသား တိုက်ပါတာမျိုးကိုလည်း  
လေ့လာခဲ့ပြီးပြီ။ တကယ်တော့ အိပ်စံပရက်ရှင် (expression) ဆိုတာလည်း အသစ်အဆန်း မဟုတ်ပါဘူး။ တန်ဖိုးတစ်ခုဟာ အရှိုးရှင်းဆုံး အိပ်စံပရက်ရှင်လို့ ဆိုနိုင်ပါတယ်။ "Hello", 2.3 စတဲ့ တန်ဖိုးတွေ  
ဟာ အိပ်စံပရက်ရှင်တွေပါပဲ။ မေရ့ရောဘဲလဲဟာလည်း တန်ဖိုးကို ကိုယ်စားပြုတဲ့အတွက် အိပ်စံပရက်ရှင်  
လို့ ယူဆရမှာပါ။

ရုံးရှင်းတဲ့ အိပ်စံပရက်ရှင်တော့ အနေဖြင့် ပေါင်းစပ် အိပ်စံပရက်ရှင် (compound expression)  
တွေ ဖွဲ့စည်းတည်ဆောက် ယူနိုင်ပါတယ်။

```
>>> 2 + 5
7
>>> (3 + 2) * (2 / 5)
2.0
>>> 'Hello, ' * 3 + 'World'
```

```
'Hello, Hello, Hello, World'
```

အပိုစ်ပရက်ရှင်ကို ရှုထောင့်အမျိုးမျိုးကနေ အဓိပ္ပာယ်ဖွင့်ဆိုကြတာ တွေ့ရပါတယ်။ “အပိုစ်ပရက်ရှင် ဆိုတာ တန်ဖိုးတစ်ခု ပြန်ပေးတဲ့ အော်ပရေးရှင်း အတွဲအဆက်ဖြစ်တယ်” လိုဆိုရင် အတိုင်းအတာတစ်ခုအထိ တိတိကျကျရှိပြီး နားလည်ရလည်း လွယ်ပါတယ်။ တချို့စာအုပ်တွေမှာတော့ အပိုစ်ပရက်ရှင်ကို တန်ဖိုးပြန်ပေးတဲ့ စတိတ်မန့်လို့ သတ်မှတ်ပါတယ်။

ပထမအဓိပ္ပာယ်အရ အပိုစ်ပရက်ရှင်နဲ့ စတိတ်မန့် မတူဘူးလို့ ယူဆပါတယ်။ ဒီရှုထောင့်က ကြည့်ရင် စတိတ်မန့်ဟာလည်း အော်ပရေးရှင်း အတွဲအဆက်ဖြစ်ပေမဲ့ တန်ဖိုးပြန်မပေးဘူး။ အပိုစ်ပရက်ရှင်ကတော့ တန်ဖိုးပြန်ပေးရမှာပါ။  $3 * 2$  ကို လုပ်ဆောင်တဲ့အခါ 6 ရပါတယ်။ ဒါကြောင့်  $3 * 2$  ဟာ အပိုစ်ပရက်ရှင်ဖြစ်တယ်။ result =  $3 * 2$  က စတိတ်မန့်ဖြစ်တယ်။ အဆိုင်းမန့်ဟာ ဖေရာ့ရော့လ်ကို တန်ဖိုးတစ်ခုနဲ့ တဲ့ဖက်ပေးတာ။ တန်ဖိုးပြန်မပေးဘူး။

ဒုတိယအဓိပ္ပာယ်အရ အပိုစ်ပရက်ရှင်သည်လည်း စတိတ်မန့်ပဲ။ စတိတ်မန့်တွေကို တန်ဖိုးပြန်ပေးတဲ့ စတိတ်မန့်နဲ့ ပြန်မပေးတဲ့ စတိတ်မန့် အုပ်စုနှင့်စဲခြားတယ်။ တန်ဖိုးပြန်ပေးတဲ့ စတိတ်မန့်တွေကို အပိုစ်ပရက်ရှင် သို့မဟုတ် အပိုစ်ပရက်ရှင်စတိတ်မန့်လို့ ဒုတိယအဓိပ္ပာယ် သတ်မှတ်ချက်အရ ခေါ်တာပါ။

တန်ဖိုးပြန်ပေးတဲ့ ဖန်ရှင်ကောလ်တွေကိုလည်း အပိုစ်ပရက်ရှင်လို့ ယူဆပါတယ်။ ဥပမာ sqrt(9) က 3.0 ရပါတယ်။

```
>>> from math import *
>>> sqrt(9)
3.0
```

print(3) ကတော့ အပိုစ်ပရက်ရှင် မဟုတ်ပါဘူး။ အခုလို စမ်းကြည့်ရင် 3 ထုတ်ပေးတဲ့အတွက် အပိုစ်ပရက်ရှင်လို့ ထင်စရာ အကြောင်းရှိပါတယ်။

```
>>> print(3)
3
```

ဒါပေမဲ့ ဒါဟာ print ဖန်ရှင်က output ထုတ်ပေးတာပါ။ တန်ဖိုးပြန်ပေးတာ မဟုတ်ပါဘူး။ တန်ဖိုးပြန်ရတယ်ဆိုရင် အခြားအပိုစ်ပရက်ရှင်တစ်ခုမှာ တန်ဖိုးအနေနဲ့ သုံးလို့ရ ရမယ်။ ဥပမာ print(3) + 2 ကို စမ်းကြည့်ပါ။

```
>>> print(3) + 2
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

## ၅.၆ ဘူလီယန် တိုက်ပန့် ဘူလီယန် အီပိုစ်ပရက်ရှင်

Python မှာ bool တိုက်ပ်ဟာ ဘူလီယန် (boolean) တိုက်ပ်ကို ဆိုလိုတာဖြစ်ပြီး True နဲ့ False တန်ဖိုး နှစ်ခုပဲ ပါဝင်တယ်။ မှန်ခြင်း/မှားခြင်း၊ ရှိခြင်း/မရှိခြင်း၊ ဖြစ်ခြင်း/မဖြစ်ခြင်း စတဲ့အချက်အလက်မျိုးတွေကို ဖော်ပြန့် (boolean) တိုက်ပ်ကို အသုံးပြန်ပါတယ်။

```

is_winter = True
has_four_legs = False
is_adult = True

```

တန်ဖိုးရှာတဲ့အခါ ဘူလီယန်တိုက်ပ် ရလာမဲ့ အိပ်စ်ပရက်ရှင်တွေကို ဘူလီယန် အိပ်စ်ပရက်ရှင်လို့ ခေါ်တယ်။ အောက်ပါတို့ကို ကြည့်ပါ

```

>>> 'abc' == 'abc'
True
>>> 'Apple' < 'Opple'
True
>>> 'Opple' < 'Apple'
False
>>> 5 == 5
True
>>> 2 < 5.0
True
>>> 2.0 <= 2.0
True
>>> 2 != 2
False
>>> 2 != 3
True

```

<, >, <=, >=, ==, != စတဲ့ သက်တတွေဟာ comparison operator တွေပါ။ Relational operator တွေလိုလည်း ခေါ်ကြတယ်။ အလယ်တန်းသချုပ်မှာ သင်ခဲ့ရတဲ့ <, >, ≤, ≥, =, ≠ စတေတွဲနဲ့ အဓိပ္ပာယ်တူပါတယ်။ ညီ/မညီ စစ်ချင်ရင် ညီမျှခြင်းသက်တန်ခဲ့ == နဲ့ စစ်ရပါမယ်။ Comparison operator တွေကို ဒေတာတိုက်ပ် အမျိုးမျိုးနဲ့ တွဲဖက် အသုံးပြုနိုင်တာကို တွေ့ရမှာပါ။ တန်ဖိုးတစ်ခုနဲ့ တစ်ခု နှင့်ယူဉ်နိုင်ခြင်းဟာ အရေးပါတဲ့ ကိစ္စဖြစ်ပါတယ်။

Python မှာ True == 1, False == 0 ဖြစ်တယ်လို့ သတ်မှတ်ပါတယ်။

```

>>> True == 1
True
>>> False == 0
True
>>> True == 5
False

```

ဘူလီယန် အိပ်စ်ပရက်ရှင်တွေနဲ့ ပါတ်သက်ပြီး နောက်ထပ် သိထားရမဲ့ အော်ပရိတ်တာ သုံးခကေတ္တာ and, or, not တို့ပဲ ဖြစ်ပါတယ်။ ငါးတို့ကို ဘူလီယန်အော်ပရိတ်တာလို့ ခေါ်ပြီး ဘူလီယန်တန်ဖိုး (သို့) ဘူလီယန် အိပ်စ်ပရက်ရှင်တွေနဲ့ တွဲဖက်အသုံးပြုရပါတယ်။

ဘူလီယန်တန်ဖိုး/အိပ်စ်ပရက်ရှင် နှစ်ခုမှာ နှစ်ခုလုံး မှန်/မမှန် စစ်ကြည့်ချင်တဲ့အခါ and အော်ပရိတ်တာ သုံးရပါမယ်။ နှစ်ခုလုံး အမှန်ဖြစ်မှ True ရပါတယ်။

```

>>> True and True
True

```

```
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

ကိန်းကဏ္ဍားတစ်ခုက တစ်နဲ့ တစ်ဆယ်ကြား ရှိ/မရှိ အခုလိုစစ်ဆိုပါတယ်

```
>>> x = 5
>>> x > 1 and x < 10
True
>>> y = 11
>>> y > 1 and y < 10
False
```

တစ်ထက်လည်းကြီး၊ တစ်ဆယ်ထက်လည်း ငယ်တယ်ဆိုရင် တစ်နဲ့တစ်ဆယ်ကြား ဖြစ်ပါတယ်။ (Python မှာ  $1 < x < 10$  နဲ့ စစ်လိုလည်း ရတယ်)။ ယဉ်မောင်းလိုင်စင် ဖြဖို့ အသက် ဆယ့်ရှစ်နှစ်နဲ့ အထက် ဖြစ်ရမယ်၊ မှတ်ပုံတင်လည်း ရှိရမယ်ဆိုပါစို့။ ဘူလီယန် အပ်စ်ပရက်ရှင်နဲ့ အခုလို ဖော်ပြနိုင်တယ်

```
>>> is_18 = True
>>> has_nric = True
>>> is_18 and has_nric
True
>>> is_18 = False
>>> has_nric = True
>>> is_18 and has_nric
False
```

(အခု ဥပမာမှာ True/False တွေ ပုံသေထည့်ထားပေမဲ့ နောက်ပိုင်းမှာ ပရိုဂရမ် input ပေါ် မှတည် ပြီး တွက်ချက်လပ်ဆောင်တေတွေ တွေ့ရမှာပါ)။

ဘူလီယန်တန်ဖိုး/အပ်စ်ပရက်ရှင် နှစ်ခုအနက် အနည်းဆုံး တစ်ခု မှန်/မမှန် or နဲ့ စစ်ဆိုင်တယ်။ နှစ် ခုလုံး အမှားဖြစ်မှ False ရပါတယ်။ တစ်ခုမှန်တာနဲ့ အမှန်ထွက်ပါတယ်။

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

ကိန်းတစ်ခုဟာ တစ်နဲ့ တစ်ဆယ်ရဲ့ ပြင်ပမှာရှိလား အခုလို စစ်ဆိုင်တယ်

```
>>> x = 11
>>> x < 1 or x > 10
```

```

True
>>> y = -2
>>> y < 1 or y > 10
True
>>> z = 5
>>> z < 1 or z > 10
False

```

တစ်ထက်ငယ်ရင် (သို့) တစ်ဆယ်ထက်၏ခြားရင် တစ်နှုတ်ဆယ်ကြားမှာ မဟုတ်လိုပါ။ (Python မှာ တန်ဖိုးနှစ်ခုကြားမှာရှိလား သို့မဟုတ် တန်ဖိုးနှစ်ခု ပြင်ပမှာရှိလား စစ်လို့ရတဲ့နည်း တစ်ခုမကရှိပါတယ်။ အခုနည်းလမ်းက and နဲ့ or ကို အသံချွဲတဲ့ ဥပမာတချို့ကို ပြခြင်းသာဖြစ်တယ်။)

ကုမ္ပဏီ တစ်ခုက အလုပ်ခေါ်တဲ့အခါ ဘွဲ့ရဲ (သို့) ဒီပလိုမာနဲ့ လုပ်သက် (၂) နှစ်အထက် အနည်းဆုံး ရှိသူ ဖြစ်ရမယ် ဆိုပါမို့။ သတ်မှတ် အရည်အချင်း ပြည့်မီ/မမီ အခုလို ဖော်ပြနိုင်ပါတယ်

```

>>> is_graduate = False
>>> has_2yrs_exp = True
>>> has_diploma =True
>>> is_graduate or (has_diploma and has_2yrs_exp)
True

```

တန်ဖိုးနှစ်ခု ကြား (သို့) ပြင်ပမှာ ရှိ/မရှိ စစ်တဲ့အခါ အဲဒီတန်ဖိုးနှစ်ခု အပါအဝင်လား (inclusive)၊ ဒါမှုမဟုတ် မပါဝင်ဘူးလား (exclusive) ဂရိစိက်ဖို့ လိုပါတယ်။ တစ်နှုတ်ဆယ်ကြား (ငှါးတို့ အပါအဝင်) ဆိုရင် အခုလို စစ်ရမှာပါ

```

>>> y = 1
>>> y >= 1 and y <= 10
True

```

not အော်ပရိတ်တာကတော့ True/False တန်ဖိုးကို ပြောင်းပြန် ဖြစ်စေတယ်။

```

>>> not True
False
>>> not False
True

```

တစ်နှုတ်ဆယ် ပြင်ပ (ငှါးတို့မပါ) မှာ ရှိ/မရှိကို အခုလိုလည်း စစ်လိုရတယ်

```

not (x >= 1 and x <= 10)

```

‘ငါးရာအောက်’ လို့ ပြောတာဟာ ‘ငါးရာနှင့်အထက် မဟုတ်’ လို့ ပြောတာနဲ့ အဓိပါယ်တူတူပါပဲ။

```

>>> z = 499
>>> z < 500
True
>>> not (z >= 500)
True
>>> w = 500
>>> w < 500

```

```
| False  
|>>> not (w >= 500)  
| False
```

not အော်ပရိတ်တာ သုံးခြင်းအားဖြင့် အကြောင်းအရာတစ်ခုကိုပဲ အဓိပ္ပာယ်အားဖြင့် ညီမျှတဲ့ အိပ်စပရက်ရှင် အမျိုးမျိုးနဲ့ ဖော်ပြလိုရလာတာကို တော်းခိုင်တယ်။





# အခန်း ၆

## အော်ဂျက်များ

“အော်ဂျက် (object) ဆိတာဘာလ” ရှုထောင့် အမျိုးမျိုးကနေ ရှင်းပြနိုင်ပါတယ်။ အပြည့်စုံဆုံး၊ အမှန်ကန်ဆုံး ဥပမာ သို့မဟုတ် အဓိပ္ပာယ်ဖွင့်ဆိုချက် ဆိတာ မရှိပါဘူး။ သူနည်း သူ့ဟန်နဲ့ မှန်ကန်ကြတေပါပဲ။ ဒီအခြားမှာတော့ အော်ဂျက်ဆိတာ ဘာလဲ၊ ဘယ်လိမ့်မျိုးလဲ ခံစားလို့ရအောင်နဲ့ အခြေခံ အသုံးချကာတ်ရုံး လောက်ပဲ အဓိကထား လေ့လာကြမှာပါ။ လက်ရှိအခြေအနေနဲ့ သင့်တော်မဲ့ အဓိပ္ပာယ်ဖွင့်ဆိုချက် တချို့ကို လည်း ဖော်ပြပေးသွားမှာပါ။ သိထားသင့်တဲ့ ဘာသာရပ်ဆိုင်ရာ စကားလုံး အသုံးအနှစ်းတွေကိုလည်း မိတ်ဆက်ပါမယ်။

ဆော်စဲ အော်ဂျက်တွေဟာ အပြင်မှာ တကယ်ရှိတဲ့ အရာတွေရော တကယ်မရှိဘဲ စိတ်ကူးသက်သက်ဖြစ်တဲ့ ဒုဝေးအိုးပဲ ပရိုက်မူတဲ့မှာ ထင်ဟပ် ဖော်ပြတယ်။ ဥပမာ ကေသို့ဘဏ်အကောင့်၊  $\frac{7}{13}$  (အပိုင်းကဏ်း၊ တစ်ခု)၊ 1948-01-04 (မြန်မာပြည် လွှတ်လပ်ရေးရတဲ့နေ့)၊ စနီပိုင်တဲ့ အနီရောင် တို့ယို တာကား စတာတွေကို အော်ဂျက်တွေနဲ့ ဖော်ပြနိုင်တယ်။

အော်ဂျက်မှာလည်း တိုက်ပ်သဘောတရား ရှိတယ်။ တိုက်ပ်တူတဲ့ အော်ဂျက်အားလုံး အော်ဂျက်အားလုံး စည်းထားပဲ တူတယ်။ လုပ်ဆောင်လို့ရတဲ့ အော်ပရေးရှင်းတွေလည်း တူပါတယ်။ ဘဏ်အကောင့် အော်ဂျက်အားလုံးဟာ လက်ကျော်ငွေနဲ့ အကောင့်နံပါတ် ပါရှိပြီး ငွေသွေး၊ ငွေထုတ်၊ ငွေလွှဲ အော်ပရေးရှင်းတွေ လုပ်ဆောင်လို့ ပါမယ်။

အော်ဂျက်တွဲရဲ တိုက်ပဲနဲ့ နီးနီးစပ်စပ် ဆက်နှုတ်နေတာကတော့ ကလပ်စ် (class) သဘောတရားပါ။ ကလပ်စ်ကို တိုက်ပ်တူအော်ဂျက်တွေ ဖန်တီးဖို့အတွက် သတ်မှတ်ထားတဲ့ ပရိုက်မူများ အစုအဝေး လို့ ယေဘုယျ ပြောနိုင်ပါတယ်။ Account ကလပ်စ်၊ date ကလပ်စ်၊ Fraction ကလပ်စ် စသည်ဖြင့် အော်ဂျက် တိုက်ပ် တစ်မျိုးအတွက် ကလပ်စ်စ်ခု ရှိမှာပါ။ တိုက်ပ်တူ အော်ဂျက်တွေ အားလုံးမှာ ပါဝင်မဲ့ အချက်အလက်တွေ၊ အော်ပရေးရှင်းတွေနဲ့ အော်ဂျက် ဖန်တီးယူတဲ့ ဖန်ရှင်တွေကို ကလပ်စ် တစ်ခုနဲ့ သတ်မှတ်ရတာပါ။ ကလပ်စ်ကနေ အော်ဂျက်တွေ (တိုက်ပ် တူပါမယ်) ထုတ်ယူရတာ ဖြစ်တဲ့ အတွက် ကလပ်စ်ကို အော်ဂျက် စက်ရှုလိုလည်း ဆိုနိုင်ပါတယ်။ ပုံစံတူ အော်ဂျက်တွေ ထုတ်ပေးတာ မို့လို့ ကလပ်စ်ဆိုတာ အော်ဂျက်တည်ဆောက်တဲ့ blueprint သို့မဟုတ် template ပဲလို့ ယူဆတာ ဟာလည်း သဘာဝကျပ်တယ် ဆုံးရမှာပါ။

အော်ဂျက်တွေကို အက်ဘ်စရက်ရှင်း (abstraction) အနေနဲ့လည်း ရှုမြင်နိုင်တယ်။ ဘယ်လို ဖန်တီး တည်ဆောက်ထားလဲ မသိဘဲ အသုံးပြုလိုရတဲ့ အရာအားလုံးကို အက်ဘ်စရက်ရှင်းလို့ ဆိုနိုင်တယ်။ အပြင်မှာသုံးကြတဲ့ ကား၊ တို့စွာ ကွန်ပူးတာ စတာတွေဟာ အက်ဘ်စရက်ရှင်းတွေ ဖြစ်တယ်။ ဖန်ရှင် တွေဟာလည်း အက်ဘ်စရက်ရှင်းတွေပါပဲ။ အော်ဂျက်တွေကတော့ အော်ပရေးရှင်း တွဲဖက်ပေါင်းစပ်ထားတဲ့ အက်ဘ်စရက်ရှင်းတွေပါ။ အော်ဂျက်တစ်ခုနဲ့ တွဲဆက်ထားတဲ့ အော်ပရေးရှင်းတွေဟာ

အဲဒေသအော်ဘက်ရဲ ဒေတာတွေကို အသုံးပြုတယ်။ အဲဒေသအော်ဘက်ရဲ ဒေတာအပေါ် သက်ရောက်မှ ရှိနိုင်တယ်။ အခြားအော်ဘက်ရဲ ဒေတာကို မသုံးဘူး။ သက်ရောက်မှုလည်း မရှိစေဘူး။ အော့ဘ်ဘက် အတွင်းပိုင်း ဒေတာတွဲ ဖွဲ့စည်းထားပဲ့၊ တိုက်ပ်ကို မသိဘဲ အော့ဘ်ဘက်ကို အသုံးပြုလိုပါတယ်။ အော်ပရေးရှင်းတွေဟာ တကယ်ကတော့ အော့ဘ်ဘက်ဒေတာ အသုံးပြုတဲ့ ဖုန်းရှင်တွေပါပဲ။ ဒီဖုန်းရှင်တွေ ဘယ်လိုရေးထားလဲ ဒေတာကို ဘယ်ပုံဘယ်နည်း အသုံးပြုတာလဲ သိစေရာမလိုဘဲ အသုံးပြုလို ရပါတယ်။

ဖန်ရှင် အသင့်ရှိပြီးသား ဆုံးရင် အသုံးပြုလို ရသလို ကလပ်စ အသင့်ရှိပြီးသား ဆုံးရင် အော့သုတေသန တွေ ဖန်တီးအသုံးပြုနိုင်ပါတယ်။ ဖန်ရှင်ရေးရတာ ခက်ခဲနိုင်ပါတယ်။ ရှိပြီးသား ဖန်ရှင်သုံးတာကတော့ မ ခက်ပါဘူး။ ဒီသဘောပါပဲ။ ကလပ်စသတ်မှတ်ရတာ၊ ဒီဇိုင်းလုပ်ရတာ ရွှေပြထွေး ခက်ခဲနိုင်ပါတယ်။ ရှိပြီးသား ကလပ်စကနေ အော့သုတေသန ဖန်တီးအသုံးပြုရတာ မခက်ပါဘူး။ အသုံးပြုသူ လွယ်ကူအဆင်ပြေစေဖို့ တည်ဆောက်သူက အဓိကစဉ်းစား ဖျောရင်းရတာပါ။ သုံးစွဲသုံးအဆင့်ကနေ စတင်ပြီး တည်ဆောက်သူ ပ ရိုက်မှုပါ ဖြစ်လာအောင် တစ်ဆင့်ချင်း တက်လာမှုးဖို့ဟာ အဓိကပန်းတိုင်ပါ။ အော့သုတေသန မိတ်ဆက် ခဏ ရပ်၊ အခုပဲ လက်တွေ့စ်းသပ် ကြည့်လိုက်ရအောင် ...

## 6.0 date, time and datetime

```
>>> from datetime import *
>>> date(1948, 1, 4)
datetime.date(1948, 1, 4)
```

ဒုတိယလိုင်းက အော်ဂျက် ဖန်တီးတာပါ။ ဖန်ရှင်ခေါ်တာနဲ့ ပုံစံတူတာ တွေ့ရတယ်။ အော်ဂျက်ဖန်တီးဖို့ စပယ်ရှယ် ဖန်ရှင်တစ်ခု ခေါ်ထားတယ်လို့ ယူဆနိုင်ပါတယ် (နောက်ပိုင်း ကလပ်စ်အခန်းမှာ အသေးစိတ် လေ့လာရမှာပါ)။ ကလပ်စ်ကနေ အော်ဂျက် ဖန်တီးယူတာကို instantiation လိုခေါ်ပြီး ရရှိလာတဲ့ အော်ဂျက်ကို အဲဒီကလပ်စ်ရဲ့ instance လိုလည်း ခေါ်ပါတယ်။

```
>>> mmid = date(1948, 1, 4)
```

အော်ဂျက်ကို mmid ဖေရီရော့လွှာ ရည်ညွှန်း အသုံးပြုလို ရမှာဖြစ်တယ်။

```
| >>> mmid.year
```

1948

Dot notation ( . အမှတ်အသား ) နဲ့ အော့ဘ်ဂျက်ရဲ ခုနှစ်ကို ရယူထားတာပါ။ လနဲ့ ရက်ကိုလည်း အလားတုန်ညွှန်ပေးလမ်းနဲ့ ယူကြည့်စိုင်တယ်။

```
| >>> mmid.month
```

1

```
>>> mmid.day
```

အော်ဂျက်တစ်ခုမှာ ပါဝင်တဲ့ ဒေတာကို *attribute* လို့ ခေါ်တယ်။ *Attribute* တွေဟာ အော်ဂျက် ရဲ့ လက်ရှိအခြေအနေ (*state*) ကိုဖော်ပြတယ်။ စတိတ် ပြောင်းလဲနိုင်တဲ့ အော်ဂျက်တွေကို *mutable object* လို့ ခေါ်တယ်။ စတိတ် မပြောင်းလဲနိုင်တဲ့ အော်ဂျက်တွေကို *immutable object* လို့ ခေါ်တယ်။ *date* အော်ဂျက်တွေဟာ *immutable object* တွေပါ။ ဆိုလိုတာက *attribute* တွေဖြစ်တဲ့ *year*, *month*, *day* တန်ဖိုးတွေ မပြောင်းလဲနိုင်ပါဘူး။

ခုနှစ် လ၊ ရက် အတွက် *attribute* သုံးခဲ့ဟာ *date* အော်ဂျက် တစ်ခုစီတိုင်းအတွက် ကိုယ်ပိုင်ပါရှိမှုပါ။ ဆိုလိုတာက အောက်ပါ *usid* အော်ဂျက်ရဲ့ *attribute* တွေနဲ့ ခုနှစ် *mmid* ရဲ့ *attribute* တွေဟာ သီးခြားစီပါ။ နံမည်တူပေမဲ့ တစ်ခုနဲ့တစ်ခု မရောယူက်ဘူး။

```
>>> usid = date(1776, 7, 4)

>>> usid.year
1776
>>> usid.month
7
>>> usid.day
4
```

အဥုလို တန်ဖိုးပြန်ယူကြည့်ရင်လည်း ဖြစ်သင့်တဲ့အတိုင်း သက်ဆိုင်ရာ အော်ဂျက်ရဲ့ *attribute* တန်ဖိုး တွေပဲ ပြန်ရတာပေါ့။ လွှတ်လပ်ရေး ရခဲ့တဲ့နေ့က ဘာနေ့ဖြစ်မလဲ

```
>>> usid.isoweekday()
4
>>> mmid.isoweekday()
7
```

ဖန်ရှင်တစ်ခုတည်းကို မတူညီတဲ့ အော်ဂျက်နှစ်ခုအပေါ်မှာ အသုံးချတာ ဖြစ်တယ်။ ဒေါ်ထိုပဲ သုံးတယ်။ ပထမတစ်ခုက *usid* အော်ဂျက်၊ နောက်တစ်ခုက *mmid* အော်ဂျက်အပေါ်မှာ သုံးထားတာပါ။ အမေရိကန် လွှတ်လပ်ရေးရနဲ့တာ ကြောသာပတေးနေ့ ပြန်မာကတော့ တန်ခိုက်နေ့နေ့ပါ (တန်လ်က တစ်၊ တန်ခိုက်နေ့က ခုနှစ်ပါ)။ *isoweekday* ဖန်ရှင်ကို အော်ဂျက်တစ်ခုအပေါ် အသုံးပြုတဲ့အခါ ငင်းအော် ဂျက်နဲ့ သက်ဆိုင်တဲ့ ဒေတာနဲ့ ဖန်ရှင်က အလုပ်လုပ်သွားတာပါ။ ဒါကြောင့်လည်း *attribute* မတူတဲ့ အော်ဂျက်တွေအပေါ်မှာ အသုံးချတဲ့အခါ မတူညီတဲ့ ရလဒ်တွေ ထွေက်လာရတာပေါ့။ ‘ဒေတာနဲ့ အော်ပရေးရှင်း တွဲဖက်ထားတယ်’ ဆိုတာ ဒီသဘောတရားကို ဆိုလိုတာပါ။ အော်ဂျက်ဒေတာနဲ့ တွဲဖက်အလုပ်လုပ်တဲ့ ဖန်ရှင်တွေကို မက်သွဲ (method) လို့ ခေါ်တယ်။ နောက် မက်သွဲတစ်ခုက *isoformat* ပါ။ နောက်ကို စားသားအဖြစ် ‘yyyy-mm-dd’ ဖော့မတ်နဲ့ ပြန်ပေးတယ်။

```
>>> usid.isoformat()
'1776-07-04'
>>> mmid.isoformat()
'1948-01-04'
```

*date* တစ်ခု ဖန်တီးတဲ့အခါ ခုနှစ် လ၊ ရက် နေရာ မှုနှစ် အရေးကြီးပါတယ်။ *date(1948, 4, 1)* လို့ ရောမိရင် လေးလပိုင်း တစ်ရက်နဲ့ ဖြစ်သွားမှုပါ။ ဒါပေမဲ့ Python မှာ အုပ္ပန်တွေကို နံမည်နဲ့ တဲ့ ပြီး ထည့်ပေးလို့ရတယ်။

```
>>> mmid = date(day=4, year=1948, month=1)
```

ဒီနည်းနဲ့ ဆိုရင်တော့ year, month, day ကြိုက်သလို အစီအစဉ်နဲ့ ထည့်လိုရမှာပါ။  
replace မက်သဒ် ဘယ်လို အလုပ်လုပ်လဲ ကြည့်ရအောင်

```
>>> usid = date(1776, 7, 4)
>>> usid100 = usid.replace(year=1876)
```

နုဂ္ဗ ပေါ်ရက်ရဲ ခုနှစ်ကို 1876 နဲ့ အစားထိုးထားတဲ့ အော့သံဂျက် အသစ်တစ်ခု ပြန်ရပါတယ်။ နုဂ္ဗ ရက်စွဲက မပေါ်တော်သွားဘူး (date အော့သံဂျက် ဟာ immutable ဖြစ်တာ သတိပြုပါ)။

```
>>> usid
datetime.date(1776, 7, 4)
>>> usid100
datetime.date(1876, 7, 4)
```

ခုနှစ်၊ လ၊ ရက် သုံးခုလုံး အစားထိုးချင်ရင်

```
>>> dt1 = date(2000,2,21)
>>> dt2 = dt1.replace(2010,10,10)
>>> dt3 = dt1.replace(day=20,month=12,year=2020)
```

အားဂုဏ်နဲ့ မပါရင် ခုနှစ်၊ လ၊ ရက် အစဉ်အတိုင်း ဖြစ်ရပါမယ်။ ရလဒ်တွေ ကြည့်ရင်

```
>>> dt1
datetime.date(2000, 2, 21)
>>> dt2
datetime.date(2010, 10, 10)
>>> dt3
datetime.date(2020, 12, 20)
```

ခုနှစ်၊ လ၊ ရက် တစ်ခု ချုပ်ထားကြည့်ပါ

```
>>> dt4 = dt1.replace(2020)
>>> dt5 = dt1.replace(2030,11)
```

ချုပ်ထားခဲ့တော့ နုဂ္ဗအတိုင်းရှိပါမယ်

```
>>> dt4
datetime.date(2020, 2, 21)
>>> dt5
datetime.date(2030, 11, 21)
```

ရက်တစ်ခုတည်း အစားထိုးမယ်ဆိုရင် နံမည်နဲ့တဲ့ နည်းကပဲ အဆင်ပြပါမယ်

```
>>> dt6 = dt1.replace(day=28)
>>> dt6
datetime.date(2000, 2, 28)
```

နံမည်မပါရင် ခုနှစ်၊ လ၊ ရက် အစဉ်အတိုင်းဖြစ်တော်ကြောင့် ခုနှစ်ကို အစားထိုးမှာပါ

```
>>> dt7 = dt1.replace(28)
>>> dt7
datetime.date(28, 2, 21)
```

### time and datetime

နေရက်နဲ့ အချိန် တွဲရက်ကို datetime, ရက်စွဲမလိုဘဲ အချိန်ပဆိုရင် time သုံးပါတယ်

```
>>> t1 = time(10, 15, 20)
>>> t1.hour
10
>>> t1.minute
15
>>> t1.second
20
>>> mmid2 = datetime(1948,1,4,4,20)
>>> mmid3 = datetime(1948,1,4,4,20,0)
>>> mmid2.second
0
>>> mmid3.second
0
```

### timedelta

အချိန်ကာလနဲ့ ပါတ်သက်ပြီး မဖြစ်မနေ သိထားသင့်တဲ့ နောက်ထပ်ကလပ်စ် တစ်ခုကတော့ ကြောချိန် (duration) ကို ဖော်ပြတဲ့ timedelta ကလပ်စ်ပါ။

```
>>> duration = timedelta(
...     days=50,
...     seconds=27,
...     microseconds=10,
...     milliseconds=29000,
...     minutes=5,
...     hours=8,
...     weeks=2
... )
>>> duration
datetime.timedelta(days=64, seconds=29156, microseconds=10)
```

(... က အော်တို့ ထည့်ပေးသွားတာ။ ကိုယ်တိုင် ရိုက်ထည့်စရာမလိုဘူး။ တစ်လိုင်းချင်း Enter ခေါက် သွားရုံပဲ။ အပိတ်ဝိုက်ကွင်းမှာ စတိတ်မနဲ့ ဆုံးတယ်ဆိုတာ အင်တာပရက်တာက နားလည်တယ်။)

အော့ဘာ့ဂျက် အသုံးပြုသူအနေနဲ့ အချိန်ကာလ ကြောမြင့်ချိန်ကို days, weeks, hours ... စတာ တွေနဲ့ သတ်မှတ်လို့ရတယ်။ ငွေးတို့ကို ရက်၊ စက်နှင့် မိုက်ခရီးစက္ကန် ဖွဲ့ပြီး အော့ဘာ့ဂျက် အတွင်းပိုင်း days, seconds, microseconds attributes ဒေတာအနေနဲ့ သိမ်းမှာပါ။

```
>>> twoweeks_twomin = timedelta(weeks=1, days=7, minutes=2)
>>> twoweeks_twomin
datetime.timedelta(days=14, seconds=120)
```

ဖော်ပြခဲ့ပြီးတဲ့ အော့ဘ်ဂျက်တွေနဲ့ အပေါင်း၊ အနှစ် အော်ပရေးရှင်းတွေ လုပ်လိုပါတယ်။ ဥပမာ  
တရူး၊ လေ့လာကြည့်ပါ

```
>>> dt1 = datetime(2021, 2, 10, 23, 45, 43)
>>> dt2 = datetime(2022, 2, 10, 23, 44, 42)
>>> duration1 = dt2 - dt1
>>> duration1
datetime.timedelta(days=364, seconds=86339)
```

ဒီလိုစစ်ကြည့်ပါ

```
>>> dt3 = dt1 + duration1
>>> dt3
datetime.datetime(2022, 2, 10, 23, 44, 42)
>>> dt4 = dt2 - duration1
>>> dt4
datetime.datetime(2021, 2, 10, 23, 45, 43)
>>> dt1 == dt4
True
>>> dt2 == dt3
True
```

## ၆.J list

*List* ဆိတာ အိုက်တမ် item တွေ အတွဲလိုက် စုစည်းထားဖို့ အသုံးပြုတဲ့ စထရက်ချေတစ်မျိုး ဖြစ်တယ်။ Python မှာ list အော့ဘ်ဂျက်တွေကို item တွေ အတွဲလိုက် စုစည်းထားဖို့ သုံးတယ်။ ဘာအိုက်တမ်  
မှ မပါတဲ့ list အသစ်တစ်ခု လိုချင်ရင် ဒီလို

```
>>> odds = list()
ဒီ lsit ထဲမှာ ပါတွေပါလဲ
```

```
>>> odds
[]
```

လေးထောင့်ကွင်းနဲ့ list ကိုပြပေးတယ်။ လက်ရှိ list ထဲမှာ အိုက်တမ် မရှိသေးဘူး။ အိုက်တမ် တစ်  
ခုချင်း ထည့်ချင်ရင် append မက်သဒ်ရှိတယ်

```
>>> odds.append(1)
>>> odds.append(3)
>>> odds.append(5)
>>> odds.append(7)
```

```
>>> odds
[1, 3, 5, 7]
```

ပါဝင်တဲ့ အိုက်တမ်တစ်ခုစီကို ကော်မားပြီး ပြတယ်။ နောက်ထပ် နည်းလမ်းတစ်ခုနဲ့လည်း list ဖန်တီးလို့ ရတယ်။ လေးထောင့်ကွင်း သံးတဲ့နည်းပါ

```
>>> empty = []
>>> lst1 = [1,2,3,4,5,6]
>>> empty
[]
>>> lst1
[1, 2, 3, 4, 5, 6]
```

list ဟာ mutable ဖြစ်တယ်။ အိုက်တမ် နောက်တစ်ခု ထပ်ထည့်ကြည့်ပါ

```
>>> odds.append(9)
>>> odds
[1, 3, 5, 7, 9]
```

နှင့် အော့ဘ်ဂျက်မှာ အိုက်တမ်တစ်ခု ထပ်တိုးသွားတာ။ အော့ဘ်ဂျက်ရဲ့ စတိတ် ပြောင်းသွားတယ်။ အိုက်တမ်တစ်ခုကို ဖယ်ထုတ်ချင်ရင်

```
>>> odds.pop(0)
1
>>> odds
[3, 5, 7, 9]
```

list အိုက်တမ်တွေရဲ့ တည်နေရာကို index လို့ခေါ်တယ်။ သူညန့် စတယ်။ ဒုတိယက တစ်၊ တတိယက နစ် စသည်ဖြင့် ဖြစ်မယ်။ အခုဗ္ဗ odds မှာ အိုက်တမ် လေးခုရှိနေတယ်။ 7 ဖြူတ်ချင်ရင် index နံပါတ် 2 ကို pop လုပ်ရမှာ

```
>>> odds.pop(2)
7
>>> odds
[3, 5, 9]
```

ဖယ်လိုက်တဲ့ အိုက်တမ်တွေ နှင့်နေရာမှာ ပြန်ထည့်ချင်တယ်ဆိုပါစို့။ insert ရှိပါတယ်

```
>>> odds.insert(2, 7)
>>> odds
[3, 5, 7, 9]
>>> odds.insert(0,1)
>>> odds
[1, 3, 5, 7, 9]
```

အိုက်တမ် အစားထိုးတာ၊ index နဲ့ နေရာတစ်ခုက အိုက်တမ်ကို ပြန်ထုတ်ကြည့်တာကို လေးထောင့်ကွင်းနဲ့ ရေးနည်းလည်းရှိတယ်

```
>>> evens = [2,4,6,8,10,12]
```

```
>>> evens[0]
2
>>> evens[1]
4
```

`pop` နဲ့ မတူတာကို သတိပြုပါ။ `pop` က အိုက်တမ်းကို ဖယ်ထုတ်လိုက်တယ်။ စတိတ်ပြောင်းလဲစေတယ်။ အခုနှစ်းက မဖယ်ထုတ်ဘူး။ အိုက်တမ်းကိုပဲ ပြန်ပေးတာပါ။

```
>>> evens
[2, 4, 6, 8, 10, 12]
```

`replace` နဲ့ သဘောတရားတူတာကတော့

```
>>> evens[5] = 14
>>> evens
[2, 4, 6, 8, 10, 14]
```

နောက်ဆုံး နေရာ (index နံပါတ် 5) ကို 14 လဲထည့်လိုက်တာ။

ပေါ်ရော့လဲ နှစ်ခုက အော့သုတေသနများကို ရည်ညွှန်းနေရင် သတိပြုသင့်တဲ့ ထူးခြားချက် တွေ ရှိလာပါတယ်။

```
>>> fruits = ['mango', 'apple', 'strawberry', 'kiwi']
>>> myfav = fruits
```

ဒီလိုဆိုရင် အော့သုတေသနများကို `fruits` ရော `myfav` နဲ့ပါ သုံးလို့ရပါမယ်။ `fruits` နဲ့ အိုက်တမ်းအတွက် ထပ်ထည့်ကြည့်မယ်

```
>>> fruits.append('orange')
```

`myfav` ပါ လိုက်ပြောင်းတာကို တွေ့ရမှာပါ

```
>>> myfav
['mango', 'apple', 'strawberry', 'kiwi', 'orange']
```

Mutable အော့သုတေသနများ ဒီအချက်ကို သတိပြုဖို့ လိုပါတယ်။ ပေါ်ရော့လဲ တစ်ခုကနေ အော့သုတေသနများ စတိတ်ကို ပြောင်းလဲတဲ့အခါ အဲဒီအော့သုတေသနများကို ရည်ညွှန်းတဲ့ အခြား ပေါ်ရော့လဲ အားလုံးက လည်း အပြောင်းအလဲကို မြင်ရမှာ ဖြစ်တယ်။ Immutable ဆိုရင်တော့ စတိတ်မပြောင်းနိုင်တော့ကြောင့် ဒီလိုကိစ္စမျိုး စဉ်းစားစရာ မလိုဘူး။



# အခန်း ၃

## ကွန်ထရီးလ် စတိတ်မန်များ

ကွန်ထရီးလ် စတိတ်မန်တွေက ကားရဲ့လူ တွေ့တော့ တွေ့ခြုံပြီးသားပါ။ ဒါပေမဲ့ ကားရဲလ်ပရိုဂရမ်မင်း အတွက် လိုသလောက် အခြေခံကိုပဲ ကန့်သတ်ဖော်ပြခဲ့တာပါ။ ဒီအခန်းမှာ ပြည့်စုံအောင် အသေးစိတ် ဆက်လက် လေ့လာကြပါမယ်။ လက်တွေ့အသုံးချု ဥပမာတွေ၊ လေ့ကျင့်ခန်းတွေ ကရာတစိုက် စီစဉ်ပေးထားတယ်။ စတိတ်မန် အသစ်တချို့လည်း တွေ့ရမယ်။ စာအုပ်တွေမှာ ဖော်ပြတာ သိပ်မတွေရပေမဲ့ ဘိုင်နာ အများစုံ အခက်အခဲတွေ့ကြတဲ့ နေရာတွေ၊ တိတိကျကျ နားလည်ဖို့ လိုတဲ့ ပိုင့်တွေကိုလည်း အလေးပေးရှင်းပြထားတယ်။ အထူးခြားဆုံးကတော့ ရုပ်ပုံတွေဆုံးတာနဲ့ အန်နီမေးရှင်း အခြေခံကို Arcade ဂိမ်းလိုက်ဘရှိ အသုံးပြုပြီး စတင်မိတ်ဆက်ထားတာပဲ ဖြစ်တယ်။ စာသားတွေချည်းပဲထက် စိတ်လှပ်ရှားဖို့ ပိုကောင်းမယ်ထင်ပါတယ်။

### ၃.၁ if စတိတ်မန်

စားသောက်ဆိုင် တစ်ဆိုင်က ကျပ်ငွေ 50,000 နှဲ့ အထက် သုံးတဲ့ ကတ်စတမ်္မာတွေကို 10% လျှော့ပေးပြီး ပရီမိုးရှင်း လုပ်တယ် ဆိုပါစို့။ ကိုးဘုဒ်ကနေ ကျသင့်ငွေ ရိုက်ထည့်ပေးရမယ်။ ဒစ်စကောင့် ရမဲ့ ကတ်စတမ်္မာအတွက်ပဲ 'Get 10 % discount.' ပြပေးပြီး လာရောက် စားသုံးတဲ့အတွက် ကျေးဇူးတင်ကြောင်း 'Thanks for coming!' ကိုတော့ ကတ်စတမ်္မာတိုင်းကို ပြပေးချင်ပါတယ်။

```
amt = float(input("Enter amount: "))
if amt >= 50_000:
    print("Get 10% discount.")
print("Thanks for coming!")
```

amt > 50\_000 ဘူးလိုယန် အိပ်စ်ပရက်ရှင် true ဖြစ်မှုပဲ if ဘလောက်ကို လုပ်ဆောင်ပေးမှာပါ။ ဒစ်စကောင့် ဘယ်လောက်ရလဲရော ကျသင့်ငွေပါ ပြပေးမယ်ဆိုရင် ဒီလို

```
amt = float(input("Enter amount: "))
amt_to_pay = amt
if amt >= 50_000:
    discount = amt * 0.1
    print(f'Get 10% discount ({discount}).')
    amt_to_pay = amt - discount
```

```
print(f'Please pay: {amt_to_pay}')
print('Thanks for coming!')
```

Python ဘားရင်း 3.6 ကတဲ့ f-string (formatted string) ခေါ်တဲ့ string အသစ်တစ်မျိုး ပါလာပါတယ်။ String ရှေ့မှာ f နဲ့ စရင် f-string လို့ သတ်မှတ်တယ်။ Single/double quote ရှေ့မှာ f ထည့်ပေးရတာပါ။

```
>>> f"Two plus three is {2 + 3}"
'Two plus three is 5'
>>> f'Two plus three is {2 + 3}'
'Two plus three is 5'
```

F-string နဲ့ဆိုရင် ဖေရီရောဘဲလ် (သို့) အပိုစ်ပရက်ရှင် တွေကို တွန်ကွင်းထဲမှာ ထည့်ရေးလို့ရတယ်။ ငွေးတို့ကို f-string က တန်ဖိုးရာပြီး အတားထိုးပေးမှုပါ။ ဒါကြောင့် {2 + 3} က 5 ဖြစ်သွားတာပါ။ ရိုးရိုး string နဲ့ဆို အခါလို့

```
>>> 'Two plus three is ' + str(2 + 3)
'Two plus three is 5'
```

ရေးနေရမယ်။ F-string နဲ့ဆို ပိုအဆင်ပြေတယ်။ နမူနာတချို့ကို လေ့လာကြည့်ပါ

```
>>> x = 9
>>> y = 3
>>> f'2x + y = {2*x + y}'
'2x + y = 21'
>>> f'Times three hello {'hello' * 3}'
'Times three hello hellohellohello'
>>> f'Times three hello length is {len('hello' * 3)}'
'Times three hello length is 15'
```

ဒစ်စကောင့်ပေးပြီး ပရိုမိုးရှင်းလုပ်တဲ့အခါ သတ်မှတ်ပမာဏ မပြည့်သေးရင် ဘယ်လောက်ဖိုးထပ်သုံးတာနဲ့ ဒစ်စကောင့် ရဟန်ဖြစ်ကြောင့်ပြောပြီး ဆွဲဆောင်လေ့ရှိတယ်။ ဒစ်စကောင့်ရအောင် ဘယ်လောက်ထပ်သုံးရမလဲ ပရိုက်ရပ်က ပြေားချင်တယ် ဆိုပါစို့ if...else သုံးနိုင်ပါတယ်။

```
from math import *

amt = float(input("Enter amount: "))
amt_to_pay = amt
if amt >= 50_000:
    discount = amt * 0.1
    print(f"Get 10% discount({discount}).")
    amt_to_pay = amt - discount
else:
    amt_req = ceil(50_000 - amt)
    print(f"Spend just {amt_req} to get 10% discount!")

print("Please pay: " + str(amt_to_pay))
```

```

| print("Thanks for coming!")

amt >= 50_000 မှန်ရင် if ဘလောက် မှားရင် else ဘလောက် လုပ်ဆောင်မှာဖြစ်တယ်။
    if နဲ့ if...else ထော့ယျိုးခံကို ကြည့်ရင် အခုလိုရှိပါတယ်

if test:
    statement1
    statement2
    statement3
    ...etc.

if test:
    statement1a
    statement2a
    statement3a
    ...etc.

else:
    statement1b
    statement2b
    statement3b
    ...etc.

```

*test* ဟာ ဘူလီယန် အိပ်စ်ပရက်ရှင်း ဖြစ်ရပါမယ်။ (ကားရဲလ် ကွန်ဒီဂျင်တွေဟာ ဘူလီယန်တန်ဖိုး ပြန်ပေးတဲ့ predicate မက်သခ်တွေပါ။ predicate မက်သခ်တွေကို ဘူလီယန် အိပ်စ်ပရက်ရှင်းလို့ ယူဆနိုင်တယ်။)

အခုဆက်ကြည့်ကြမဲ့ if...elif...else ပုံစံကတော့ ရွှေ့ပိုင်းမှာ မတွေးဖူးသေးဘူး။ “Cascading if statement” လိုပေါ်တယ်။ အောက်ပါ ပေါ်မော်တော်က အနေ grading ထုတ်ပေးမယ် ဆိုပါစို့။

Score	Grade
90...100	A
80...89	B
70...79	C
60...69	D
(below 60) 0...59	F

### တေဘာလ် ၃.၁ Score and Grading

ကော်များ/သား နံမည်နဲ့ ရမှတ်ကို ထည့်ပေးရင် ပရီဂရမ်က အခုလို ပြပေးရပါမယ်။

```

Student name: Amy
Score: 95
Amy get grade A

```

ဒီပရီဂရမ် အတွက် cascading if သံဃားထားတဲ့ ကြည့်ပါ

```

stu_name = input("Student name: ")
score = int(input("Score: "))
grade = 'F'

if 90 <= score <= 100:
    grade = 'A'
elif 80 <= score <= 89:
    grade = 'B'
elif 70 <= score <= 79:
    grade = 'C'
elif 60 <= score <= 59:
    grade = 'D'
elif 0 <= score <= 59:
    grade = 'F'
else:
    print(f'You entered {score}. Score must be between 0 and 100.')

print(f'{stu_name} get grade {grade}')

```

အပေါ်ဆုံး if ပြီးတဲ့အခါ အောက်မှာ elif တွေ အတဲ့လိုက် တွေ့ရပါမယ်။ နောက်ဆုံးမှာ else အပိုင်းကို တွေ့ရတယ် (ဒီအပိုင်းက optional ပဲ၊ မပါလိုလဲရတယ်။ ခဏနေ ရှင်းပြပါမယ်)။ အလုပ်လုပ်ပုံက ဒီလို ... သက်ဆိုင်ရာ if (သို့) elif တွေရဲ့ ဘူလီယန် အိပ်စ်ပရက်ရှင် တစ်ခုချင်းကို အထက်အောက် အစဉ်အတိုင်း တန်ဖိုးရှာပါတယ်။ ပထမဆုံး True ဖြစ်တဲ့ အိပ်စ်ပရက်ရှင်နဲ့ သက်ဆိုင်တဲ့ ဘလောက်ကို လုပ်ဆောင်ပေးမှာ ဖြစ်တယ်။ အားလုံး False ဖြစ်ရင်တော့ else ဘလောက်ကို လုပ်ဆောင်တယ်။

နောက်ဆုံး else အပိုင်းက မပါလိုလည်းရတယ်။ အပေါ်မှာ တစ်ခုမှ True မဖြစ်တော့မှာ else ဘလောက်ကို လုပ်ဆောင်တယ်။ နောက်ဆုံးမှာ else အပိုင်းမပါဘူး၊ အပေါ်မှာလည်း ဘယ်တစ်ခုကမှ True မဖြစ်ဘူး ဆိုရင်တော့ လုပ်ဆောင်ပေးစရာ ဘလောက်လည်း မရှိဘူးပေါ့။ ဒီတော့ အားလုံး False ဖြစ်ခဲ့ရင် လုပ်ချင်တဲ့ကိစ္စ ရှိ/မရှိ အပေါ်မှာတည်ပြီး else အပိုင်း လို/မလို ဆုံးဖြတ်ရတယ်။

အခု grading ပရိုဂရမ်မှာ ရမှတ်ဟာ သူညန် တစ်ရာကြား ဖြစ်သင့်တယ်။ အကယ်၍ ထည့်ပေးတာမှားရင် မှားတယ်လို့ ပြပေးချင်တယ်။ ဥပမာ

```

Student name: Sandy
Score: 110
You entered 110. Score must be between 0 and 100.

```

သူညန် တစ်ရာအတွင်း မဟုတ်ရင် အပေါ်မှာ စစ်ထားတဲ့ ဘူလီယန် အိပ်စ်ပရက်ရှင်တွေ တစ်ခုမှ မမှန်နိုင်ဘူး။ ဒါကြောင့် else အပိုင်းနဲ့ မှားထည့်ထားတယ်လို့ ပြပေးလိုက်တယ်။

အခုပရိုဂရမ်မှာ သိပ်စိတ်တိုင်းကျစရာ မကောင်းတဲ့ ပြသုနာတစ်ခုတွေ့ရပါတယ်။ အောက်ပါအတိုင်း စမ်းကြည့်ရင် Sandy က grade F ရတယ်လို့ ပြနေပါတယ်

```

Student name: Sandy
Score: 110
You entered 110. Score must be between 0 and 100.
Sandy get grade F.

```

အမှတ်ထည့်ပေးတာ မှားနေရင် grade ကို မပြပေးသင့်ပါဘူး။ ဒီလို ပြင်ရေးလိုက်မယ် ဆိုရင်

```
stu_name = input("Student name: ")
score = int(input("Score: "))

if 0 <= score <= 100:
    grade = 'F'
    if 90 <= score <= 100:
        grade = 'A'
    elif 80 <= score <= 89:
        grade = 'B'
    elif 70 <= score <= 69:
        grade = 'C'
    elif 60 <= score <= 59:
        grade = 'D'
    elif 0 <= score <= 59:
        grade = 'F'

    print(f'{stu_name} get grade {grade}.')
else:
    print(f'You entered {score}. Score must be between 0 and 100.')
```

ရုတ် သုညနဲ့ တစ်ရာကြားဖြစ်မှ grading ထုတ်ပေးတဲ့ ကိစ္စလုပ်တယ် (if 0 <= score <= 100: နဲ့ စစ်ထားတာ)။ မဟုတ်ရင် ထည့်ထားတာ မှားနေတယ်ဆိုတာ else အပိုင်းက ပြပေးမှာပါ။ အပြင် if ဘလောက်ထဲက cascading if အဆုံးမှာ else မပါတွေ့တာ သတိပြုပါ။ ဘာကြောင့်ပါလဲ ...

Cascading if မှ ဘူလီယန် အပိုစ်ပရဂ်ရှင် တစ်ခုချင်းကို အထက်အောက် အစဉ်အတိုင်း တန်ဖိုးရှာတယ်, 'ပထမဆုံး True ဖြစ်တဲ့ ဘလောက တစ်ခုကိုပဲ လုပ်ဆောင်ပေးတယ်' ဆိတဲ့အချက်ကို နားလည် ဖို့ အရေးကြီးတယ်။ အောက်ကို ရောက်လာတာဟာ အပေါ်မှာ မှားခဲ့လိုပဲ။ တစ်ခုမှန်ပြီဆိုတာနဲ့ သက်ဆိုင် တဲ့ ဘလောက်ကို လုပ်ဆောင်ပြီး cascading if တစ်တဲ့လုံး ပြီးဆုံးသွားမှာ ဖြစ်တယ်။ အောက်ပိုင်းက elif (သို့) else တွေကို မရောက်လာတော့ဘူး။ ဒီအကြောင်းကြောင့် grading အတွက် အခဲ့လိုလည်း ရေးလိုရတယ်

```
stu_name = input("Student name: ")
score = int(input("Score: "))

if 0 <= score <= 100:
    grade = 'F'
    if score >= 90:
        grade = 'A'
    elif score >= 80:
        grade = 'B'
    elif score >= 70:
        grade = 'C'
    elif score >= 60:
        grade = 'D'
    else:
        grade = 'F'
```

```

    print(f'{stu_name} get grade {grade}.')
else:
    print(f'You entered {score}. Score must be between 0 and 100.')

```

ပထမ elif ကို ရောက်လာရင် ကိုဆယ်အောက် ဖြစ်မှာတော့ သေချာတယ် ( $score \geq 90$  မဟုတ်လို့ ဒါ ကို ရောက်လာတာ)၊ ဒါကြောင့် ရှစ်ဆယ့်အထက် ( $score \geq 80$ ) ဖြစ်လား စစ်ရင်ရပြီ။ နောက်တစ်ဆင့် ကို ရောက်လာရင် ရှစ်ဆယ်အောက် မို့လို့ သေချာတယ်၏  $score \geq 70$  ဖြစ်လားစစ်ရုံပဲ။ စသည်ဖြင့် အောက်အဆင့်တွေ အတွက်လည်း ထိန်ညွှေ့တူစွာ စဉ်းစားနိုင်တယ်။

Cascading if မသုံးဘဲ if...else တွေနဲ့လည်း ရေးလိုတော့ ရပါတယ်။ Nesting လုပ်တာ တွေ အရမ်းများပြီး ဖတ်ရမလွှာယူကူးတာ တွေရမှာပါ။ Grading ပရိုဂရမ်ကို cascading if မသုံးဘဲ ရေးထားတာပါ။

```

stu_name = input("Student name: ")
score = int(input("Score: "))
if 0 <= score <= 100:
    grade = 'F'
    if score >= 90:
        grade = 'A'
    else:
        if score >= 80:
            grade = 'B'
        else:
            if score >= 70:
                grade = 'C'
            else:
                if score >= 60:
                    grade = 'D'
                else:
                    grade = 'F'
print(f'{stu_name} get grade {grade}.')
else:
    print(f'You entered {score}. Score must be between 0 and 100.')

```

## ၇.၂ for Loop

Python for loop ဟာ အဆင့်မြင့် အက်ဘ်စရက်ရှင်း တစ်ခု ဖြစ်ပါတယ်။ စတိတ်မန်တစ်စုံကို သတ်မှတ်ထားတဲ့ အကြိမ်အရေအတွက် ပြည့်အောင် ထပ်ခါထပ်ခါ လုပ်ဆောင်ဖို့ လိုတဲ့အခါ for loop ကို အသုံးပြုတယ်။ Loop ကို စတင် လုပ်ဆောင်တဲ့အချင်းမှာ ဘယ်နှစ်ကြိမ် ပြန်ကျောမလဲ အတိအကျ ကြိုသိရင် definite loop လို့ သတ်မှတ်တယ်။ for loop ဟာ definite loop ဖြစ်ပါတယ်။ ‘အဆင့်မြင့် အက်ဘ်စရက်ရှင်း’ လို့ ပြောရတာက list, dictionary, set, range စတဲ့ စထရက်ချာ အမျိုးမျိုး နဲ့ အသုံးပြုလိုရတဲ့ အတွက်ကြောင့်ပါ။

for loop နဲ့ list ထဲက အိုက်တစ်တစ်ခုချင်း ထုတ်ယူအသုံးပြုနိုင်ပါတယ် ...

```

fruits = ['Orange', 'Kiwi', 'Banana', 'Papaya', 'Apple', 'Plum', 'Mango']
for item in fruits:

```

## ၆၆

```
    print(item)
```

Output:

```
Orange  
Kiwi  
Banana  
...
```

Loop တစ်ခေါက်ပြန်ကျော့တိုင်း item ဖော်ရောင်တဲ့မှာ list အိုက်တမ်တစ်ခုချင်း အစဉ်အတိုင်း ထည့်ပေးမှုပါ။ အိုက်တမ်တွေကို နံပါတ်စဉ်နဲ့ တဲ့ချင်ရင် enumerate လုပ်ပြီး အခုလို ထုတ်လိုက်ယ်

```
for idx, item in enumerate(fruits, start=1):  
    print(idx, item)
```

Output:

```
1 Orange  
2 Kiwi  
3 Banana  
...
```

နံပါတ်စဉ်ကို idx၊ အိုက်တမ်ကို item နဲ့ ယူသုံးထားတာပါ။ str တစ်ခုထဲက ကာရ်ကာတစ်လုံးချင်းလိုချင်ရင်လည်း ရတာပဲ

```
for ltr in 'This is a sentence written with full of emotion':  
    print(ltr)
```

Output:

```
T  
i  
s  
...
```

List နှစ်ခုရဲ့ cartesian product ၏ (ဖြစ်နိုင်တဲ့ အတွဲအားလုံး ရှာတာပါ)

```
colors = ['black', 'white']  
sizes = ['S', 'M', 'L']  
for color in colors:  
    for size in sizes:  
        print((color, size))
```

Output:

```
('black', 'S')  
('black', 'M')  
('black', 'L')  
('white', 'S')  
('white', 'M')
```

```
| ('white', 'L')
```

Dictionary နဲ့လည်း သုံးလို့ရတာပေါ့

```
scientist_birthdate = {'Newton': date(1643, 1, 4),
                       'Darwin': date(1809, 2, 12),
                       'Turing': date(1912, 6, 23)}
for sci, bdt in scientist_birthdate.items():
    print(sci, bdt)
```

Output:

```
Newton 1643-01-04
Darwin 1809-02-12
Turing 1912-06-23
```

ဖော်ပြခဲ့တဲ့ ဥပမာတွေကို ကြည့်ခြင်းအားဖြင့် Python for loop ဟာ list, dictionary, string စိတ် စထရက်ချာအမျိုးမျိုးနဲ့ အလုပ်လုပ်နိုင်တာ တွေ့ရမှာပါ။ တစ်ခါကျော့တိုင်း အိုက်တမ်တစ်ခုကို loop ဖော်ရော့လဲထဲမှာ ထည့်ပေးထားတယ်။ အိုက်တမ်တွေအားလုံး ပြီးတဲ့အခါ for loop ရပ်သွားမှာ ဖြစ်တယ်။ ပြန်ကျော့တဲ့ အကြိမ်အရေအတွက်ဟာ အိုက်တမ်အရေအတွက်ပဲ ဖြစ်တယ်။

### range ဖန်ရှင်နှင့် for loop

သူညောန တစ်ဆယ်ထိ အစဉ်အတိုင်း ရေတွက်ချင်ရင် နည်းလမ်းတစ်ခုက

```
for n in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    print(n)
```

ဒီလိုသာ ကဏ္ဍာတစ်လုံးချင်း ရိုက်ထည့်ရရှင် အဆင်မပြေပါဘူး။ ပိုကောင်းတဲ့ နည်းလမ်း ရှိရမှာပါ။ range ဖန်ရှင်ဟာ ဒီလိုနေရာမျိုးအတွက် အသင့်တော်ဆုံးပါပဲ။ range(0,11) က သူညောန တစ်ဆယ်ထိ အစဉ်အတိုင်း ထုတ်ပေးမဲ့ range အော့သုက်ကို ပြန်ပေးတယ်။ ဒီကဏ္ဍာတော်ကို တစ်ခါတည်း ကြိုးထုတ်ထားတာ မဟုတ်ပါဘူး။ လိုအပ်မှ တစ်ခုချင်း ထုတ်ပေးတာပါ။ (ဒီအတွက်ကြောင့် range(0, 11) နဲ့ range(0, 1\_000\_000) နှစ်ခုလုံး မမော်ရှိသုံးစွဲတာအရ သိပ်မကွာခြားပါဘူး။)

```
for i in range(0, 11):
    print(i)
```

2, 4, 6, ..., 12 စုံကိန်းတွေ လိုချင်ရင် range(2, 13, 2)၊ 5, 8, 11, ..., 98 လိုချင်ရင် range(5, 99, 3) စသည်ဖြင့် သုံးခုထည်ပြီး ဖန်ရှင်ခေါ်ခြုံမယ်။ အစ အဆုံးနဲ့ နောက်ဆုံးတစ်ခုကတော့ ကိုန်းတန်းမှာပါတဲ့ ကပ်လျက်ကိန်းနှစ်ခုခဲ့ ကွာခြားချက်ပါ။

```
for i in range(2, 13, 2):
    print(i)
```

```
for i in range(5, 99, 3):
    print(i)
```

အစေကဏ္ဍာ မသတ်မှတ်ပေးရင် သူညေလို့ ယူဆတယ်။ ဒါကြောင့် သူညောန တစ်ဆယ်ထိကို range(11)

နဲ့ ယူလိုရတယ်။ ကွားချက်က အနှစ်ကိန်း ဖြစ်လိုရတယ်

```
for i in range(10, 0, -1):
    print(i)

for i in range(0, -11, -2):
    print(i)
```

မှတ်ချက်။ ။ ကွားချက် တစ်မဟုတ်ရင် အစ အဆုံး ကွားချက် သုံးခုလုံး လိပါမယ်။

### for loop အသုံးချုပ်များ

စတိတ်မန်တစ်စုံကို သတ်မှတ်ထားတဲ့ အကြိမ်အရေအတွက် ပြည့်အောင် ပြန်ကျော့စိုး for loop ကို အသုံးပြုနိုင်ပါတယ်။ ကိုးဘုဒ်ကနေ ထည့်ပေးတဲ့ ကဏ္ဍး ဆယ်လုံးကို ပေါင်းမယ်ဆိုပါစိုး။

```
tot = 0
for i in range(10):
    val = float(input("?" ))
    tot += val

print(f"Total: {tot}")
```

ပရိုဂရမ် run တဲ့ အချိန်ကျတော့မှ အကြိမ်အရေအတွက် သတ်မှတ်လိုလည်း ရတယ်။ ဥပမာ

```
cnt = input("How many numbers you want to add? ")
tot = 0
for i in range(cnt):
    val = float(input("?" ))
    tot += val

print(f"Total: {tot}")
```

Loop တစ်ကျော့ပြီး တစ်ကျော့ ဖေရီရောဘဲလ် တန်ဖိုးတွေ ဘယ်လောက်ဖြစ်နေလဲ အခုလို လိုက် ကြည့်နိုင်ပါတယ်။

```
cnt = input("How many numbers you want to add? ") # 4 ထည့်တယ် ယူဆပါ
tot = 0

1st iter:
i = 0
val = float(input("?" ))      # 2 ထည့်တယ် ယူဆပါ
tot += val                   # 2 (လက်ရှိ tot တန်ဖိုး)

2nd iter:
i = 1
val = float(input("?" ))      # 3 ထည့်တယ် ယူဆပါ
tot += val                   # 5 (လက်ရှိ tot တန်ဖိုး)
```

**3<sup>rd</sup> iter:**

```
i = 2
val = float(input(" ? "))
tot += val
```

# 4 ထည့်တယ် ယူဆပါ  
# 9 (လက်ရှိ tot တန်ဖိုး)

**4<sup>th</sup> iter:**

```
i = 3
val = float(input(" ? "))
tot += val
```

# 11 ထည့်တယ် ယူဆပါ  
# 20 (လက်ရှိ tot တန်ဖိုး)

print(f"Total: {tot}")

# 20 ထုတ်ပေးမှာပါ

အောက်ပါ nested list ထဲမှ list တစ်ခုစီ ပေါင်းလဒ်နဲ့ list အားလုံး စုစုပေါင်း (grand total) ထုတ်ပေးပါမယ်။

```
rows = [[1, 3, 5, 2],
        [2, 9, 3, 7],
        [4, 4, 8, 3],
        [6, 2, 7, 9]]
```

```
# File: sum_of_rows.py
grand_tot = 0
for row in rows:
    row_tot = 0
    for val in row:
        row_tot += val
    print('Row total: ' + str(row_tot))
    grand_tot += row_total

print('Grand total: ' + str(grand_tot))
```

Nested `for` loop သုံးထားတယ်။ ပေါင်းလဒ်ကို ထည့်ထားဖို့ ပေရီရောဘဲလ် နှစ်ခု သုံးထားတာ သတိထား ကြည့်ပါ။ ဒီနေရာမှာ ပေရီရောဘဲလ် စကုပ် (scope) သဘောတရားကို နားလည့်ဖို့ လိုအပ်လာပါတယ်။ ပေရီရောဘဲလ်တစ်ခုကို ဘယ်နေရာကနေ သုံးလို့ရလဲဆိုတာဟာ ငြင်းပေရီရောဘဲလ်ရဲ့ စကုပ်နဲ့ သက်ဆိုင်ပါတယ်။ `grand_tot` ဟာ top level ပေရီရောဘဲလ် ဖြစ်တယ်။ ငြင်းကို ကြော်တဲ့ နေရာကစပြီး အောက်ပိုင်းတလျောက်လုံး သုံးလို့ရတယ်။ `grand_tot` ရဲ့ စကုပ်ဟာ ငြင်းကို ကြော်ထားတဲ့ ဖိုင်အဆုံးထိ ဖြစ်တယ်။ `rot_tot` ကတော့ block level ပေရီရောဘဲလ်ပါ။ Block level ပေရီရောဘဲလ်ကိုတော့ ငြင်းကို ကြော်ထားတဲ့ ဘလောက်အတွင်းမှာပဲ သုံးလို့ရပါမယ်။ Block level ပေရီရောဘဲလ်တစ်ခုရဲ့ စကုပ်ဟာ ငြင်းကို ကြော်ထားတဲ့ ဘလောက် အဆုံးထိ ဖြစ်တယ်။

`row` တစ်ခုချင်း ပေါင်းလဒ်ကို `grand_tot` မှ ပေါင်းထည့်ပေးဖို့ လိုတယ်။ အောက်ဆုံးမှာလည်း `grand_tot` ကို `print` ထုတ်ပေးရမယ်။ အကယ်၍ block level မှာ ထားလိုက်ရင် အောက်ဆုံးမှာ သုံးလို့ရမှာ မဟုတ်တော့ဘူး။ `row` တစ်ခု ပေါင်းပြီးရင် `rot_tot` ကို ထုတ်ပေးဖို့ လိုတယ်။ ဒါကြောင့် ငြင်းကို အပြင်ဘလောက်မှာ ကြော်ရမယ်။ အတွင်းဘလောက်မှာဆိုရင် အပြင်ကနေ သုံးလို့ရမှာ မဟုတ်တော့ဘူး။ (အတွင်း `for` loop ဟာ အပြင် `for` loop ရဲ့ ဘလောက် အတွင်းမှာ ပါဝင်တဲ့အတွက် `rot_tot` ကို သုံးလို့ရပါတယ်)။

Loop တစ်ကျော်ပြီး တစ်ကျော် မေရီရေဘဲလ် တန်ဖိုးတွေ ပြောင်းလဲသွားတာကို ပြထားတယ်။ တစ်ဆင့်ချင်း ဂရုစိုက်ပြီး လိုက်ကြည့်ပါ။ (အကြိမ်အရေအတွက် သိပ်မများအောင် အိုက်တမ် နည်းတဲ့ list နဲ့ ဥပမာ ပြထားတာပါ)။

```
# ဒါ list ဆဲ ကတ်နှုန်းတွေ ပေါင်းမှာပါ
rows = [[1, 3, 5],
        [2, 4, 6]]
```

```
grand_tot = 0

# အပြင်း for loop
1st iter:
row = [1, 3, 5]
row_tot = 0

# အပွဲ့ဌး for loop
1st iter:
val = 1
row_tot += val      # 1

2nd iter:
val = 3
row_tot += val      # 4

3rd iter:
val = 5
row_tot += val      # 9

print('Row total: ' + str(row_tot))
grand_tot += row_total # 9
# အပြင်ဘေးလေက် တစ်ကျော်ပြီး row_tot စောင် အဆုံး
```

```
# အပြင်း for loop
2nd iter:
row = [2, 4, 6]
row_tot = 0 # မေရီရေဘဲလ် တစ်ခါတယ်ကြော်တယ်

# အပွဲ့ဌး for loop
1st iter:
val = 2
row_tot += val      # 2

2nd iter:
val = 4
row_tot += val      # 6
```

3<sup>rd</sup> iter:

```
val = 6
row_tot += val           # 12
```

```
print('Row total: ' + str(row_tot))
grand_tot += row_total  # 21
# အပြင်ဘလောက် နောက်တစ်ကျွေးမီး row_tot စက်ပါ အဆုံး
```

```
print('Grand total: ' + str(grand_tot))
```

ခရစ်စမတ် သစ်ပင်လေး ဆွဲကြည့်ရအောင်။ တစ်တန်းမှာ စပေါ်ဘယ်နှစ်ခုပါလဲ ကြည့်ရလွယ်အောင် မြတ်လေးတွေနဲ့ ပြထားတယ်။

```
# File: christmas_tree.py
LEAF_ROWS = 8
TRUNK_ROWS = 3
# the width of the trunk
TRUNK_SZ = 3
# formula: LEAF_ROWS - 2
SPC_FOR_TRUNK = 6

for r in range(LEAF_ROWS):
    for i in range(LEAF_ROWS - 1 - r):
        print(' ', end=' ')
    for i in range(r * 2 + 1):
        print('*', end=' ')
    print()

for r in range(TRUNK_ROWS):
    for i in range(SPC_FOR_TRUNK):
        print(' ', end=' ')
    for i in range(TRUNK_SZ):
        print('*', end=' ')
    print()
```

```
******
***** ***
***** *****
***** ***** *
***** *****
***** *****
***** *****
***** *****
***** ***
```

အပေါ်ပိုင်း ဤကိုမှာ အတန်း ရှစ်ခု (LEAF\_ROWS)၊ ပင်စည်မှာ သုံးခု (TRUNK\_ROWS) သတ်မှတ်ထားတယ်။ ပင်စည် အကျယ် (TRUNK\_SZ) ကိုလည်း \* သုံးခု ထားတယ်။ အပေါ်ဆုံးကို row နံပါတ်သူညာ ဒုတိယကို တစ် စသည်ဖြင့် ယူဆပါမယ်။ row နံပါတ်စဉ်ကို ၁ ပေရဲ့ရေဘဲလဲက ဖော်ပြတယ်။ ဤကိုပူမှာ စပေါ်အရေအတွက်နဲ့ row နံပါတ်စဉ်ဟာ (LEAF\_ROWS - 1 - ၁) ဖော်မြှုလာနဲ့ ဆက်စပ်နေတယ်။ \* အရေအတွက်ကတော့ နှစ်ခုစီတိုးသွားပြီး (၁ \* ၂ + ၁) ဖြစ်တယ်။ ပင်စည်ပိုင်း စပေါ်အရေအတွက်ကတော့ တစ်တန်း ခြောက်ခုပါ (LEAF\_ROWS - 2) ။

## ၄.၃ Python Arcade Library

Python Arcade (ဝက်ဘ်ဆိုက် <https://api.arcade.academy>) ဟာ အရည်အသွေးကောင်းတဲ့ ထိပ်ဆဲ့ Python ဂိမ်းလိုက်ဘရီတွေထဲက တစ်ခုဖြစ်တယ်။ Arcade အပြင် အသုံးများတဲ့ အခြားတစ်ခုက pygame (ဝက်ဘ်ဆိုက် <https://www.pygame.org>) ပါ။ ဒီစာအပ်မှာ Arcade ကို သုံးပါမယ်။ Arcade ပိုကောင်းတယ်လို့ မဆိုလိုပါဘူး။ ဘိုင်နာတွေအတွက် ပို့သင့်တော်မယ် ယူဆတဲ့အတွက် အသုံးပြုတာပါ။ အောက်ပါအတိုင်း အင်စတောလ်လုပ်နိုင်ပါတယ်

```
| pip install arcade
```

Arcade နဲ့ ပုံဆွဲဖော်အတွက် ကရပ်ဖစ် ဝင်းဒီးတစ်ခုကို အောက်ပါအတိုင်း ယူရပါတယ်။ Run လိုက်ရင် ပုံ (၇.၁) မှာ တွေ့ရတဲ့ နောက်ခံရောင်နဲ့ ဝင်းဒီးအလွတ် တစ်ခု တက်လာမှာပါ။

```
# File: arcade_starter.py
import arcade

arcade.open_window(300, 200, "Arcade Starter")

# Set the background color
arcade.set_background_color(arcade.color.PINK_PEARL)

# Get ready to draw
arcade.start_render()

# Finish drawing
arcade.finish_render()

# Keep the window up until someone closes it.
arcade.run()
```

အပေါ်ဆုံးမှာ arcade လိုက်ဘရီ အင်ပို့လုပ်ထားတာပါ။ ရော်ပို့မှာသုံးတဲ့ from lib import \* ပုံစံ နဲ့ ကွာခြားတာက အခုနည်းနဲ့ အင်ပို့လုပ်ထားရင် လိုက်ဘရီမှာ ပါတဲ့ အစိတ်အပိုင်းတွေကို ဒေါ်ထဲအမှတ်အသားနဲ့ အသုံးပြုရပါမယ်။ ဥပမာ open\_window ဖန်ရှင်ကို

```
|arcade.open_window(arguments)
```

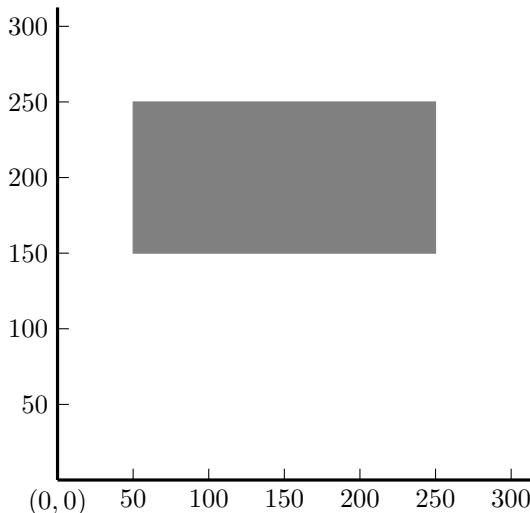
လိုက်ဘရီ နံမည်နောက်မှာ (.) အမှတ်အသား ခံပြီး ခေါ်ရှုဗာပါ။ ဒီဖန်ရှင်မှာ လိုချင်တဲ့ဝင်းဒီး အကျယ်အမြင်၊ ခေါင်းစည်းစာသား ထည့်ပေးထားတယ်။



ပုံ ၃.၁

### ကိုဖြေစီနိုင်ခန်းနှင့် ဝင်းဒီးပေါ်တွင် တည်နေရာ သတ်မှတ်ခြင်း

ဝင်းဒီး အောက် ဘယ်ဘက်ထောင့်စွန်းကို origin အမှတ် ( $x = 0, y = 0$ ) အဖြစ် သတ်မှတ်ပါ တယ်။ ဉာဘက်သွားရင်  $x$  တန်ဖိုး များလာမယ်။ အပေါ်ဘက်တက်ရင်  $y$  တန်ဖိုး တိုးလာမှာပါ။ အခြား ကွန်ပျူးတာ ဂရပ်ဖော်စနစ်တွေမှာလိုပဲ  $x, y$  တန်ဖိုးကို ယူနစ်အားဖြင့် pixel နဲ့ ဖော်ပြတယ်။ ပုံ (၇.J) မှာ rectangle ဘယ်ဘက်အောက် ထောင့်စွန်း ( $x, y$ ) တန်ဖိုး (50, 150), ဉာဘက် အပေါ်ထောင့်ဟာ (250, 250) ဖြစ်တယ်။



ပုံ ၃.၂

အောက်ပါ ဖန်ရှင်ကတွေ့ ဝင်းဒီးရဲ့ နောက်ခံရောင် သတ်မှတ်တာပါ။ လိုက်ဘရီရဲ့ color မော်ချိုး (module) မှာ အရောင်တန်ဖိုးတွေ အဆင်သင့် သတ်မှတ်ပေးထားတယ်။ (မော်ချိုးဆိုတာ လိုက်ဘရီရဲ့ အစိတ်အပိုင်းတစ်ခုလို့ အကြမ်းဖျဉ်း ယူဆနိုင်တယ်)။

```
arcade.set_background_color(arcade.color.PINK_PEARL)
```

အခုလို အင်ပိုလုပ်ထားရင် အရောင်တွေ သုံးရတာ ပိုအဆင်ပြေတယ်

```
import arcade
from arcade.color import *
```

```

...
arcade.set_background_color(PINK_PEARL)

```

PINK\_PEARL, RED စသည်ဖြင့် အရောင်နံမည် တမ်းရေးလိုဂုဏ်။ ရှေ့မှာ arcade.color. ထည့်စိုး  
မလိုတော့ဘူး။

ပုံဆွဲဖို့ အဆင်သင့်ဖြစ်အောင် start\_render ခေါ်ပေးရမယ်။ ဆွဲပြီးရင်လည်း finish\_render  
ခေါ်ဖို့လိုတယ်။ ပုံဆွဲတဲ့ ကိစ္စကို ငင်းတိနှစ်ခုကြားမှာ လုပ်ရမှာပါ။

```

arcade.start_render()
# call drawing functions here
...
...
arcade.finish_render()

arcade.run()

```

ဝင်းဒီးကို မပိတ်မချင်း ပေါ်နေအောင် run ဖန်ရှင် ခေါ်ပေးရတာပါ။ မခေါ်ထားဘဲ ပရိုဂရမ်ကို run ရင်  
ဝင်းဒီးပွင့်လာပြီး ဖုတ်ခနဲ့ ပြန်ပိတ်သွားမှာပါ။ မကျန်ခဲ့ဖို့ သတိပြုရပါမယ်။

Arcade မှာ ပါတဲ့ အခြေခံ ပုံဆွဲဖို့ရှင် တရာ့ပို့ကို ဆက်ကြည့်ရအောင်။ ထောင့်မှန်စတုဂံဆွဲတဲ့ ဖန်  
ရှင်တွေထဲက နှစ်ခု သုံးပြထားတယ်။ နှစ်ခုလုံးက ထောင့်မှန်စတုဂံရဲ့ ဘယ်ဘက်အောက် ထောင့်စွန်းနဲ့  
တည်နေရာကို သတ်မှတ်ပြီး အရွယ်အစားကို အကျယ်၊ အမြင့်နဲ့ သတ်မှတ်ပေးရတာပါ။ draw\_xywh\_rect  
angle\_filled က အတွင်းပိုင်း အရောင်နဲ့ ဆွဲပေးတယ်။ အနားတွေကိုပဲ ဆွဲချင်ရင် draw\_xywh\_rect  
angle\_outline ဖန်ရှင်သုံးရပါမယ်။

```

# File: arcade_drawing.py
import arcade
from arcade.color import *

arcade.open_window(300, 200, "Drawing Example")

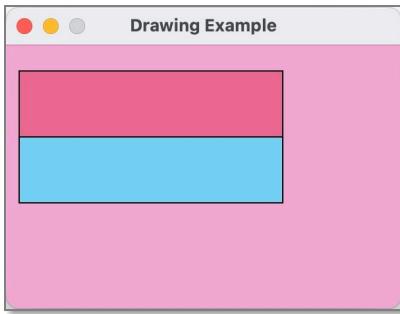
# Set the background color
arcade.set_background_color(PINK_PEARL)

# Get ready to draw
arcade.start_render()

# Lower rectangle (blue)
# Lower left corner (x,y) = (10,80)
# width = 200, height = 50
arcade.draw_xywh_rectangle_filled(10, 80, 200, 50, BABY_BLUE)
arcade.draw_xywh_rectangle_outline(10, 80, 200, 50, BLACK)

# Upper rectangle (pink)
# Lower left corner (x,y) = (10,130)
arcade.draw_xywh_rectangle_filled(10, 130, 200, 50, PALE_VIOLET_RED)
arcade.draw_xywh_rectangle_outline(10, 130, 200, 50, BLACK)

```



ပုံ ၃၃

```
# Finish drawing
arcade.finish_render()

# Keep the window up until someone closes it.
arcade.run()
```

အပေါ် ထောင့်မှန်စတုဂံပုံကို ဒီနှစ်ခုနဲ့

```
arcade.draw_xywh_rectangle_filled(10, 130, 200, 50, PALE_VIOLET_RED)
arcade.draw_xywh_rectangle_outline(10, 130, 200, 50, BLACK)
```

ဆွဲထားတာပါ 〔ပုံ (၇.၃)〕။ ပါရျမီတာတွေက  $x, y, width, height, color$  အစဉ်အတိုင်းပဲ။ အောက်က ထောင့်မှန်စတုဂံကို အခုလို

```
arcade.draw_xywh_rectangle_filled(10, 80, 200, 50, BABY_BLUE)
arcade.draw_xywh_rectangle_outline(10, 80, 200, 50, BLACK)
```

ဆွဲထားတာပါ။ ထောင့်မှန်စတုဂံ နှစ်ခုလုံး အကျယ် (၂၀၀)၊ အမြင့် (၅၀) pixels ရှိတယ်။

**ကျားကွက်ခု (သို့) စစ်တရာင်ခု ဆွဲခြင်း**

ကျားကွက် (သို့) စစ်တရာင်ခု ပုံဖော်တဲ့ checkerboard ဥပမာကို အောက်မှာလေလာကြည့်ပါ။ ပုံ (၇.၄) က ဒီပုံရှိရမ်းနဲ့ ဆွဲထားတာပါ။

```
# File: arcade_checkerboard.py
import arcade
from arcade.color import *

WIN_WIDTH = 600
WIN_HEIGHT = 420
arcade.open_window(WIN_WIDTH, WIN_HEIGHT, "Arcade Checkerboard")

arcade.set_background_color(WHITE_SMOKE)

arcade.start_render()
```

```

# width/height of the whole board
BOARD_SIZE = 400
COLS = 8
ROWS = 8
# size of each square of the board
SQ_SIZE = BOARD_SIZE // ROWS
# x of left side of the board
X_LFT = (WIN_WIDTH - BOARD_SIZE) // 2
# y of the bottom side of the bord
Y_BTM = (WIN_HEIGHT - BOARD_SIZE) // 2

y = Y_BTM
for i in range(ROWS):
    # start x from the left again
    x = X_LFT
    for j in range(COLS):
        # if sum of row and col numbers is even
        if (i + j) % 2 == 0:
            arcade.draw_xywh_rectangle_filled(x, y, SQ_SIZE, SQ_SIZE,
                                              WOOD_BROWN)
        else:
            arcade.draw_xywh_rectangle_filled(x, y, SQ_SIZE, SQ_SIZE,
                                              BLACK)
        # move to right
        x += SQ_SIZE

    # update y after each row
    y += SQ_SIZE

arcade.finish_render()

arcade.run()

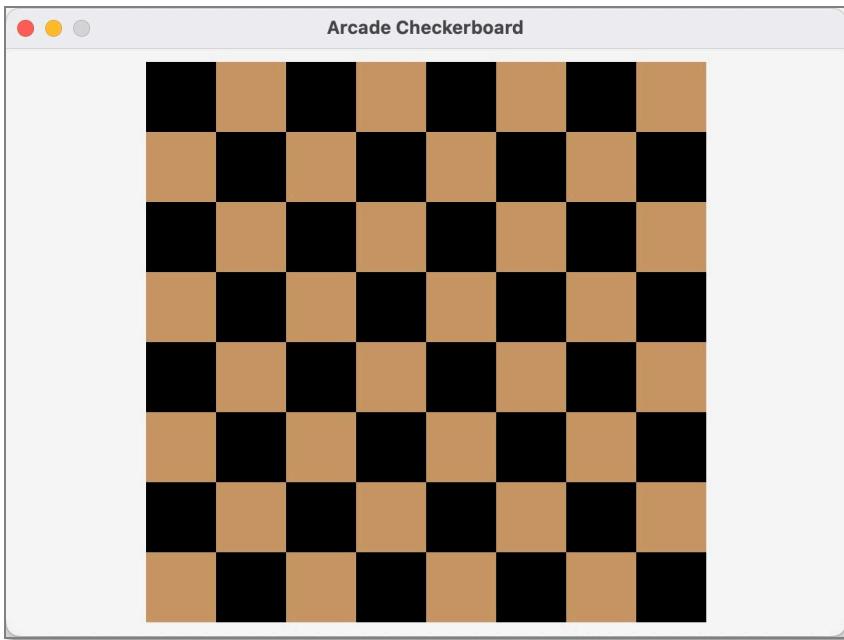
```

ပထမပိုင်းမှာ လိုအပ်တဲ့ constants တွေ သတ်မှတ်ထားတယ်။ ကျားကွဲက်တစ်ခုလုံး အရွယ်အစား ကို (၄၀၀) pixels ထားတယ်။ စတုရန်းပုံဆိုတော့ အကျယ်/အမြင် နှစ်ခုလုံး (၄၀၀)။ X\_LFT က row တစ်ခုချင်းရဲ့ စမ်တ် x တန်ဖိုး၊ Y\_BTM ကတော့ အောက်ဆုံး row y စမ်တ် တန်ဖိုး ဖြစ်ပါတယ်။

အတွင်း for loop က တစ်တန်းမှုပါတဲ့ ကျားကွဲက် (၈) ခုကို ဘယ်ကနေနော့ တစ်ကွဲက်ချင်း ဆဲ သွားမှာဖြစ်ပြီး အပြင် for loop ကတော့ row တစ်ခုပြီး တစ်ခု အောက်ဆုံးကစပြီး ပုံဖော်ပေးမှုပါ။ Row တစ်တန်းမှာ ပါဝင်တဲ့ အကွဲက်တွေအားလုံး y တန်ဖိုး တူတယ်။ Row တစ်ခုပြီး နောက်တစ်ခု ကူးတဲ့အခါ လက်ရှိ row ရဲ့ y တန်ဖိုးကို ကျားကွဲက်တစ်ကွဲက်စာ ပေါင်းပေးရပါမယ်

y += SQ\_SIZE

Row တစ်ခုပြီးမှ y တန်ဖိုးကို ပေါင်းပေးရမှာပါ။ ဒါကြောင့် ဒီစတိတ်မန်က အတွင်း for loop အပြီးမှာ ဖြစ်ရပါမယ် (Indentation ဂရိစိက်ကြည့်ပါ)။ Row တစ်ခုအတွက် ကျားကွဲက်တစ်ကွဲက်ချင်းကို ဘယ်



ပုံ ၃၄

ဘက်ကနေစပီး ဆွဲမှာပါ။ တစ်ကွဲက်ပြီး နောက်တစ်ကွဲက် ကူးတဲ့အခါ လက်ရှိ  $\times$  တန်ဖိုးကို ကျားကွဲက် တစ်ကွဲက်စာ ပေါင်းပေးရပါမယ်။ ဒီအတွက်

```
| x += SQ_SIZE
```

ကို အတွင်း for loop တစ်ကျော်ပြီးတိုင်း လုပ်ဖို့လိုတယ်။ Row တစ်ခု ပြီးသားပြီဆိုရင် နောက် row တစ်ခုက ဘယ်ဘက်ကနေ ပြန်စရူမှာပါ။ ဒီတော့ အတွင်း for loop မစခင်မှာ

```
| x = X_LFT
```

ပြန် လုပ်ပေးရပါမယ်။

ကျားကွဲက်ပတ်တန် တစ်လျဉ်းစီ အရောင်ခြယ်ဖို့ ဘယ်လိုလုပ်ရမလဲ။ တစ်ကွဲက်ချင်းရဲ့ row နံပါတ် နဲ့ column နံပါတ်ကို ပေါင်းကြည့်ပါ။ ပေါင်းလင် စုံကဏ္ဍး ဖြစ်တဲ့ အကွဲက်တွေ အရောင်တူတယ်။ မ ကဏ္ဍးဖြစ်တဲ့ အကွဲက်အားလုံး အရောင်တူတယ်။ အခုလို

```
if (i + j) % 2 == 0:
    arcade.draw_xywh_rectangle_filled(x, y, SQ_SIZE, SQ_SIZE,
                                      WOOD_BROWN)
else:
    arcade.draw_xywh_rectangle_filled(x, y, SQ_SIZE, SQ_SIZE,
                                      BLACK)
```

အရောင် တလုညွှန်စီခြယ်တယ်။ ဒီလောက်ဆိုရင် ကျားကွဲက်ပရိုကရမ်ကို နားလည်မယ် ထင်ပါတယ်။

## ၄.၄ while loop

while loop သာ *indefinite loop* ဖြစ်ပါတယ်။ Loop ကို စတင် လုပ်ဆောင်တဲ့အချင်မှာ ဘယ်နှစ် ကြိမ် ပြန်ကျောမလဲ အတိအကျ ဖြို့ ‘မသိ’ ရင် indefinite loop လို့ သတ်မှတ်တယ်။ တစ်နဲ့ တစ်ဆယ် ကြား (အပါအဝင်) ကိန်းပြည့်တစ်လုံးကို ကျပ်န်း (random) ထွက်ထားမယ်။ မမှန်မချင်း အဒီဂါဏ်းကို ခန့်မှန်းပေးရမယ်။ ခန့်မှန်းတာ မှန်သွားရင် ဘယ်နှစ်ခါ ခန့်မှန်းရလဲနဲ့ မှန်တဲ့ကဏ်းကို ပြပေးတဲ့ ပရိုဂရမ် မှာ while loop အသုံးပြု ရေးထားတာ တွေ့ရပါမယ်။

```
from random import *

num = randint(1, 10)

guess = int(input('? '))
times = 1

while guess != num:
    guess = int(input('? '))
    times += 1

print(f'You get correctly after {times} guesses.')
print(f'The number is {num}.')
```

guess != num (ဘူလီယန် အိပ်စပရက်ရှင်) မဟားမချင်း loop က ပြန်ကျောပေးနေမှာပါ။ မှားတဲ့ အခါ ထပ်မကျော့တော့ဘဲ ရပ်သွားမှုဖြစ်တယ်။ ကျပ်န်းကဏ်းက ခုနှစ်ကျေတယ် ဆိုပါစို့။ တစ်၊ ကိုး၊ သုံး၊ တစ်ဆယ်နဲ့ ခုနှစ် တို့ကို အစဉ်အတိုင်း ခန့်မှန်း ထည့်ပေးမယ်ဆိုရင် တစ်ကျော်ချင်း လိုက်ကြည့်တဲ့အခါ အခုလို တွေ့ရမှာပါ။

```
num = randint(1, 10)                      # 7 ကျတယ် ယူဆပါ

guess = int(input('? '))                   # 1 ထည့်ပေးတယ်
times = 1

guess != num:                            # 1 != 7 ကဲ True
    1st iter:
        guess = int(input('? '))          # 9 ထည့်ပေးတယ် ယူဆပါ
        times += 1                         # 2 ဖွစ်သွားမယ်

guess != num:                            # 9 != 7 ကဲ True
    2nd iter:
        guess = int(input('? '))          # 3 ထည့်ပေးတယ် ယူဆပါ
        times += 1                         # 3 ဖွစ်သွားမယ်

guess != num:                            # 10 != 7 ကဲ True
    3rd iter:
        guess = int(input('? '))          # 10 ထည့်ပေးတယ် ယူဆပါ
        times += 1                         # 4 ဖွစ်သွားမယ်
```

```

guess != num:                                # 7 != 7 သော ဖြစ်သွားပါ
    # ထပ်မကျေတော့ဘူး

# loop အောက်က စတိတ်မန်တွေ ဆက်လုပ်ပါတယ်
print(f'You get correctly after {times} guesses.')
print(f'The number is {num}.')

```

Loop က ထပ်မကျေတော့ဘဲ ရပ်သွားတာကို *loop exits* ဖြစ်သွားတယ်လို့ ပြောတယ်။ မြန်မာ လိုတော့ loop ကနေ ထွက်သွားတယ်ပေါ့။ အကယ်၍ စော့စော့ ဥပမာ့မှာ ပထမဆုံးတစ်ခါမှာပဲ ခုနှစ် ထည့်လိုက်ရင်တော့ ဘလောက်ကို တစ်ခါမှာ မလုပ်ဆောင်ဘဲ loop က ချက်ချင်းထွက်သွားမှာပါ။

### Sentinel Controlled Loop

ကဏ္နားတွေ ဘယ်နှစ်လုံး ပေါင်းမှာလဲ ဤသိရင် `for` loop နဲ့ အလုပ်ဖြစ်တယ်။ ကဏ္နား ဘယ်နှစ်လုံးရှိ လဲ ကြိုမသိတားဘဲ ရှိသလောက် တစ်လုံးချင်းရှိက်ထည့်ပြီး အားလုံးပေါင်းချင်တာဆိုရင်တော့ `for` loop နဲ့ အဆင်မပြေား (မရနိုင်ဘူးလို့ မဆိုလိုပါ၊ မရမက လုပ်တဲ့နည်းတွေလည်း ရှိပါတယ်)။ ဒီလိုအခြေအနေ မျိုးမှာ *sentinel controlled loop* ကို သုံးပါတယ်။

Sentinel controlled loop မှာ loop ကနေ ထွက်ဖို့အတွက် အသုံးပြုတဲ့ သီးသန့်တန့်ဖို့တစ်ခု ရှိရပါမယ်။ ဒီတန့်ဖိုးကို *sentinel value* လို့ ခေါ်တယ်။ ကဏ္နားတွေပေါင်းတဲ့ ကိစ္စအတွက် *sentinel value* ရွေးချယ်သတ်မှတ်တဲ့အခါ ပေါင်းရမဲ့ကဏ္နား မဖြစ်နိုင်တဲ့ တစ်ခုကို ရွေးရပါမယ်။ သုညဟာ အပေါင်း ထပ်တူရ ဂုဏ်သွေးရှိတဲ့အတွက် ငါးကို *sentinel value* အဖြစ် ထားရိုင်တယ်။

```

# File: add_nums_sentinel.py
SENTINEL = 0
total = 0

val = int(input('? '))
while val != SENTINEL:
    total += val
    val = int(input('? '))

print(f'Total is {total}')

```

ထည့်ပေးတဲ့ ကဏ္နားဟာ သုညမဟုတ်မချင်း `total` မှာ ပေါင်းထည့်ပြီး နောက်ကဏ္နားတစ်ခု ထည့်ပေး ဖို့ `input` တောင်းမှာပါ။ Sentinel တန့်ဖိုး သုည ထည့်လိုက်ရင် loop က ထွက်သွားမယ်။ ပြီးတဲ့အခါ `total` ကို ပြေားမှာပါ။ စုချင်းပဲ သုည ထည့်လိုက်ရင် loop က တစ်ခါမှာ မကျေတဲ့ ထွက်သွားမယ်။ `total` လည်း သုညပဲ ထွက်ပါမယ်။ အခုပ်ရှိကျမ်းကို နောက်တစ်နည်း ရေးလို့ရပါတယ်။

```

# File: add_nums_sentinel2.py
SENTINEL = 0
total = 0

while True:
    val = int(input('? '))
    if val == SENTINEL:
        break

```

```

    total += val

print(f'Total is {total}')

```

while True: က ပုံမှန်ဆိုရင် infinite loop ဖြစ်ပါလိမ့်မယ်။ အမြဲမှန်နေတဲ့အတွက် ပြန်ကျော်  
တဲ့ ဘယ်တော့မှ ရပ်မှာ မဟုတ်ဘူး။ ဒီလိုဖြစ်နေတာကို ဖြတ်ချေပစ်ဖို့အတွက် break စတိတ်မန်ကို သုံး  
ထားပါတယ်။ if val == SENTINEL: (ထည့်ပေးတဲ့ တန်ဖိုးက သုညဆိုရင်) လက်ရှိ အလုပ်လုပ်နေ  
တဲ့ loop ကို break လိုက်မှာဖြစ်တယ်။

အထက်ပါ sentinel loop ပုံစံနှစ်ခုကို ယူဉ်ကြည့်ရင် ပထမတစ်ခုမှာ input တန်ဖိုးထည့်ဝိုင်းတာ  
ကို loop မစမိ တစ်ခါ ကြိုလုပ်ထားရတယ်။ နောက်တစ်ခုမှာတော့ အဲဒီလို ကြိုလုပ်ထားစရာမလိုဘူး။  
Loop ဘလောက်ထဲက စတိတ်မန်တစ်ခုကို အပြင်မှာထုတ်ထားရတဲ့အတွက် တချို့က ပထမပုံစံတက်  
ဒုတိယပုံစံက ကုဒ်စတိုင်လအားဖြင့် ပိုပြီး သပ်ရပ်ကြောရင်းတယ်လို့ ယူဆကြပါတယ်။

### Result Controlled Loop

တစ်နှစ်ငါးရာခိုင်နှစ်း အတိုးနဲ့ ဘဏ်မှာ ဒေါ်လာတစ်ထောင် စုထားတယ်။ တစ်နှစ်ပြည့်တိုင်း အတိုးအရင်း  
ပေါင်းပြီး နောက်နှစ်အတွက် အတိုးတွက်တယ်။ နှစ်ထပ်တိုး (yearly compound interest) လိုခေါ်ပါ  
တယ်။ ဒီနည်းလမ်းနဲ့ အနည်းဆုံး ဒေါ်လာတစ်သန်း စုမ္ပါဒ္ဓါ (မိလျုံနာတစ်ယောက်ဖြစ်ဖို့)  
ဘယ်နှစ်နှစ်တောင့်ရ မလဲ။ သုံးနှစ်ပြည့်ရင် စုမ္ပါမဲ့ ငွေပမာဏ တွက်ထားတာကို ကြည့်ပါ။။

Year	Interest	Balance
1	$1000 \times 0.05 = 50$	$1000 + 50 = 1050$
2	$1050 \times 0.05 = 52.5$	$1050 + 52.5 = 1102.5$
3	$1102.5 \times 0.05 = 55.125$	$1102.5 + 55.125 = 1157.625$

တေဘဲလ ဂ.၂ သုံးနှစ်စာ နှစ်ထပ်တိုး

ဒေါ်လာတစ်သန်း အနည်းဆုံးရှိဖို့ နှစ်ဘယ်လောက်ကြာမလဲ အခုလို ရှာနိုင်ပါတယ်

```

# File: one_million.py
balance = 1000
INT_RATE = 0.05
TARGET = 1_000_000
yr = 0

print(f'{yr:>4s} {'interest':>10s} {'balance':>10s}'")
while balance < TARGET:
    interest = balance * INT_RATE
    balance += interest
    yr += 1
    print(f'{yr:4d} {interest:10.2f} {balance:10.2f}'')

print(f'You have to wait {yr} years!')

```

while loop ထဲမှာ တစ်နှစ်ကုန်တဲ့အခါ ရမဲ့ အတိုးနဲ့ အတိုးအရင်းပေါင်း တွက်ထားပြီး yr ကိုလည်း

တစ်နှစ် တိုးပေးတယ်။ `yr`, `interest`, `balance` ထုတ်ပြပေးတယ် (မပြလည်း ပြသေနာမရှိပါ၊ တစ်နှစ် စာ တွက်ထားတာ စစ်ကြည့်လို့ရအောင် ထုတ်ကြည့်တာပါ)။ တစ်သန်းမပြည့်မချင်း ပြန်ကျော့ပေးအောင် loop ကွန်ဒီရှင်ကို `balance < TARGET` နဲ့ စစ်ထားတယ်။ တစ်သန်းနဲ့ညီရှင် (သို့) တစ်သန်းကျော့သွားတာနဲ့ ကွန်ဒီရှင် `False` ဖြစ်သွားပြီး ထပ်မကျော့တော့ဘူး။ Loop တစ်ခါကျော့တိုင်း တစ်နှစ်စာ အတိုးအရင်ပေါင်း `balance` ကို တွက်ချက်ပြီး loop ကနေ ဘယ်အချိန် ထွက်မလဲကလည်း အဲဒီရလဒ်အပေါ် မူတည်နေတာ ထွေရပါမယ်။ ဒီလို့ သဘောတရားရှိတဲ့ loop မျိုးကို result controlled loop လို့ ခေါ်ပါတယ်။ ကျော့မဲ့ အကြိမ်အရေအတွက်က loop ထဲမှာ တွက်ချက်ထားတဲ့ ရလဒ်ပေါ် မူတည်တယ်။

ပရိုဂရမ် output ကို အခုလို့ ကော်လံတစ်ခုစီကို အကျယ် တစ်သမတ်တည်းဖြစ်၊ စာသားတွေ ညာဘက်ကပ်ပြီး ညီနေအောင် f-strings ကို format spec လို့ခေါ်တဲ့ ဖော့မတ်သတ်မှတ်တဲ့ နည်းစနစ်ကို သုံးထားတာပါ။

```

yr      interest      balance
1        50.00       1050.00
2        52.50       1102.50
...
142     48602.79   1020658.53
You have to wait 142 years!

```

Format spec နဲ့ ပါတ်သက်ပြီး အကျယ်တဝ် မဖော်ပြတော့ဘဲ အခုလို့ ဖော့မတ်ရအောင် ဘယ်လိုလုပ်ထားလဲကိုပဲ ရှင်းပြပါမယ်။

```
f"{'yr':>4s} {'interest':>10s} {'balance':>10s}"
```

ကော်လံ ခေါင်းစည်း အတွက် f-string ပါ။ ဖော့မတ်လုပ်ရမဲ့ တန်ဖိုး/ဖေရီရောဲလ်/အိပ်စံပရဂ်ရှင်ကဲ့ (ကော်လံ) ဘယ်ဘက်မှာ ရှိမယ်။ ညာဘက်မှာ format spec ရှိမယ်။ `>4s` က '`yr`' အတွက် format spec ဖြစ်တယ်။ `>` က ညာဘက်ကပ်ဖို့ `4` က ကော်လံအကျယ်ကို ကာရက်တာ လေးလုံးသတ်မှတ်တာ။ `s` ကတော့ တန်ဖိုးကို string အနေနဲ့ ပြပေးပါလို့ ဆိုလိုတယ်။ ကျွန်ုတဲ့ ကော်လံနှစ်ခု အတွက် format spec ကို `>10s` သတ်မှတ်တယ်။ ကာရက်တာ ဆယ်လုံး ကော်လံအကျယ် သတ်မှတ်တယ်။

```
f'{yr:4d} {interest:10.2f} {balance:10.2f}'
```

ဒါကတော့ `yr`, `interest`, `balance` တန်ဖိုးတွေကို ပြပေးဖို့ပါ။ ကော်လံအကျယ် 4, 10, 10 သတ်မှတ်တယ်။ `yr` ကို ကိန်းပြည့်ကဏ္ဍားအနေနဲ့ ပြပေးအောင် `d` သုံးတယ်။ ကျွန်ုတဲ့နှစ်ခုကို ဒသေမနဲ့ ပြဖို့ `f` သုံးတယ်။ .2 ကတော့ ဒသေမနောက် ဂဏ်နှစ်လုံး ပြပေးခိုင်းတာ။ ကိန်းကဏ္ဍားဆိုရင် သူနှင့်အတိုင်း ညာဘက်ကပ်ပေးတဲ့အတွက် `>` ထည့်ဖို့ မလိုဘူး။ ဘယ်ဘက်ကပ်ချင်ရင် < သုံးလို့ရတယ်။

## ၃.၅ လေ့ကျင့်ရန် ဥပမာဏ်း

ကုန်ထရီးလ် စထရက်ချာတွေ အသုံးချုတ်တော်လာအောင် များများလေ့ကျင့်၊ များများစဉ်းစား၊ များများရေးကြည့် ရှုံးပါ။ ဘယ်အတတ်ပညာမဆို များများလေ့ကျင့်မှ ကျွမ်းကျင်လာနိုင်မှုပါ။ ဒီသဘောအရ ပရိုဂရမ်းမင်း ပညာရပ်ဟာလည်း ခွင့်းချက်မဖြစ်နိုင်ပါဘူး။

အောက်ပါ ဥပမာတစ်ချင်းကို ပုဂ္ဂိုနားလည်အောင်ဖတ်ပြီး ကိုယ်တိုင်စဉ်းစား ရေးကြည့်စို့ လေး လေးနက်နက် အကြံပြုပါတယ်။ ရေးပြီးသွားရင် ပုဂ္ဂိုမှာ ဖော်ပြထားချက်နဲ့ အညီ အလုပ်လုပ်/မလုပ် ဟာကွက်မရှိအောင် ဘယ်လိုစစ်ဆေး စိစစ်မလဲ မိမိဘာသာ စဉ်းစားကြည့်ပါ။

## Internet Delicatessen

Online ကနေ အစားအသောက်တွေ delivery ပိုပေးဖို့ အမှာလက်ခံတဲ့ ဆိုင်လေးတစ်ဆိုင်အတွက် ပရိ ဂရုံးပေးရပါမယ်။ အောင်ဒါမှာတဲ့အခါ မှာမဲ့ အစားအသောက် နံမည်၊ ဈေးနှုန်းနဲ့ အိပ်စံပရက်စံ delivery ယူမယူ ပရိုကရမ်မှာ ထည့်ပေးရပါမယ်။ ပရိုကရမ်က အောင်ဒါနဲ့ စုစုပေါင်းကျေသင့်ငွေကို ထုတ်ပေးရပါ မယ်။ တစ်သောင်းနဲ့အထက်မှာရင် delivery ဖို့ ပေးစရာမလိုပါ။ တစ်သောင်းအောက်ခို့ရင်တော့ နှစ်ရာ ပေးရပါမယ်။ အိပ်စံပရက်စံ delivery ယူမယ်ဆိုရင် သုံးရာအပို့ ထပ်ပေးရပါမယ်။

Enter the item: Tuna Salad

Enter the price: 4500

Express delivery (0==no, 1==yes): 1

Invoice:

Tuna Salad	4500
delivery	500
total	5000

```
itm_name = input('Item name: ')
price = int(input('Price: '))
is_ex_deli = int(input('Express delivery (0==no, 1==yes): '))

tot_deli_fee = 0
if price >= 10_000 and is_ex_deli == 1:
    tot_deli_fee = 300
elif price >= 10_000 and is_ex_deli == 0:
    tot_deli_fee = 0
elif price < 10_000 and is_ex_deli == 1:
    tot_deli_fee = 200 + 300
elif price < 10_000 and is_ex_deli == 0:
    tot_deli_fee = 200
else:
    print("You may have wrong value for express deli.")

tot_cost = price + tot_deli_fee

print("Invoice: ")
print(f'{itm_name:<10s} {price:8.2f}')
print(f'{delivery:<10s} {tot_deli_fee:8.2f}')
print(f'{total:<10s} {tot_cost:8.2f}')
```

အိပ်စံပရက်စံ delivery ယူမယူ တစ် (သို့) သုည ထည့်ပေးရမှာပါ။ တစ်/သုည မဟုတ်တဲ့ ကဏ္နားတစ်

ခု မှားထည့်မိရင် if...elif ကွန်ဒီရှင်တွေ တစ်ခုမှ True ဖြစ်မှာ မဟုတ်ပါဘူး။ မှားထည့်ထားတယ်လို့ သတိပေးဖို့ else အပိုင်းမှာ လုပ်ထားတယ်။ မူန်/မမူန် စိစစ်တဲ့အခါ အောက်ပါဇယားမှ ဖြစ်နိုင်ခြေအားလုံး ခြိုင်မံအောင် စစ်သင့်တယ်။ အိပ်စ်ပရက်စ် delivery အတွက် input ကဏ္ဍား မှားထည့်ရှင်သတိပေးတာကိုလည်း စစ်သင့်တယ်။ တစ်သောင်းဖိုး ဝယ်တာကို သုံးမျိုးစစ်ထားတာ လေ့လာကြည့်ပါ။ တစ်သောင်း မပြည့်တာနဲ့ တစ်သောင်းကျော်ဖိုး အတွက်လည်း အလားတူ စစ်ကြည်လို့ အားလုံးမျှန်တယ်ဆိုရင် ဒီပရိုဂရမ်မှာ bug ပါနိုင်ခြေ လုံးဝမရှိသလာက် ဖြစ်သွားပါပြီ။

---

10,000 MMK and above?      Express Delivery?

---

Yes	Yes
Yes	No
No	Yes
No	No

---

တေဘလ် ၃.၃ သုံးနှစ်စာ နှစ်ထပ်တိုး

Test Output:

```
Item name: Salad
Price: 10000
Express delivery (0==no, 1==yes): 1
Invoice:
Salad      10000.00
delivery    300.00
total      10300.00
```

```
Item name: Salad
Price: 10000
Express delivery (0==no, 1==yes): 0
Invoice:
Salad      10000.00
delivery    0.00
total      10000.00
```

```
Item name: Salad
Price: 10000
Express delivery (0==no, 1==yes): 2
You may have wrong value for express deli.
Invoice:
Salad      10000.00
delivery    0.00
total      10000.00
```

ပရိုဂရမ်တစ်ခုရဲ့ လိုအပ်ချက်ဟာ မကြာခဏ ပြောင်းလဲသွားလေ့ရှိတယ်။ အခုပရိုဂရမ်မှာ ဈေးနှုန်းသတ်မှတ်ချက်တွေ ပြောင်းလဲနိုင်တယ်။ အနည်းဆုံး တစ်သောင်းခွဲဝယ်မှ ပို့ခ free ရမှာဖြစ်ပြီး တစ်သောင်းခွဲအောက်ဆိုရင် ပို့ခ သုံးရွားဝါးဆယ် ပြောင်းလဲသတ်မှတ်လိုက်တဲ့အပြင် အိပ်စ်ပရက်စ် delivery

ကလည်း တစ်ရာရွေးထပ်တက်သွားတယ် ဆိုပါစို့။ တကယ့်လက်တွေမှာလည်း ဒါမံးဖြစ်လျှေးရှိပါတယ်။ ဒါ အတွက်ကို ပရိုဂရမ်ကို ပြင်ပေးရပါမယ်။ ဆိုင်ပိုင်ရှင်ကလည်း ရွေးနှင့်မကြာခဏ ပြောင်းစိုးနိုင်ကြောင်း ပြောလာပါတယ်။ နောင်လည်း ထပ်ပြင်ပေးဖို့လို့မဲ့ သဘောပါ။ လွှာယ်လွှာယ်ကူကူ ပြင်ပေးနိုင်ရင် အကောင်း ဆုံးပါ။ ပြင်ဆင်တဲ့အခါ မှားနိုင်ခြေနည်းဖို့လည်း အရေးကြီးပါတယ်။ ခုရေးထားတဲ့ အတိုင်းဆိုရင် ပြီးခဲ့တဲ့ ပရိုဂရမ်မှာ ပြဿနာရှုနေပါတယ်။

ပြီးခဲ့တဲ့ ပရိုဂရမ်မှာ ပြင်မယ်ဆိုရင် 10\_000 ကို လေးနေရာ၊ ပုံမှန်ပို့ခ 200 နဲ့ အပိုစ်ပရက်စ်အပိုကြေး 300 စတာတွေကို နှစ်နေရာစီ လိုက်ပြင်ရပါမယ်။ အလုပ်ရှုပ်တဲ့ အပြင် ပြင်ဖို့ကျန်ခဲ့တာလို့ မှားတာလည်း ဖြစ်နိုင်တယ်။ “Find and Replace” လုပ်မှာပေါ့လို့ စောဒကတက်စရာ ရှိပါတယ်။ အခုလို့ ကုဒ်လိုင်း နည်းနည်းပဲရိုတဲ့ ပရိုဂရမ်အသေးလေးမှာ အဆင်ပြနိုင်ပေမဲ့ ပို့ချုပ်တွေးပြီး ကုဒ်လိုင်းတွေများတဲ့ ပရိုဂရမ် မျိုးတွေမှာ “Find and Replace” လုပ်ရင် မပြင်သင့်တာတွေကိုပါ မရည်ရွယ်ပဲ ပြင်မိသွားတာဖြစ် တတ်ပါတယ်။ ပရိုဂရမ်ရေးတဲ့ အခါ ကျင့်သုံးရမဲ့ အလေ့အထကောင်းတစ်ခုက ကုဒ်ထဲမှာ ဒီတိုင်းချေရေးထား တဲ့ တန်ဖိုးတွေ (literal constants) တွေကို နာမည်ပေးထားတာပါ။ အပေါ်က ပရိုဂရမ်မှာ literal constants တွေချည်း သုံးထားတယ်။ နံမည်ပေးထားတဲ့ constants (named constants) တွေ အဖြစ် ပြောင်းရေးသင့်တယ်။

```

FREE_DELI_AMT = 15_000
DELI_FEE = 350
EXP_DELI_FEE = 400

itm_name = input('Item name: ')
price = int(input('Price: '))
is_exp_deli = int(input('Express delivery (0==no, 1==yes): '))

tot_deli_fee = 0

if price >= FREE_DELI_AMT and is_exp_deli == 1:
    tot_deli_fee = EXP_DELI_FEE
elif price >= FREE_DELI_AMT and is_exp_deli == 0:
    tot_deli_fee = 0
elif price < FREE_DELI_AMT and is_exp_deli == 1:
    tot_deli_fee = DELI_FEE + EXP_DELI_FEE
elif price < FREE_DELI_AMT and is_exp_deli == 0:
    tot_deli_fee = DELI_FEE
else:
    print("You may have wrong value for express deli.")

tot_cost = price + tot_deli_fee
print("Invoice: ")
print(f'%-10s %.2f' % (itm_name, price))
print(f'%-10s %.2f' % ('delivery', tot_deli_fee))
print(f'%-10s %.2f' % ('total', tot_cost))

```

ဒီပရိုဂရမ်ကို cascading if မသုံးဘဲ ရှိုးရှိုး if နဲ့ ရေးလို့လည်း ရတယ်။

| FREE\_DELI\_AMT = 15\_000

```

DELI_FEE = 350
EXP_DELI_FEE = 400

itm_name = input('Item name: ')
price = int(input('Price: '))
is_exp_deli = int(input('Express delivery (0==no, 1==yes): '))

tot_deli_fee = 0

if price < FREE_DELI_AMT:
    tot_deli_fee += DELI_FEE

if is_exp_deli == 1:
    tot_deli_fee += EXP_DELI_FEE

if not (is_exp_deli == 0 or is_exp_deli == 1):
    print("You may have wrong value for express deli.")

tot_cost = price + tot_deli_fee

print("Invoice: ")
print(f'%-10s %8.2f' % (itm_name, price))
print(f'%-10s %8.2f' % ('delivery', tot_deli_fee))
print(f'%-10s %8.2f' % ('total', tot_cost))

```

ပထမ if က delivery အ ပေါ် လိုတယ်ဆိုရင် tot\_deli\_fee မှာ DELI\_FEE ပေါင်းထည့်ပေးတယ်။ ဒုတိယ if က အိပ်စပ်ရက်စ် delivery ယူမယ်ဆိုရင် tot\_deli\_fee မှာ EXP\_DELI\_FEE ထပ်ပေါင်းထည့်တယ်။ အောက်ဆုံး if ကတော့ အိပ်စပ်ရက်စ် delivery အတွက် တစ်နှုန်း သုည မဟုတ်ဘာ ထည့်မိရင် သတိပေးစာသား ပြပေးတယ်။

ဒီပုဂ္ဂရမ်နဲ့ ဒီမတိုင်ခင် သူရှေ့က ပရိုဂရမ်က ဖြစ်နိုင်ခြေအားလုံးအတွက်တော့ ရလဒ် တူတူမထွက်ပါဘူး။ အခြေအနေ တစ်ခုကလဲလို့ ကျွန်တော့အတွက်တော့ ရလဒ်တူပါတယ်။ တစ်သေင်းငါးတောင် ထက်ငယ်တဲ့ တန်ဖိုးနဲ့ အိပ်စပ်ရက်စ် delivery အတွက် ကဏ္ဍနဲ့ လွှဲထည့်ကြည့်ပါ။ ဥပမာ

```

Item name: salad
Price: 12000
Express delivery (0==no, 1==yes): 2
You may have wrong value for express deli.

Invoice:
salad      12000.00
delivery     0.00
total       12000.00

```

```

Item name: salad
Price: 12000
Express delivery (0==no, 1==yes): 2
You may have wrong value for express deli.

```

Invoice:

salad	12000.00
delivery	350.00
total	<b>12350.00</b>

နောက်ဆုံး ပရိုဂရမ်က အပိုဒ်ပရက်စံ delivery အတွက် မပေါင်းထည့်ပေမဲ့ ပုံမှန် delivery ခ သုံးရှုင်းဆယ်ကိုတော့ ထည့်ပေါင်းသွားတာ တွေ့ရမယ်။ မှားထည့်တာကိုပဲ သတိပေးစာသားပြေပေးတာ၊ ထည့်မတွက်သွားတာက ပိုပြီး သဘာဝကျတယ်လို့ ယူဆရမှာပါ။

ပြင်ပကနေ ထည့်ပေးတဲ့အခါ မဖြစ်သင့်တဲ့ input တန်ဖိုးတွေ ဝင်မလာအောင် စိစစ်တာကို input validation လို့ ခေါ်တယ်။ တကယ့် လက်တွေ့ အသုံးချ ပရိုဂရမ်တွေမှာ input validation လုပ်ထားဖို့ အရေးကြီးပေမဲ့ ဘိုင်နာအဆင့် လေ့လာတဲ့ ဥပမာတွေမှာတော့ လေ့လာရင်းကိစ္စကနေ လမ်းကြောင်းမချော်သွားအောင် ဆင်ခြင်ရမှာဖြစ်ပြီး သင့်တော်ရဲ့ ဆက်စပ်ရှင်းပြပါမယ်။

## တာရာ ပရက်ရှာ

ကားဘီးလေထိုးတာဟာ ကားရဲ့ စွမ်းဆောင်ရည်ရော အနဲ့ရှုယ်က်းဖို့အတွက်ပါ ပဓနကျပါတယ်။ ကားတ်စီးအတွက် အကြိုပြုထားတဲ့ တာယာပရက်ရှာ (recommended pressure) အပေါ် မူတည်ပြီး လေ ဘယ်လောက်တင်းလို့ရလဲ လျှော့လို့ရလဲ ရှိပါတယ်။ ဥပမာ recommended pressure က 35 psi (pounds per square inch) ဖြစ်ရင် အလွန်ဆုံး 31.5 psi ထိ လေလျှော့လို့ရပါတယ်။ လေပိုတင်းမယ်ဆိုရင်လည်း အလွန်အလွန်ဆုံး 44 psi အထိ ရပါနိုင်ပါတယ်။

ရှုံးတာယာနှစ်လုံး ပရက်ရှာအနီးစပ်ဆုံးတူသင့်ပြီး 3 psi အထိ ကွာဟလို့ရတယ်။ ကွာဟချက်က 3 psi ထက်တော့ မများသင့်ဘူး။ နောက်တာယာနှစ်လုံးလည်း ထိန်ည်းတူစွာပဲ ဖြစ်တယ်။ ကားမော်ဒယ်အလိုက် recommended pressure ကွာခြားပေမဲ့ အမိန့်းကားအများစုအတွက် 35 psi ဖြစ်တယ်လို့ ယူဆပြီး တာယာပရက်ရှာ အိုကောမကေ စစ်ပေးတဲ့ ပရိုဂရမ် ရေးပေးရပါမယ်။ Input အနေနဲ့ တာယာတစ်ခုချင်းအတွက် ပရက်ရှာ psi တန်ဖိုး ထည့်ပေးရမှာပါ။ ထည့်ပေးလိုက်တဲ့ တာယာပရက်ရှာ 31.5 psi ထက်နည်းနေရင် သို့မဟုတ် 44 psi ထက်များနေတာနဲ့ သတ်မှတ်ဘောင် မဝင် (out of range) ဖြစ်နေကြောင်း သတိပေးရပါမယ်။ တာယာအားလုံးရဲ့ ပရက်ရှာတွေ သတ်မှတ်ဘောင်အတွင်း ဝင်တယ်။ ရှုံးတာယာနှစ်လုံး ကွာဟချက်၊ နောက်တာယာနှစ်လုံး ကွာဟချက်တွေ ခွင့်ပြုလို့ရတာထက် မပိုဘူးဆိုရင် လေထိုးထားတာ အိုကေတယ်။ တာယာတစ်လုံး out of range ဖြစ်နေတာနဲ့ လေထိုးထားတာ မအိုကေဘူး ပြပေးရပါမယ်။

```

MIN_ALLOWABLE = 31.5
MAX_ALLOWABLE = 44.0
WARNING = 'Waring: Pressure is out of range!'
LEFT_RIGHT_DIFF_ALLOWABLE = 3.0

is_out_of_range = False

front_left = float(input("Front left pressure: "))
if not (MIN_ALLOWABLE <= front_left <= MAX_ALLOWABLE):
    is_out_of_range = True
    print(WARNING)

```

```

front_right = float(input("Front right pressure: "))
if not (MIN_ALLOWABLE <= front_right <= MAX_ALLOWABLE):
    is_out_of_range = True
    print(WARNING)

rear_left = float(input("Rear left pressure: "))
if not (MIN_ALLOWABLE <= rear_left <= MAX_ALLOWABLE):
    is_out_of_range = True
    print(WARNING)

rear_right = float(input("Rear right pressure: "))
if not (MIN_ALLOWABLE <= rear_right <= MAX_ALLOWABLE):
    is_out_of_range = True
    print(WARNING)

front_diff = abs(front_left - front_right)
front_diff = abs(rear_left - rear_right)
if (front_diff <= LEFT_RIGHT_DIFF_ALLOWABLE
    and rear_diff <= LEFT_RIGHT_DIFF_ALLOWABLE
    and not is_out_of_range):
    print("Inflation is OK.")
else:
    print("Inflation is not OK!")

```

is\_out\_of\_range ဘူလိယန် သုံးထားတာ နည်းနည်း ရှင်းပြို လိုပါမယ်။ စစချင်း False တန်ဖိုး ထည့်ထားတာ တွေ့ရမှာပါ။ if စတိတ်မန်တွေက တာယာတစ်ခုချင်းကို out of range ဖြစ်နေလား စစ် ထားတာတွေ့ရတယ်။ တာယာတစ်ခု out of range ဖြစ်တာနဲ့ is\_out\_of\_range က True ဖြစ်သွား မှာပါ။

အောက်ပိုင်းမှ front\_diff နဲ့ rear\_diff ရှာတဲ့အခါ abs ဖန်ရှင်းနဲ့ ပကတိတန်ဖိုး ယူထားတာ သတိပြုပါ။ ပရက်ရှာ ခြားနားချက်ရှာတဲ့အခါ အနှံတ်တန်ဖိုး ထွက်နိုင်တဲ့အတွက်ကြောင့်ပါ။ ဘယ်ဘက် တာယာက ပရက်ရှာနည်းနေတဲ့အခါ (ဥပမာ  $32 - 37 = -5$ ) အနှံတ်တန်ဖိုး ဖြစ်နေမယ်။ ဒါကြောင့် ပကတိတန်ဖိုးယူမှုပဲ ကွာဟာချက် 3 psi ကျော်မကျော်စစ်ပေးတဲ့အခါ အဖြော်ရပါမယ်။

```

front_diff <= LEFT_RIGHT_DIFF_ALLOWABLE
and rear_diff <= LEFT_RIGHT_DIFF_ALLOWABLE
and not is_out_of_range

```

and နှစ်ခုနဲ့ ဆက်ထားရင် သုံးခုလုံးမှန်မှပဲ True ထွက်မှာပါ။ တစ်ခုမှားတာနဲ့ အိပ်စ်ပရက်ရှင်တစ်ခုလုံး ရလဒ် False ပဲ။ Out of range မဖြစ်ရဘူး ဆိုတာကို not is\_out\_of\_range နဲ့ စစ်ထားတယ်။ is\_out\_of\_range တန်ဖိုး False ဖြစ်မှ not is\_out\_of\_range က True ဖြစ်မယ်။

# အခန်း ၈

## ဖန်ရှင်များ

ဖန်ရှင်တွေဟာ ပရီဂရမ် အဆောက်အအုံတစ်ခုရဲ့ အခြေခံ အုပ်ချပ်တွေပါ။ ဒီအခြေခံ အုပ်ချပ်တွေကိုမူ ကောင်းကောင်းမွန်မွန် နားလည် အသုံးမချတတ်ရင် ကျမ်းကျင်တဲ့ ပရော်ဖက်ရှင်နယ် ပရီဂရမ်မာ တစ်ယောက် မဖြစ်နိုင်ဘူးဆိုတာ အထူးပြောစရာ လိုမယ် မထင်ပါဘူး။ ကားရဲလှို့မှာလည်း ဖန်ရှင် အခြေခံ သဘောတရားနဲ့ အသုံးချပုံကို လေ့လာခဲ့ကြရပါတယ်။ ဒီအခန်းမှာတော့ တန်ဖိုးပြန်ပေးခြင်း၊ ပါရာမီတာ တွေ အသုံးပြုပုံနဲ့ exception-handling တို့ကို ဆက်လက် လေ့လာကြမှာပါ။ လက်တွေနဲ့ နီးစပ်ပြီး အမှန် တကယ် အသုံးဝင်ဗဲ့ ပရီဂရမ်လေးတွေကို အစကနေ အဆုံးထိ တစ်ဆင့်ပြီးတစ်ဆင့် ဘယ်လိုဒီမိမိပြုလုပ် တည်ဆောက်လဲ ကိုလည်း ဖော်ပြပေးသွားမှာပါ။

### ၈.၁ တန်ဖိုးပြန်ပေးတဲ့ ဖန်ရှင်များ

အခန်း (၅) မှာ ဖော်ပြခဲ့တဲ့ နှစ်ထပ်ကိန်းရာတဲ့ square ဖန်ရှင်ကိုပဲ အသေးစိတ် တစ်ခါထပ်ကြည့်ရအောင်။ ဒီလောက် ရှင်းရှင်းလေးကို အကျယ်ချွဲနေတယ်လို့ ထင်ကောင်း ထင်ပါလိမ့်မယ်။ နည်းနည်းတော့ စိတ်ရှည်သည်းခဲ့ ပေးရပါမယ်။ အခြေခံကျတဲ့ သဘောတရားတွေ ကျောက်ထားမှ ရွှေ့ဆက်တဲ့ အခါ လွှာယ်ကူးမှ မိုလိုပါ။

```
>>> def square(x):
...     return x ** 2
...
>>>
```

ဦးကိုကွင်းထဲက ဖော်ရောဘဲလ် x က ဖန်ရှင် ပါရာမီတာ (parameter) ဖြစ်ပြီး ဖန်ရှင်ခေါ်တဲ့အခါ ထည့်ပေးမဲ့ အူဂုံမန်း (argument) တန်ဖိုးကို ကိုယ်စားပြုတယ်။ return စတိတ်မန်က ဖန်ရှင်ခေါ်တဲ့နေရာ ကို တန်ဖိုးပြန်ပေးတဲ့ စတိတ်မန်ပါ။

ဖန်ရှင်အသုံးပြုတာကို function call လုပ်တယ်လို့ သိထားပြီးပါပြီ။ မြန်မာလိုတော့ ‘ဖန်ရှင်ခေါ်တယ်’ သို့မဟုတ် ‘ဖန်ရှင်ကောလတယ်’ လို့ အပြောများတယ်။ ဖန်ရှင်ခေါ်တဲ့ ပုံစံက ဒီလိုပါ

```
>>> square(2.5)
6.25
```

အခါ ဖန်ရှင်ကောလ် အတွက် ပါရာမီတာ x ရဲ့ တန်ဖိုးက 2.5 ဖြစ်မှာပါ။ (ဖန်ရှင်ခေါ်တဲ့အခါ ပါရာမီတာ

ဖေရီရောလဲ x ကို အားဂုံးမန်လုပ်ပေးတယ်လို့ ယူဆနိုင်တယ်။ ဒီကိစ္စအတွက် အားဂုံးက 2.5 ဖြစ်တယ်။ အားလုံးသိပြီး ဖြစ်တဲ့အတိုင်း ဖန်ရှင်ခေါ်ချင် ဖန်ရှင်ဘေးလောက်ကို လုပ်ဆောင်ပေးမှာပါ။ ဖန်ရှင်ဘေးလောက်ထဲက return စတိတ်မန် လုပ်ဆောင်တဲ့အခါ အိပ်စ်ပရက်ရှင် x \*\* 2 ကို တန်းအရင်ရှာတယ်။ 6.25 ရတယ်။ ဒီတန်းကို ဖန်ရှင်ခေါ်ဘားတဲ့ နေရာကို return က ပြန်ပိုပေးလိုက်တာပါ။ အောက်ပါ ဖန်ရှင်ကောလှမှုလည်း ဒီဖြစ်စဉ် သဘောအတိုင်း တစ်ခါထပ်ဖြစ်မှာ ဖြစ်တယ်။

```
>>> a = 1024
>>> result = square(a)
>>> result
1048576
```

အခုံတစ်ခါ ပါရာမီတာ x ဟာ အားဂုံး a ရဲ့ တန်းမှုး ဖြစ်တယ် ( $x = a$  အဆိုင်းမန် လုပ်တဲ့သဘောပါ)။  $x ** 2$  က ရလာတဲ့ 1048576 ကို ဖန်ရှင်ခေါ်တဲ့ နေရာက ပြန်ရတယ်။ နောက်ဆုံးတော့ ဒီတန်းကို result မှာ အဆိုင်းမန်လုပ်တယ်။ ဖြစ်စဉ်အရ ရုံးရှင်းပါတယ်။

```
>>> x = 10
>>> square(x)
```

ဒီလိုဆိုရင်ရော ဘယ်လို့ ဖြစ်မလဲ။ နည်းနည်းထူးခြားတာက အားဂုံးနှင့် ပါရာမီတာ နံမည်တူနေတာ။ ပါရာမီတာရဲ့ စကုပ်ဟာ ဖန်ရှင်သတ်မှတ်ချက် အတွင်းမှာပဲ ရှိတယ်လို့ ယူဆရမှာပါ။ ဒါကြောင့် အားဂုံး x နဲ့ ပါရာမီတာ x နဲ့က သီးခြား ဖေရီရောဘဲလဲတွေ။

```
>>> u = 15
>>> t = 5
>>> square(u + 2*t)
```

အားဂုံးက အိပ်စ်ပရက်ရှင် ဖြစ်နေရင် တန်းအရင်ရှာပြီး ရလဒ်ကို ပါရာမီတာနဲ့ အဆိုင်းမန် လုပ်ပါတယ် ( $x = u + 2*t$ )။

```
>>> z = square(2.0) + 5
>>> square(z)
81.0
>>> square(square(2.0) + 5)
81.0
```

ဒုတိယ ဖန်ရှင်ခေါ်တဲ့နေရာမှာ အိပ်စ်ပရက်ရှင်ကို z နဲ့ အဆိုင်းမန် မလုပ်တော့ဘဲ တစ်ခါတည်း အားဂုံး အနေနဲ့ ထည့်လိုက်တာပါ။ သဘောတရား တူတူပါပဲ။

ဖန်ရှင် return လုပ်တဲ့ သဘောကို နားလည်ထားဖို့လည်း အရေးကြီးတယ်။ return စတိတ်မန် ဟာ ဖန်ရှင်ကနေ တန်းမှုးတဲ့ခုကို ဖန်ရှင်ခေါ်တဲ့ဆိုကို ပြန်ပေးတယ်လို့ သိထားပြီးပါပြီ။ ဖန်ရှင်ထဲကနေ return လုပ်လိုက်တာနဲ့ ခေါ်ဘားတဲ့နေရာကို ချက်ချင်း ပြန်ရောက်သွားတာ။

```
>>> def get_sign(r):
...     if r > 0:
...         return 'positive'
...     elif r < 0:
...         return 'negative'
...     else:
```

```

...
    return 'zero/nosign'
...
>>>
>>> '10 is ' + get_sign(10)
'10 is positive'

```

အခုအိပ်စံပရက်ရှင်ရဲ့ တန်ဖိုးရှာဖို့ `get_sign(10)` ခေါ်လိုက်တဲ့အခါ လက်ရှိနေရာကနေ လုပ်ဆောင်မှုက ဖန်ရှင်သာလောက်ဆီ ပြောင်းရွှေ့ ရောက်ရှိသားပါမယ်။ ဖန်ရှင်ထဲက စတိတ်မန်တွေ အစဉ်အတိုင်း စတင် လုပ်ဆောင်တယ်။ ဖန်ရှင်က `return` လုပ်တဲ့အခါ လုပ်ဆောင်မှုက ဖန်ရှင်သာလောက်ထဲကနေ ခေါ်ခဲ့တဲ့ နေရာကို တယန်ပြန်၍ ပြောင်းရွှေ့သွားတယ်။ `return` ပြန်လိုက်တဲ့ တန်ဖိုးကို ဖန်ရှင်ခေါ်တဲ့နေရာမှာ ရရှိပြီး လုပ်လောက်စ အိပ်စံပရက်ရှင်ကို ဆက်လုပ်ပါတယ်။ ဒီလိုမြင်ကြည့်ပါ ...

```

def get_sign(r):
    if r > 0:
        return 'positive'
    elif r < 0:
        return 'negative'
    else:
        return 'zero/nosign'

'10 is ' + get_sign(10)

```

မြားအနေက်က ဖန်ရှင်ခေါ်လိုက်တဲ့အခါ လုပ်ဆောင်မှု ပြောင်းရွှေ့သွားတာကို ပြတယ်။ မြားအနီက `return` ပြန်တဲ့အခါ ခေါ်ခဲ့တဲ့နေရာ ပြန်ရောက်သွားဘာကို ပြတာပါ။

ဆက်လက်ပြီး ပါရာမီတာ တစ်ခုထက်ပိုတဲ့ ဖန်ရှင်တူဂျို့ကို ကြည့်ပါမယ်။ ပါရာမီတာဆိတာ ဖန်ရှင် အတွက် လိုအပ်တဲ့ `input` ကို လက်ခံတဲ့ ပေရီရောကဲလုပ်ပါပဲ။ ထောင့်မှန်စတုဂံရဲ့ အလျေားနဲ့ အနံကနေ ရော့ယာရှာပေးတဲ့ ဖန်ရှင်က ဒီလိုပါ

```

def rect_area(wid, len):
    return wid * len

```

ဖန်ရှင်တစ်ခုကို အခြေခံ အပ်ချပ်သဖွယ် အသုံးပြု၍ အခြားဖန်ရှင်တွေ တည်ဆောက်ယူနိုင်တယ်။ `rect_area` ကို `box_vol` မှာ သုံးထားတာပါ

```

def box_vol(w, l, h):
    return rect_area(w, l) * h

```

ဒီဖန်ရှင်ကို ခေါ်ရင် ဘယ်လိုဖြစ်မလဲ ကြည့်တတ်သင့်တယ်။ အခုလို ခေါ်မယ် ဆိုပါစို့

```
>>> box_vol(10, 5, 3)
```

`w=10, l=5, h=3` ဖြစ်တယ်။ ဖန်ရှင် ဘလောက်ထဲကို ရောက်သွားမယ်။ `return` ပြန်ပေးဖို့ အိပ်စံပရက် ရှင်ကို တန်ဖိုးရှာပါတယ်

```
rect_area(w, l) * h
```

`rect_area` ဖန်ရှင်ခေါ်တယ်။ `wid=w, len=l` ဖြစ်မယ်။ အခုကိစ္စအတွက် ပါရာမီတာနှစ်ခုရဲ့ တန်ဖိုး

က 10 နဲ့ 5 အသီးသီး ဖြစ်မယ်။ 50 ရပါမယ်။ 50 \* h ကို တန်ဖိုးဆက်ရှုပြီး ရလာတဲ့ 150 ကို box\_no1 ခေါ်ထားတဲ့နေရာကို return ပြန်ပေးမှာ ဖြစ်တယ်။ အခြေခံသဘာတရားတွေ သိပြီးတဲ့အခါ အတန်အသင့်ရှုပ်ထွေးတဲ့ ဖန်ရှင်တချို့ကို ကြည့်ပါမယ်။

### ဖန်ရှင်များနှင့် အက်ဘ်ခက်ရှင်းလုပ်ခြင်း

မွေးသက္ကရာဇ် (date of birth) ကနေ အသက် တွေ့က်ပေးတဲ့ ဖန်ရှင်ကို လေ့လာကြည့်ပါ။ အသက်တွေ့က် တဲ့ လေ့ဂျစ်ကို မရှင်းပြတော့ဘူး။ လေ့ကျင့်ခန်းအနေနဲ့ မိမိဖာသာ နားလည်အောင်ကြည့်ပါ။

```
# File: age_today.py
from datetime import *

def age_today(dob):
    today = date.today()
    this_bd = dob.replace(year=today.year)
    if today - dob >= this_bd - dob:
        return today.year - dob.year
    else:
        return today.year - dob.year - 1

print(age_today(date(1990, 4, 2)))
```

ဖန်ရှင်အတွင်းပိုင်း လေ့ဂျစ်တွေ ဘယ်လိုပဲ ရှုပ်ထွေးပါစေ၊ အသုံးပြုရတာကတော့ မခက်ပါဘူး။ ဖန်ရှင် ခေါ်တဲ့အခါ ဘယ်လို တည်ဆောက်ထားလဲ အတွင်းပိုင်း အယ်လ်ဂိုရှစ်သမ်တွေ၊ လေ့ဂျစ်တွေ သိစရာမ လိုဘဲ သုံးရတာပါ။ ဖန်ရှင်က ငှင့်ရဲ့ အတွင်းပိုင်း ကုဒ်တွေကို အက်ဘ်စရက်ရှင်း (abstraction) လုပ်ပေးလိုက်တာ ဖြစ်တယ်။ ဒါဟာ ဖန်ရှင်ရဲ့ အရေးပါဆုံး ဂုဏ်သွေးလိုလို ဆိုရင်လည်း မမှားဘူး။

age\_today ဖန်ရှင်ဟာ ပို့ကြီးတဲ့ ပရိုက်မှုတစ်ခုရဲ့ တစ်စိတ်တစ်ပိုင်း ဖြစ်လာနိုင်ပါတယ်။ ပရိုက်မှု အသေးစားလေးတစ်ခုမှာ အသုံးပြုထားတာကို လေ့လာကြည့်ပါ။ နိုင်ငံအများစုံမှာ (၁၈) နှစ် မပြည့်လေး တဲ့သူကို ဆေးလိပ်ရောင်းခွင့် မရှိဘူး။ ဥပဒေရှိပါတယ်။ စားသုံးသူရဲ့ မွေးသက္ကရာဇ် ထည့်ပေးလိုက်တာနဲ့ ရောင်းလို့ ရ/မရ ပြပေးတဲ့ ပရိုက်မှုလေးပါ။

```
# File: sell_cigarette.py
from datetime import *

def age_today(dob):
    today = date.today()
    this_bd = dob.replace(year=today.year)
    if today - dob >= this_bd - dob:
        return today.year - dob.year
    else:
        return today.year - dob.year - 1

def can_by_cig(dob):
    age = age_today(dob)
    return True if age >= 18 else False
```

```

def main():
    """
    Given date of birth, this program tells whether the customer
    is eligible to buy cigarette or not.

    Enter 'exit' to quit the program.
    """

    print("Please enter 'quit' to exit this program.")
    while True:
        dobstr = input('Enter date of birth (yyyy-mm-dd): ')
        if dobstr == 'quit': break
        dob = date.fromisoformat(dobstr)
        print(dob)
        if can_by_cig(dob):
            print("Okay!")
        else:
            print('Too young to sell cigarette!')
    print('Program exited...')

```

---

```

if __name__ == "__main__":
    main()

```

age\_today ဖန်ရှင်ဟာ အခြားပရီဂရမ်တွေမှာလည်း အသုံးကျနိုင်ပါတယ်။ ဒီဖန်ရှင်ကို ခဏာကေ မရေးရှုဘဲ လိုအပ်တဲ့ ပရီဂရမ်တွေကနေ အလွယ်တကူ ပြန်သုံးလို့ရအောင် လိုက်ဘရီ ဒါမှုမဟုတ် မော်ဒူးလို့ခေါ်တဲ့ ကုစ်အဖွဲ့အစည်း ယူနစ်တစ်ခုအနေနဲ့ ထုတ်ထားနိုင်တယ်။ လိုက်ဘရီ (သို့) မော်ဒူးတစ်ခု ဟာ ဆက်စပ်တဲ့ ဖန်ရှင်တွေ၊ ကလပ်စွဲတွေနဲ့ ဖွဲ့စည်းထားတာ ဖြစ်တယ်။ နောက်ပိုင်းမှာ ဒါန့်ပါတ်သက်ပြီး သီးခြားလေ့လာရှုပါ။

## ၈.၂ တန်ဖိုးပြန်မပေးတဲ့ ဖန်ရှင်များ

ဖန်ရှင်အားလုံးတော့ တန်ဖိုးပြန်မပေးတဲ့ ဖန်ရှင်တွေ မဟုတ်ကြပါဘူး။ တန်ဖိုးပြန်မပေးတဲ့ ဖန်ရှင်တွေလည်း ရှိတယ်။ ဥပမာ output ထုတ်တဲ့ print ဖန်ရှင်ဟာ တန်ဖိုးပြန်မပေးတဲ့ ဖန်ရှင်မျိုးပါ။ အောက်ပါ print\_sign ဖန်ရှင်ဟာ get\_sign နဲ့ ဆင်တူပေမဲ့ တန်ဖိုး return ပြန်မပေးပါဘူး။

```

def print_sign(r):
    if r > 0:
        print('positive')
    elif r < 0:
        print('negative')
    else:
        print('zero/nosign')

```

ဒီဖန်ရှင်မှာ return မပါတာ တွေ့ရပါမယ်။ ကားခဲ့လုပ်ဖန်ရှင်တွေမှာလည်း return မသုံးခဲ့တာ ပြန် အမှတ်ရှုမှုပါ။ append\_n\_times ကို လေ့လာကြည့်ပါ

```
def append_n_times(lst, item, n):
    for i in range(n):
        lst.append(item)

lst = []
append_n_times(lst, 'hello', 10)
print(lst)
```

အိုက်တမ်တစ်ခုကို သတ်မှတ်ထားတဲ့ အရေအတွက်ပြည့်အောင် list တစ်ခုနောက်ကနေ ဆက်ပေးတယ်။ နုဂ္ဗာ list မှာ အိုက်တမ်တွေ တိုးသွားပြီး စတိတ်အပြောင်းအလဲ ဖြစ်စေတယ်။

Output ထုတ်တဲ့ ဖန်ရှင်တွေဟာ တန်ဖိုးပြန်ပေးလေ့မရှိဘူး။ စခရင်မှာ စာသား (သို့) ရုပ်ပုံ ပြပေး တာဟာ output ဖြစ်တယ်။ ဖိုင်တစ်ခုမှာ ရေးတာလည်း output ပဲ (ဥပမာ Python ကုဒ်ဖိုင်ကို ပြင် ပြီး save လုပ်တာ)။ အော့သံဂျက် စတိတ်ကို ပြောင်းလဲစေတဲ့ ဖန်ရှင်တွေဟာလည်း တန်ဖိုးပြန်မပေး တဲ့ ဖန်ရှင်တွေ ဖြစ်လေရှိတယ် (ဥပမာ list ရဲ့ append နဲ့ insert ဖန်ရှင်)။ စတိတ်အပြောင်းအလဲ ဖြစ်စေတဲ့ ဖန်ရှင်အားလုံး တန်ဖိုးပြန်မပေးတာတော့ မဟုတ်ဘူး။ ဥပမာ pop ဟာ တန်ဖိုးပြန်ပေးပါတယ်။ စတိတ်အပြောင်းအလဲလည်း ဖြစ်စေတယ်။

တန်ဖိုးပြန်တဲ့ ဖန်ရှင်ပဲ return ပြန်လိုက်တာ မဟုတ်ပါဘူး။ တန်ဖိုးပြန်မပေးတဲ့ ဖန်ရှင်တွေမှာလည်း return ပါနိုင်ပါတယ်။ print\_sign ကို ဒီလိုရေးလိုလည်း ရပါတယ်

```
def print_sign2(r):
    if r > 0:
        print('positive')
        return
    elif r < 0:
        print('negative')
        return
    else:
        print('zero/nosign')
        return
```

တန်ဖိုးပြန်မပေးတဲ့အတွက် return ပဲဖြစ်ရပါမယ်။ တန်ဖိုး/အိပ်စံပရက်ရှင် တဲ့ပြီး ပါလိုမရပါဘူး။ ဖန်ရှင်ဘလောက် ပြီးတဲ့အခါ ခေါ်တဲ့နေရာကို ပြန်ရောက်သွားရမှာ ဖြစ်တဲ့အတွက် return မပါတဲ့ ဖန်ရှင် တွေရဲ့ ဘလောက်အဆုံးမှာ return ရှိတယ်လို့ ယူဆနိုင်တယ်။ ဥပမာ return မပါတဲ့ print\_sign ကို အခုလို ယူဆနိုင်တယ်

```
def print_sign(r):
    if r > 0:
        print('positive')
    elif r < 0:
        print('negative')
    else:
        print('zero/nosign')
    return
```

(ဖန်ရှင် ဘလောက်အဆုံးမှာ return လုပ်ထားတယ်လို့ ယူဆရမှာပါ)။

ပိုမြို့တိတိကျကျ ပြောမယ်ဆိုရင် Python မှာ တန်ဖိုးပြန်မပေးတဲ့ ဖန်ရှင်တွေက None အော့သိ ဂျက် ပြန်ပေးပါတယ်။ None အော့သိဂျက်ကို 'တန်ဖိုးတစ်စုံတစ်ရာမရှိ' (သို့) 'မည်သည့် တန်ဖိုးကိုမှ ကိုယ်စားမပြု' ဆိုတဲ့ အဓိပ္ပာယ်အတွက် အသုံးပြုတာပါ။

## ၈.၃ Exceptions

ပုံမှန်အားဖြင့်တော့ ဖန်ရှင်တစ်ခုဟာ သူလုပ်ဆောင်ပေးရမဲ့ ကိစ္စကို ပြီးမြောက် အောင်မြင်အောင် ဆောင် ရွက်ပေးရမှုပါ။ ဒါပေမဲ့ ပုံမှန်မဟုတ်တဲ့ (သို့) မမျှော်လင့်ထားတဲ့ အခြေအနေ တစ်စုံတစ်ရာကြောင့် ဖန်ရှင် တစ်ခုဟာ သူလုပ်ဆောင်ပေးရမဲ့ ကိစ္စကို ပြီးအောင်ဆက်လုပ်ပေးလို့ မရနိုင်တော့တာ ဖြစ်နိုင်ပါတယ်။ 'ပုံမှန် မဟုတ်တဲ့ (သို့) မမျှော်လင့်ထားတဲ့ အခြေအနေ' ဆိုတာ ဘယ်လိုမျိုးပါလဲ။ အီးမေးလိုပို့ send\_email ဖန်ရှင် ခေါ်တယ်ဆိုပါစို့။ အင်တာနက် ကွန်နက်ရှင် ရှိပါမယ်။ မရှိရင် send\_email က အီးမေးလိုပို့လို မရနိုင်ပါဘူး။ date အော့သိဂျက်တစ်ခု ဖန်တီးမယ်ဆိုပါစို့။ လနဲ့ ရက် မဖြစ်နိုင်တဲ့ ကဏ္နာတွေ ထည့်ပေးရင် အော့သိဂျက် ဖန်တီးလိုမျှရနိုင်ပါဘူး (သို့) ဖန်တီးမပေးသင့်ဘူး။

```
>>> date(2024, 13, 32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: month must be in 1..12
>>> date(2024, 12, 32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: day is out of range for month
```

ဖိုင်တစ်ခုကို ဖွင့်တဲ့အခါ ဖိုင်နဲ့မည် (သို့) path လမ်းကြောင်းမှားနေရင် ဖွင့်လိုမရပါဘူး။ ဖိုင်သိမ်းတဲ့အခါ မှာလည်း ခွင့်မပြုထားတဲ့ နောက်မှ သိမ်းလိုမရပါဘူး။ ဖျက်ပစ်မယ်ဆိုလည်း ခွင့်မပြုတဲ့ဖိုင်ကို ဖျက်လိုမရဘူး။ ဖိုင်စနစ်နဲ့ သက်ဆိုင်တဲ့ ဖန်ရှင်တွေကို အခန်း (၁၀) မှာ လေ့လာကြတဲ့အခါ ဒီလိုမျိုး ပြဿနာတွေ ကြော်တွေ့ရှုမှုပါ။

```
>>> open('abc.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'abc.txt'
```

အခုံဖြေခဲ့တာတွေဟာ 'ပုံမှန်မဟုတ်တဲ့ (သို့) မမျှော်လင့်ထားတဲ့ အခြေအနေ' ဥပမာတချို့သာ ဖြစ်ပါတယ်။ နောက်ပိုင်းမှာ အခြားဟာတွေ ထပ်တွေ့ရမှုပါ။ ပရိုကရမ်တစ်ခု run နေတဲ့အချိန် ဒီလို အခြေအနေတွေ ဖြစ်လာခဲ့ရင် ပရိုကရမ်က ရပ်သွားပြီး ဆက်လက် အလုပ် မလုပ်ပေးနိုင်တော့တာမျိုး မဖြစ်သင့်ပါဘူး။ ဒီလိုမဖြစ်အောင် ကိုင်တွေယ်ထိန်းကြောင်း ပေးဖို့အတွက် ခေတ်မီ programming language တွေ အားလုံးလိုလိုမှ ရှိရှင်းတဲ့ နည်းစက်တစ်ခု ထည့်သွင်းပေးထားပါတယ်။ အဲဒီ နည်းစနစ် ကတော့ exception-handling ပဲ ဖြစ်ပါတယ်။

### `raise`-ing Exceptions

Exception-handling မှာ အပိုင်း နှစ်ပိုင်းပါဝင်တယ်။ ပထမတစ်ခုက တစ်ခုခဲ့ ပြဿနာဖြစ်နေပြီ ဆိုတာ အသိပေးတဲ့ အပိုင်း။ ဖန်ရှင်တစ်ခုဟာ ပုံမှန်မဟုတ်တဲ့ ပြဿနာကြောင့် သူတာဝန်ကို ပြီးမြောက် မှန်ကန်အောင် မလုပ်ပေးနိုင်တော့တဲ့အခါ ဖန်ရှင်ခေါ်တဲ့သူ (သို့) ခေါ်ထားတဲ့နေရာ ကို အသိပေးနိုင်ဖို့ လို

ပါတယ်။ ဒါမှာလည်း ဖြစ်တဲ့ ပြဿနာပေါ် မူတည်ပြီး ဘာလုပ်ရမလဲ ဆုံးဖြတ်လို့ ရမယ်။ (အီးမေးလ်ပိုတာ အင်တာနက်မရရင် ခဏနောင့်ပါလို့ အသိပေးတာမျိုး၊ ရက်စွဲထည့်တာ မှားရင် ပြန်ထည့်ခိုင်းတာမျိုး၊ ဖိုင်မရှိလို့ ဖွင့်မရရင် နောက်တစ်နာရီကြာမှ ပြန်စမ်းဖို့ တိုင်မာပေးတာမျိုး ... စသည်ဖြင့်ပေါ့)။

```
# File: print_n_times.py
def print_n_times(txt, n):
    if not (isinstance(n, int) and n > 0):
        raise ValueError('Positive integer expected')
    for i in range(n):
        print(txt)
```

ဒီဖန်ရှင်က စာသားတစ်ခုကို သတ်မှတ်ထားတဲ့ အကြိမ်အရေအတွက် ပြည့်အောင် ပရင်ထုတ်ပေးမှာပါ။ အကြိမ်အရေအတွက်က အပေါင်း ကိုန်းပြည့်ကဏ္ဍး ဖြစ်သင့်ပါတယ်။ မဟုတ်ဘူးဆိုရင် ဖုန်ရှင်ခေါ်တဲ့အခါ ထည့်ပေးတဲ့ အကြိမ်အရေအတွက် အကြောင်းတစ်ခုအကြောင့် မှားနေတာပဲ ဖြစ်ရမယ်။ ဒီလိုဖြစ်လာခဲ့ရင် တစ်ခုခဲ့ မှားနေပြီဆိုတာ အသိပေးဖို့အတွက် raise စတိတ်မန်ကို အသုံးပြုခိုင်တယ်။ isinstance ဖုန်ရှင်က ဖေရီရောဘဲလဲရဲ့ တိုက်ပိုက် စစ်ပေးတာပါ။ n ဟာ int ဖြစ်ရင် isinstance(n, int) ၏ True ရမှာပါ။ အပေါင်းကိုန်းပြည့် မဟုတ်ခဲ့ရင်

```
raise ValueError('Positive integer expected')
```

နဲ့ ပြဿနာကို အသိပေးပါတယ်။ ဒါကို exception ကို raise လုပ်တယ်လို့ ခေါ်တယ်။ ValueError ကတော့ exception (ပုံမှန်မဟုတ်တဲ့/မမျှော်လင့်ထားတဲ့ အခြေအနေ/ပြဿနာကို ဆိုလို့) ကို ဖော်ပြတဲ့ အော့သံဂျက်ပါ။ ValueError အပြင် ArithmeticError, FileNotFoundError စသည်ဖြင့် ဖြစ်တဲ့ exception ပေါ်မှာလည်ပြီး သင့်တော်တဲ့ အော့သံဂျက်ကို raise လုပ်နိုင်ပါတယ်။

print\_n\_times.py ကို run ကြည့်ပါ။ ဖိုင်အောက်ပိုင်းက ဒုတိယ ဖန်ရှင်ကောလ်မှာ exception တက်မှာပါ။

```
print_n_times('Hello', 10)
print_n_times('Hi', 0)
print_n_times('Hola', 3)
```

အခုလို မက်ဆေ့ချုံ တွေ့ပြုပြီး ပရိုကရမ် ဆက်အလုပ် မလုပ်တော့ဘဲ ရပ်သွားမှာပါ။

```
...
Hello
Hello
Traceback (most recent call last):
  File ".../ch08/print_n_times.py", line 10, in <module>
    print_n_times('Hi', 0)
  File ".../ch08/print_n_times.py", line 4, in print_n_times
    raise ValueError('Positive integer expected')
ValueError: Positive integer expected
```

တဗိုယဖန်ရှင် ဆက်မလုပ်တဲ့အတွက် Hola သုံးခါ မပါတာ သတိပြုပါ။

## Handling Exceptions

Exception-handling ရဲ့ ဒုတိယပိုင်းကတော့ handle (ကိုင်တွယ် ထိန်းကျောင်းတာကို ဆိုလို) လုပ်တဲ့ ကိစ္စဖြစ်ပါတယ်။ `raise` လုပ်လိုက်တဲ့ exception ကို handle မလုပ်ရင် ပရီဂရမ်ဟာ ဆက်အလုပ် မလုပ်နိုင်တော့ဘဲ ရပ်သွားမှုပါ။ Handle လုပ်ဖို့ `try` နဲ့ `except` ကို သံဃာပါတယ်။

ဖန်ရှင်တစ်ခုက `raise` လုပ်လိုက်တဲ့ exception ကို handle လုပ်ဖို့ရည်ရွယ်ချက်ရှိရင် အဲဒီဖန်ရှင်ကို `try` ဘလောက်ထဲမှာ ခေါ်ပါမယ်။ Exception ဖြစ်ခဲ့ရင် ဘယ်ထိ handle လုပ်ချင်လဲ။ ဒါ အပိုင်းကိုတော့ `except` ဘလောက်ထဲမှာ ရေးရပါမယ်။

```
try:
    print_n_times('Hi', 0)
except ValueError as err:
    print(f'Error: {err}')
```

`ValueError` ကတော့ handle လုပ်မဲ့ exception ရဲ့ တိုက်ပ်ပါ။ `ValueError` exception ကိုပဲ handle လုပ်မယ်လို့ ဆိုလိုတာ။ အဲေားဟာတွေဆိုရင် handle မလုပ်ဘူးပေါ့။ `FileNotFoundException` အတွက်ဆိုရင် `except FileNotFoundException as err:` ဖြစ်ပါမယ်။ `err` က exception ကို ကိုယ်စားပြုတဲ့ ဖော်ရေားလဲ (အဲေား နံမည်ဖြစ်လိုက်တယ်)။ `Exception` ဖြစ်ခဲ့ရင် (သို့) `raise` လုပ်ခဲ့ရင် အဲဒီ exception နဲ့ သက်ဆိုင်တဲ့ အချက်အလက်တွေကို `err` ကနေတစ်ဆင့် ရယူနိုင်ပါတယ်။ သုံးဖို့မလိုရင်တော့ `as err` မပါဘဲ `except ValueError:` နဲ့ ရတယ်။

`try` ဘလောက်ထဲမှာ exception ဖြစ်ခဲ့ရင် ဖြစ်တဲ့နေရာကနေ သက်ဆိုင်တဲ့ `except` ဘလောက်ဆိုကို ‘ချက်ချင်း’ ရောက်သွားမှုပါ။ မဖြစ်ခဲ့ရင်တော့ `try` ဘလောက် ပြီးတဲ့အထိ လုပ်ဆောင်ပြီး `except` ဘလောက်ကို လစ်လျှော့၍ သွားမှုပါ။ အခုလို စမ်းသပ်ကြည့်ပါ

```
try:
    print_n_times('Hi', 0)
    print('Done printing') # exception ဖြစ်ရင် ဒီလိုင်းကို ရောက်မလာဘူး
except ValueError as err:
    print(f'Error: {err}')

print('Program exits')
```

Output:

```
Error: Positive integer expected
Finish program
```

Exception `raise` လုပ်ရင် ကွန်းမာန်ရေးထားတဲ့ လိုင်းကို မလုပ်ပါဘူး။ `except` ဘလောက်နဲ့ အောက်ဆုံး `print` လုပ်တဲ့အထိ ဆက်အလုပ်လုပ်သွားတယ်။ 2 ထည့်ပြီး စမ်းကြည့်ရင် exception မဖြစ်ဘူး။ `try` ဘလောက် ပြီးတဲ့ထိ ပုံမှန်အတိုင်း လုပ်ဆောင်တယ်။ `Exception` မဖြစ်တော့ `except` ဘလောက်အလုပ် မလုပ်ဘဲ ကျော်သွားပါတယ်။

Exception-handling ကို အသုံးပြုပြီး အင်တီဂျာ တစ်ခု input ထည့်ခိုင်းတဲ့ ဖန်ရှင်ကို အခုလိုရေးဆိုင်ပါတယ်။

```
# File: read_int.py
def read_int(prompt):
```

```

while True:
    try:
        return int(input(prompt))
    except ValueError as err:
        print('Error: Non-integer data!')

# test
num = read_int('Enter number: ')
print(num)

```

အင်တိဂျာမဟုတ်တဲ့ တန်ဖိုး ထည့်ပေးရင် `int(input(prompt))` မှာ `ValueError` exception ဖြစ်ပြီး handle လုပ်တဲ့ ဘဏေက်ကို ရောက်သွားမှုပါ။ ဒီတော့ ဖုန်ရှင်က `return` မဖြစ်ဘူး။ `while` loop နောက်တစ်ကြိမ် ထပ်ကျော်ပါတယ်။ အင်တိဂျာ ပြောင်းလိုက်မဲ့ တန်ဖိုး ထည့်ပေးမှပဲ exception မဖြစ်ဘဲ `return` လုပ်ရင် ဖုန်ရှင်နော်တဲ့ဆီကို ချက်ချင်း ပြန်ရောက်သွားတဲ့ အတွက် `loop` ကနေလည်း ထွက်သွားစေတယ်။

လိုက်ဘရီ ဖုန်ရှင်တွေ အသုံးပြုတာပဲဖြစ်ဖြစ်၊ ကိုယ်ပိုင် ဖုန်ရှင် သတ်မှတ်တာပဲ ဖြစ်ဖြစ် exception တော့ exception-handling အခြေခံအဆင့် နားလည်ထားဖို့ လိုအပ်တယ်။ Exception-handling နဲ့ ပါတ်သက်ပြီး အခန်း (၁၀) မှာ ဒီထက် ကျယ်ကျယ်ပြန့်ပြန့် ဆက်လက် လေ့လာရအုံမှုပါ။ အခုတော့ ဒီလောက်နဲ့ ခဏရုပ်ထားပြီး လက်တွေ့နဲ့ ပိုနိုးစပ်တဲ့ အသုံးချေပမာတချို့ကို ဆက်ကြည့်ရအောင်။

## ၈.၄ ဖုန်ရှင်နှင့် ပရီဂရမ်ဒိုဝင်း ဥပမာ (၁) “Insurance Premium”<sup>၆၁</sup>

နှစ် နှစ်ဆယ် သက်တမ်းကာလ \$500,000 အသက်အာမခံထားရင် ကျေန်းမာတဲ့ အသက် (၃၀) အရွယ် အမျိုးသမီး တစ်ယောက်အတွက် နှစ်စဉ်ပျော်မျှ အာမခံကြား (premium) \$229 ဒေါ်လာ ကုန်ကျ တယ်။ ရွယ်တူ အမျိုးသမီးတစ်ယောက် ဆိုရင်တွေ \$373 ဒေါ်လာပါ။ ဒါကယောကုယျ သဘောကိုပြေ တာ။ အပိုင်းမှာ အသက်အာမခံထားရင် သက်တမ်းကာလ၊ အကျိုးခံစားနိုင်မည့် ငွေပမာဏ (coverage)၊ ကျေန်းမာရေး၊ အသက်အရွယ် စတဲ့ အချက်တွေပေါ် မူတည်ပြီး တစ်ညီးချင်းအတွက် အာမခံကြား သီးသန့် တွက်ချက်တာပါ။

အာမခံသက်တမ်း နှစ်ဆယ်နှစ်အတွက်ပဲ စဉ်းစားပါမယ်။ အသက် (၃၀) အတွက် premium လို ပြောပေမဲ့ တကယ်တမ်းက အသက် (၁၈) နှစ် ကနေ (၃၀) (အပါအဝင်) အတွက် premium ကို ဆိုလိုတာဖြစ်တယ်။ အသက် (၁၈) နှစ် ကနေ (၅၀)၊ (၅၁) နှစ် ကနေ (၆၀) ကြား (အပါအဝင်) ကို ဆိုလိုတာဖြစ်တယ်။ (၁၈) နှစ်အောက် နဲ့ (၆၀) အထက်ဆိုရင်တွေ အကျိုးမာဝ်ပါဘူး (အာမခံ ထားလို့ မရဘူး ယူဆပါ)။ ယေား (၈.၄) နှင့် (၈.၄) တွင် ကြည့်ပါ။

Coverage Amount	Age 30	Age 40	Age 50	Age 60
\$250,000	\$142	\$193	\$392	\$989
\$500,000	\$205	\$307	\$685	\$1,781
\$1 million	\$325	\$526	\$1,227	\$3,375
\$2 million	\$593	\$984	\$2,388	\$6,758

တေဘဲလ် ၈.၁ နှစ်နှစ်ဆယ် အသက်အာမခံကြား (၁)

Coverage Amount	Age 30	Age 40	Age 50	Age 60
\$250,000	\$162	\$224	\$499	\$1,375
\$500,000	\$251	\$360	\$891	\$2,567
\$1 million	\$408	\$628	\$1,681	\$4,952
\$2 million	\$749	\$1,190	\$3,267	\$9,660

### တေဘာင် ၈.၂ နှစ်နှစ်ဆယ် အသက်အာမခံပြေား (ကျော်)

မွေးသက္ကရာဇ်၊ အကျိုးခံစားလိုသည့် ပမာဏ (coverage amount)၊ ကျော်/မ အလိုက် တစ်နှစ် ပူမးမှု ပရီမီယံကြေး တွက်ပေးတဲ့ ပရိုကရမ်တစ်ခု အာမခံ အေးဂျင့်တစ်ယောက် အတွက် ရေးပေးရမယ် ဆိုပါစို့။ အေးဂျင့်က သူလိုချင်တဲ့ ပုံစံအတွက် ဥပမာတချို့ကို အခုလို ပြပါတယ်။

dob? 19-12-1980

Enter gender F (Female)/M (Male): M

Choose 1/2/3/4 for one of the coverage options:

1. \$250,000
2. \$500,000
3. \$1,000,000
4. \$2,000,000

3

Premium per year: 1681.00

dob? Feb-15-1990

Enter gender F (Female)/M (Male): F

Choose 1/2/3/4 for one of the coverage options:

1. \$250,000
2. \$500,000
3. \$1,000,000
4. \$2,000,000

4

Premium per year: 984.00

### လုပ်ငန်းသဘောနဲ့ သက်ဆိုင်တဲ့ ဖန်ရှင်များ

ပရိုကရမ် တစ်ခု ဒီဇိုင်းလုပ် ရေးသားပုံအဆင့်ဆင့်ကို ဆက်လက်ဖော်ပြပါမယ်။ လုပ်ငန်းသဘောနဲ့ သက်ဆိုင် တဲ့ အပိုင်းနဲ့ input/output အပိုင်းနဲ့ ပြီးရေးလေ့ရှိတယ်။ (ကိုဘုံးကနေ ထည့်ပေးတာကို ဖတ်တာ၊ စခင်မှာ ရလဒ်ကို ထုတ်ပြတာ စတာတွေကို input/output အပိုင်းမှာ ပါဝင်တယ်)။

လုပ်ငန်းသဘောနဲ့ ဆိုင်တဲ့ အပိုင်းကို အရင်ကြည့်ရအောင်။ Premium ကြေး သေားနှစ်ခုပါ ဒေတာ တွေကို သိမ်းထားဖို့ လိုတယ်။ Row, column တွေနဲ့ တေဘာင်ပုံစံမျိုး စီစဉ်ထားချင်ရင် nested list သုံးလို့ရပါတယ်။

```
# File: insurance_prem.py
```

```
from decimal import *
```

```
# Premium for male
```

```

MALE_PREM = [[Decimal('162.00'), Decimal('224.00'), # 1st row, $250,000
               Decimal('499.00'), Decimal('1375.00')],
              [Decimal('251.00'), Decimal('360.00'), # 2nd row, $500,000
               Decimal('891.00'), Decimal('2567.00')],
              [Decimal('408.00'), Decimal('628.00'),
               Decimal('1681.00'), Decimal('4952.00')],
              [Decimal('749.00'), Decimal('1190.00'),
               Decimal('3267.00'), Decimal('9660.00')]]

# Premium for female

FEMALE_PREM = [[Decimal('142.00'), Decimal('193.00'),
                  Decimal('392.00'), Decimal('989.00')],
                 [Decimal('205.00'), Decimal('307.00'),
                  Decimal('685.00'), Decimal('1781.00')],
                 [Decimal('325.00'), Decimal('526.00'),
                  Decimal('1227.00'), Decimal('3375.00')],
                 [Decimal('593.00'), Decimal('984.00'),
                  Decimal('2388.00'), Decimal('6758.00')]]

```

ဒီထဲကနေ လိုအပ်တဲ့ premium ကြေးကို ကျေးမှု၊ အသက်နဲ့ coverage ပေါ်မှတည်ပြီး ထုတ်ယူရမှာ ပါ။ ကျေးမှု (၄၅) နှစ်၊ coverage (၅၀၀,၀၀၀) အတွက် premium ကြေးကို MALE\_PREM[1][2] နဲ့ ယူရမှာပါ။

Premium ကြေးကို ဖန်ရှင်တစ်ခုနဲ့ အခုလိုမျိုး ယူလိုရသင့်တယ်။ အသက် အရွယ်၊ ကျေးမှု၊ coverage ပမာဏ ထည့်ပေးရမယ်။

```

retrieve_prem(45, 'M', Decimal('500_000.00'))      # should get 891.00
retrieve_prem(35, 'F', Decimal('1_000_000.00'))    # should get 526.00

```

ဒီကိစ္စအတွက် ဖန်ရှင် ဘယ်လိုရေးထားလဲ ကြည့်ရအောင်

```

# File: insurance_prem.py

COVERAGES = [Decimal("250_000.00"), Decimal("500_000.00"),
              Decimal("1_000_000.00"), Decimal("2_000_000.00")]

FEMALE = 'F'
MALE = 'M'

def retrieve_prem(age, gender, coverage):
    if not (18 <= age <= 60):
        raise ValueError(f"Age of {age} yrs not applicable!")

    age_band = None
    if age <= 30:
        age_band = 0
    elif age <= 40:
        age_band = 1
    elif age <= 50:

```

```

age_band = 2
elif age <= 60:
    age_band = 3

try:
    coverage_idx = COVERAGES.index(coverage)
except ValueError:
    raise ValueError(f'No such coverage amount, {str(coverage)}!')

if gender == FEMALE:
    return FEMALE_PREM[coverage_idx][age_band]
elif gender == MALE:
    return MALE_PREM[coverage_idx][age_band]

```

လိုအပ်တဲ့ constant တခါး၊ ကြော်ထားတယ်။ COVERAGES ၏ coverage ပမာဏတော် ပါတဲ့ list ဖြစ်တယ်။ ဘာအတွက်လဲ ခဏနေ တွေ့ရပါမယ်။ age, gender နဲ့ coverage က လိုအပ်တဲ့ ဖန်ရှင်ပါ ရာမိတာတွေပါ။ ဖန်ရှင် တစ်ခုဟာ လက်မခံနိုင်တဲ့ သို့မဟုတ် မဖြစ်သင့်တဲ့ ပါရာမိတာ တန်ဖိုးတွေအတွက် အကြောင်းပေးဖို့ မဖြစ်နိုင်ပါဘူး။ ဖန်ရှင်က သူလုပ်ဆောင်ရမဲ့ ကိစ္စကို အကြောင်းတစ်ခုခဲ့ကြောင့် မလုပ် ပေးနိုင်တဲ့အခါ exception raise လုပ်လေ့ရှိတယ်ဆိုတာ ဒီမတိုင်ခင် စက်ရှင်မှာ တွေ့ခဲ့ရတယ်။ အခါ ဖန်ရှင်ကလည်း အသက်အရွယ် လိုအပ်ချက် မကိုက်ညီရင် exception raise လုပ်တယ်။ Coverage ပမာဏ လေးချိုးထဲမှာ မပါဝင်ရင်လည်း raise လုပ်ထားပါတယ်။ COVERAGES.index(coverage) က ValueError ဖြစ်ရင် coverage ၏ list ထဲမှာ မရှိလို့ မှားနေလိုပဲ။ ဒီလိုဖြစ်ခဲ့ရင် except ဘလောက် ကလည်း ValueError ကိုပဲ raise လုပ်တယ်။ ဖြစ်တဲ့ပြဿနာကို တိတိကျကျ ဖော်ပြတဲ့မက်ဆွဲချုပ် ဖြစ်ဖို့ဟာ exception-handling အတွက် အရေးကြီးပါတယ်။

```
| ValueError('Something went wrong!')
```

ဆိုတာထက်

```
| ValueError(f'No such coverage amount, {str(coverage)}!')
```

က ပြဿနာရဲ့ အရင်းအမြစ်ကို နားလည်ဖို့ အများကြီး အထောက်အကူ ပိုဖြစ်ပါတယ်။

age\_band အတွက် သီးသန်ဖန်ရှင်တစ်ခု ခဲ့ထုတ်လိုက်ရင် ပိုကောင်းနိုင်တယ်။ age ကနေ မှန်ကန် တဲ့ ကော်လံ index ထုတ်ပေးကိစ္စ တစ်ခုကိုပဲ သီးသန်လုပ်ဆောင်ပေးတာပေါ့။

```

def age_to_age_band(age):
    if not (18 <= age <= 60):
        raise ValueError(f"Age of {age} yrs not applicable!")
    if age <= 30:
        return 0
    elif age <= 40:
        return 1
    elif age <= 50:
        return 2
    elif age < 60:
        return 3

```

ဒီဖန်ရှင်က age နဲ့ သက်ခိုင်တာကို အမိကလုပ်တာဆိုတော့ age က ဘောင်ဝင်မဝင်လည်း ဒီမှာပဲ ဆုံးဖြတ်တာ ပိုသင့်တော်မယ်လို့ ယူဆနိုင်တယ်။ ဒါကြောင့် ဒီဖန်ရှင်မှာပဲ စစ်ပြီး exception raise ထားတယ်။ ခုနက retrieve\_prem ဖန်ရှင်က ဒီလိုဖြစ်လာမယ်

```
def retrieve_prem(age, gender, coverage):
    try:
        age_band = age_to_age_band(age)
    except ValueError as age_err:
        raise age_err
    try:
        coverage_idx = COVERAGES.index(coverage)
    except ValueError:
        raise ValueError(f'No such coverage amount, {str(coverage)}!')
    if gender == FEMALE:
        return FEMALE_PREM[coverage_idx][age_band]
    elif gender == MALE:
        return MALE_PREM[coverage_idx][age_band]
```

age နဲ့ဆိုင်တဲ့ exception ဖြစ်ခဲ့ရင် ဒေါ်ဒီ exception ကိုပဲ ပြန် raise လုပ်လိုက်တယ် (raise age\_err)။ Exception ဖွင့်မဲ့ input တွေရော့၊ မဖြစ်ဘဲ အဆင်ပြေမဲ့ input တွေနဲ့ပါ မှန်/မမှန် စိစစ်ကြည့်ပါ။ အောက်ပါအတိုင်း တစ်ကြောင်းချင်း စမ်းကြည့်ပါ။

```
print(retrieve_prem(80, 'F', Decimal('1_000_000.00'))) # age error
print(retrieve_prem(50, 'F', Decimal('3_000_000.00'))) # coverage error
print(retrieve_prem(50, 'F', Decimal('1_000_000.00'))) # 1227.00
print(retrieve_prem(50, 'K', Decimal('1_000_000.00'))) # None (Why?)
```

နောက်ဆုံးတစ်ကြောင်းက None ဖြစ်နေတာ တွေ့ရတယ်။ ဘုံးကြောင့်လဲ အကြောင်းအရှင်းကို နောက်ပိုင်းမှာ သိရမှာပါ။

### Input/Output အပိုင်း

Input သုံးခုထည့်ပေးရမယ်။ မွေးသက္ကရာဇ် (date of birth) , coverage amount နဲ့ ကျား/မ (gender) တို့ဖြစ်တယ်။ အသုံးပြုသူအနေနဲ့ အဆင်ပြေဆုံး၊ အလွယ်ကူဆုံးဖြစ်အောင် အမှားအယွင်းနည်းနှင့်သမျှနည်းအောင် စဉ်းစားသင့်တယ်။

မွေးသက္ကရာဇ်ကို ကြည့်ရအောင်။ ရက်စွဲကို နိုင်ငံနဲ့ နေရာအော် ပေါ်မှုတည်ပြီး ဖော့မတ်အမျိုးမျိုးနဲ့ ရေးကြတယ်။ 04/05/2020, 04-05-2020, Apr-4-2020 စသည်ဖြင့်။ ခုနှစ်ကို ဂဏန်း နှစ်လုံးပဲ ရေးတာလည်း ရှိတယ်။ ရက်စွဲလကို ရှေ့မှာ သုည်ပါဘဲလည်း ရေးတယ်။ ဥပမာ 4/5/2020, 4/5/20, Apr-4-2020 ။ ဖော့မတ်တစ်မျိုးကိုပဲ သတ်သတ်မှတ်မှတ် သုံးသင့်တာ ဖြစ်ပေမဲ့ ဆော့ဖွဲ့အပ်သူက ဖော့မတ် သုံးမျိုးနဲ့ ထည့်လို့ရအောင် လုပ်ပေးဖို့ တောင်းဆိုထားတယ်။

01-01-2024

01/01/2024

Jan-1-2024

မွေးသက္ကရာဇ်ကို ဒီဖော့မတ်သုံးမျိုးနဲ့ ပရိုကရမ်က လက်ခံရပါမယ်။ input ဖန်ရှင်က ကီးဘုဒ်ကထည့်ပေး

တာကို string အနေနဲ့ ပြန်ပေးပါတယ်။

မွေးသက္ကရာဇ်ကနေ အသက်ကို တွက်ယူရမှပါ။ ဒီအတွက် ဖန်ရှင် (ဥပမာ calc\_age) သတ်မှတ် နိုင်တယ်။ နေ့ရက်၊ အချိန်နဲ့ သက်ဆိုင်တဲ့ အတွက်အချက်တွေ အတွက် စာသားကို အသုံးမပြုသင့်ဘူး။ date, datetime စတာတွေ သုံးသင့်တယ်။ ဒါကြောင့် စာသားကနေ မွေးသက္ကရာဇ်ကို ဖော်ပြတဲ့ date (သို့) datetime အော့သုံးဂျက် ပြောင်းဖို့ လိပါမယ်။ '01-01-2024' ကနေ 1, 1, 2024 ကတော်းတွေ ရအောင် စာသားကို တစ်ဖြတ်ချင်း ဖြတ်ပြီး ပြောင်းမယ်။

```
>>> egstr = '123abc456abc789'
>>> egstr.split('abc')
['123', '456', '789']
>>> dt1 = '01-01-2024'
>>> parts1 = dt1.split('-')
>>> parts1
['01', '01', '2024']
>>> dt2 = '01/01/2024'
>>> parts2 = dt2.split('/')
>>> parts2
['01', '01', '2024']
>>> dt3 = 'Feb-02-2024'
>>> parts3 = dt3.split('-')
>>> parts3
['Feb', '02', '2024']
```

split ဖန်ရှင်က string တစ်ခုကို အပိုင်းတွေ ပိုင်းပေးတာပါ။ အပိုင်းတွေ အားလုံးကို list အနေနဲ့ ပြန်ပေးတယ်။ ဘယ် ကာရက်တာ (သို့) string နဲ့ မြားထားရင် ပိုင်းဖြတ်ချင်လဲ ထည့်ပေးလို့ရတယ်။ အပေါ်မှာ 'abc', '-', '/' စတာတွေနဲ့ ပိုင်းဖြတ်ထားတယ်။

```
>>> dt1 = '01-01-2024'
>>> parts1 = dt1.split('-')
>>> date(int(parts1[2]), int(parts1[1]), int(parts1[0]))
datetime.date(2024, 1, 1)
```

'Feb-02-2024' ဖော်မတ်က နည်းနည်း ပို့ခြုပ်မယ်။ လက် 'Jan', 'Feb' စသည်ဖြင့် စာသား ဖြစ်နေတယ်။ အခုလို dictionary တစ်ခု ရှိရှင် လန်မည်ကနေ သက်ဆိုင်တဲ့ ကတော်းကို အလွယ်တကူ ရှိနိုင်ပါတယ်

```
>>> mths = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4,
...           'May': 5, 'Jun': 6, 'Jul': 7, 'Aug': 8,
...           'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}
>>> mths['Jan']
1
>>> mths['Dec']
12
```

ဒီလိုဆိုရင် date အော့သုံးဂျက် ရဖို့အတွက်လည်း သိပ်မခက်တော့ဘူး။ ဥပမာ

```
>>> dt4 = 'Dec-26-2024'
```

```
>>> parts4 = dt4.split('-')
>>> date(int(parts4[2]), mths[parts4[0]], int(parts4[1]))
datetime.date(2024, 12, 26)
```

နေ့ကြောင်း ဖော်မတ် သုံးမျိုးနဲ့ လက်ခံနှင့်တဲ့ ဖန်ရှင်ကို ရိုးရိုးရှင်းရှင်းလေးပဲ အခုလို

```
# File: insurance_prem.py
from datetime import *

MONTHS = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4,
          'May': 5, 'Jun': 6, 'Jul': 7, 'Aug': 8,
          'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}

def parse_date_str(dtstr):
    parts = dtstr.replace('/', '-').split('-')
    if parts[0].isdigit():
        return date(int(parts[2]), int(parts[1]), int(parts[0]))
    else:
        return date(int(parts[2]), months[int(parts[0])], int(parts[1]))
```

သတ်မှတ်ပါမယ်။ replace ဖန်ရှင်သုံးပြီး / ကို - နဲ့ အစားထိုးလိုက်တယ်။ ဒါဆုံးရင် ဖော့မတ် နှစ်ခုပဲ စဉ်းစားဖို့လိုတော့မယ် (10/01/2024 ကနေ 10-01-2024 ဖြစ်သွားမှာပါ)။ ပြီးတော့မှ split လုပ်တယ်။ ရဲခိုးအပိုင်း parts[0] ကတော်းဟုတ်လား isdigit နဲ့ စစ်ထားတယ်။ ဖော့မတ် အမှန်နဲ့ စမ်းကြည့်ရင် date အော့သုတေသနပေးပါတယ်

```
print(parse_date_str('12-03-1995'))  
print(parse_date_str('12/03/1995'))  
print(parse_date_str('Mar-12-1995'))
```

ဖော်မတ် မမှန်ရင်၊ ဒါမှုမဟုတ် အဲခြားအကြောင်းတစ်ခုခုကြောင့် date ပြောင်းလိုမရင် exception ဖြစ်တယ်။ Fab စာလုံးပေါင်း များရင် KeyError ဖြစ်တယ်

```
# Spelling error: Fab instead of Feb  
print(parse_date_str('Fab-12-1995'))
```

## Error Output:

Dictionary ထဲမှာ Fab ကို မရှိတဲ့အတွက် ဒီပြဿနာဖြစ်တာပါ။ မဖြစ်နိုင်တဲ့ ခုနှစ်၊ လ၊ ရက် တန်ဖိုး တွေဆိုရင် ValueError ဖြစ်တယ် (ဥပမာ `parse_date_str('30-02-1995')`)။

အခုဖော်ပြခဲ့တဲ့ နည်းအပြင် အခြားနည်းလမ်းတွေလည်း ရှိရမှာပါ။ `datetime` ကလပ်စူးမှာ နေရက် ကို စာသားကနေ `datetime` ပြောင်းပေးတဲ့ `strptime` ဖန်ရှင်းရှိတယ်။ သူကို အသံးပြုပြီး `parse_date_str` ကို ဘယ်လိုပေးလို့ ရမလဲ။ စဉ်းစားကြည့်ရအောင် ...

```
>>> dtstr1 = 'Jan-01-2024'
>>> datetime.strptime(dtstr1, "%b-%d-%Y")
datetime.datetime(2024, 1, 1, 0, 0)
>>> dtstr2 = '30-12-2024'
>>> datetime.strptime(dtstr2, "%d-%m-%Y")
datetime.datetime(2024, 12, 30, 0, 0)
>>> dtstr3 = '30/Dec/2024'
>>> datetime.strptime(dtstr3, "%d/%b/%Y")
datetime.datetime(2024, 12, 30, 0, 0)
```

ဒီဖန်ရှင်က စာသားနဲ့ ဖော်ပြထားတဲ့ အခိုင်နေ့ရက်ကို `datetime` အော့ဘ်ဂျက် ပြောင်းပေးဖို့ ဖော့မတ် ကုဒ် (format code) လိုပေါ်တဲ့ % နဲ့ စတဲ့ ကာရှင်တာတွေကို လက်ခံတယ်။ `%d` က ရက်၊ `%m` က လ ကို ကဏ်နဲ့ တစ်လုံး (သို့) နှစ်လုံးနဲ့ ဆုတ် အဓိပ္ပာယ် (ဥပမာ ကိုရောက်နေ့ကို 09 ဆိုမဟုတ် ၅၊ ငါးလပိုင်း ကို 05 ဆိုမဟုတ် ၅)။ `%Y` က ခုနှစ်ကို ကဏ်နဲ့ လေးလုံးနဲ့ ရေးတယ်လို့ ဆိုလိုတာပါ။ `%b` ကတော့ လရဲ့ နံမည်ကို Jan, Feb, Mar စသည်ဖြင့် အတိုကောက်ရေးတယ်လို့ ဆိုလိုတာယ်။ 'Jan-01-2024' ကို ပြောင်းမယ်ဆိုရင် သူရဲ့ဖော့မတ်နဲ့ ကိုက်ညီတဲ့ '%b-%d-%Y' ကို ဖန်ရှင်ခေါ်တဲ့အခါ ထည့်ပေးရပါမယ်။ '30/Dec/2024' ဆိုရင် '%d/%b/%Y'၊ '30-12-2024' အတွက် '%d-%m-%Y' ဖြစ်မှာပါ။ ဖော့မတ် ကုဒ် သုံးထားတဲ့ `parse_date_str2` ကို အခုလို သတ်မှတ်လိုရပါမယ်

```
formats = ['%d-%m-%Y', '%d/%m/%Y', '%b-%d-%Y']
def parse_date_str2(dtstr):
    for fmt in formats:
        try:
            return datetime.strptime(dtstr, fmt).date()
        except ValueError:
            pass
    # ဖော့မတ် သုံးမျိုးလုံးနဲ့ မကိုက်ညီလို့ ပြောင်းလိုမရရင် ဒီကိုရောက်လာမယ်
    raise ValueError(f"{dtstr} doesn't match any of acceptable formats")
```

`dtstr` ကို ဖော့မတ်တစ်ခုချင်း ပြောင်းကြည့်တယ်။ ပြောင်းလိုရရင် `return` ဖြစ်သွားမယ်။ ပြောင်းလိုမရရင် `strptime` က `ValueError` `raise` လုပ်တယ်။ ဖော့မတ်တစ်ခုနဲ့ ပြောင်းလိုမရရင် နောက်တစ်ခု နဲ့ရှိနိုင်ပါတယ်။ ဒါကြောင့် `ValueError` ဖြစ်ခဲ့ရင် `pass` ပဲ လုပ်ပေးလိုက်တယ် (ဘာမှုလုပ်စရာမလိုရင် `pass` စတိတ်မနဲ့ သုံးတာပါ)။ ဖော့မတ် သုံးမျိုးလုံးနဲ့ အဆင်မပြောင်တော့ တစ်ခုခု မှားနေလိုပဲ။ ဒါကြောင့် `return` မဖြစ်ဘဲ `for` loop ပြီးတဲ့ထိ ရောက်လာခဲ့ရင် `ValueError` exception ဖြစ်အောင် `raise` လုပ်ထားတာ။ `strptime` ပြန်ပေးတာက `datetime` ပါ။ `date` လိုချင်ရင် `date()` ဖန်ရှင် ဆက်ခေါ်လို့ ရတယ်။

```
| datetime.strptime(dtstr, fmt).date()
```

ပရီဂါရမ်တည်ဆောက်တဲ့အခါ ကိစ္စတစ်ခုကို ဖြော်ရှင်းလိုရတဲ့ နည်းလမ်းက တစ်ခုတည်းပဲ ဖြစ်လေ့မရှိတာကတော့ ထုံးစံပါပဲ။ ဖြစ်နိုင်တဲ့ ဖြော်ရှင်းနည်းတွေထားက အသင့်တော်ဆုံးတစ်ခုကို ရွှေ့ချယ် အသံးပြုရတာပါ။ နည်းလမ်းတစ်ခုက လက်ရှိမှာ အသင့်တော်ဆုံး ဖြစ်ပေါ့ နောင်တစ်ချိန်မှာ သူထက် ပိုကောင်း

တာ၊ ပိုသင့်တော်တာလည်း ရှိလာနိုင်ပါတယ်။

မွေးသက္ကရာဇ် ဖတ်တဲ့ ဖန်ရှင်နဲ့ အသက်တွက်တဲ့ ဖန်ရှင်ကို အခုလို သတ်မှတ်ပါမယ်

```
def read_date():
    while True:
        try:
            return parse_date_str(input('dob? '))
        except Exception as err:
            print("Incorrect date. Please enter the date again.")
            print('Error is probably: ' + str(err))

def age_today(dob):
    today = date.today()
    this_bd = dob.replace(year=today.year)
    if today - dob >= this_bd - dob:
        return today.year - dob.year
    else:
        return today.year - dob.year - 1
```

read\_date ၏ ဖော်မတ် မမှန်မချင်း ပြန်ထည့်ခိုင်းမှုပါ။

ဒါကတော့ coverage option အတွက် ဖန်ရှင် နဲ့ gender လက်ခံတဲ့ ဖန်ရှင်တွေပါ

```
def read_coverage():
    print("Enter 1/2/3/4 for one of the coverage options: ")
    print("1. $250,000")
    print("2. $500,000")
    print("3. $1,000,000")
    print("4. $2,000,000")
    while True:
        try:
            choice = int(input())
            if 1 <= choice <= 4:
                return COVERAGES[choice - 1]
            else:
                print('Incorrect choice! Accept only 1 to 4!')
        except ValueError as err:
            print('Incorrect choice! Accept only 1 to 4!')
```

တစ်ကနေ လေးထိပဲ လိုချင်တာ။ ကိန်းပြည့်မဟုတ်တာ (ဥပမာ 12yz, 12.3) ထည့်ရင် int() ပြောင်း  
တဲ့နေရာ exception ဖြစ်မယ်။ Handle အပိုင်းမှာ ဒီအတွက် အယ်ရာပြပေးတယ်။ အင်တီဂျာတော့  
ထည့်တာမှန်တယ်၊ ဒါပေမဲ့ တစ်နဲ့ လေးကြား မဟုတ်ရင်လည်း လက်ခံလို့ မဖြစ်ဘူး။ ဒီအတွက် if နဲ့ စစ်  
ပြီး အယ်ရာပြပေးတယ်။

```
def read_gender():
    while True:
        gender = input('Enter gender F (Female)/M (Male): ')
```

```

if gender in ['F', 'M']:
    return gender
else:
    print('Invalid gender! Accept only F or M!')

```

ကိုဘုဒ် input ဖတ်စွာ လိုအပ်တဲ့ ဖန်ရှင်အားလုံးလည်း ရှိပြီ။ နောက်ဆုံးတော့ ပရီဂရမ်တစ်ခုလုံး၏  
top level ဖန်ရှင်က အခုလိပ်ပါ။ ရှိခိုး ရှင်းရှင်းလေးပါပဲ။

```

def run_prem_program():
    while True:
        dob = read_date()
        gender = read_gender()
        coverage = read_coverage()

        try:
            print(retrieve_prem(age_today(dob),
                                gender,
                                coverage))
        except ValueError as dm_err:
            print(f'Domain error: {dm_err}')

        quit_or_cont = input('Press enter/return to continue.
                            ' To quit the program, enter quit.')
        if quit_or_cont == 'quit':
            break

```

retrieve\_prem မှာ exception ဖြစ်ရင် handle လုပ်တဲ့အပိုင်းက ဘာကြောင့် error ဖြစ်လဲ ပြေး  
ပါတယ်။ ထွက်ချင်ရင် quit ထည့်ပေးရမယ်။ Enter/return နှင့်ရင် နောက်တစ်ခါ input ပြန်တောင်း  
ပြီး ဆက်အလုပ်လုပ်နေမှာပါ။ ကိုယ်ပိုင် သတ်မှတ်တဲ့ ဖန်ရှင်အားလုံးကနေ ဖြစ်လိန့်တဲ့ exception  
အားလုံးကို သင့်တော်တဲ့နေရာမှာ handle လုပ်ထားတဲ့အတွက် ဘာပြဿနာမှ မဖြစ်နိုင်တော့ဘူးလို ယူဆ  
စရာ အကြောင်းရှိပါတယ်။ ဒါပေမဲ့ အားကိုယ် မမျှော်မှန်းနိုင်တဲ့ exception တော်လည်း ရှိနိုင်ပါဘေး  
တယ်။ (ဥပမာ ကွန်ပျိုးတာ operating system နဲ့ သက်ဆိုင်တဲ့ OSError မမေ့မှုရှိ မလုပ်လောက်ရင်  
MemoryError)။ ဒီလို exception တွေဖြစ်ရင် ပရီဂရမ် ပုံမှန်ဆက်အလုပ် လုပ်နိုင်ဖို့လည်း မဖြစ်နိုင်  
တာ များပါတယ်။ ပရီဂရမ်ကို ဆက် run လို မရတော့လောက်အောင် မဆိုးပေမဲ့ ကိုယ်မမျှော်လင့်ထားမိ  
တဲ့ exception ဖျိုးလည်း ဖြစ်နိုင်တာပဲ။ ဒီလိုအခါဖျိုးမှာ တစ်ခုခုတော့ ပြဿနာဖြစ်သွားတယ် ပရီဂရမ်  
ရေးတဲ့သူကို ဆက်သွယ်ပါ ဆိုတာမျိုး user ကို မက်ဆေ့ချုံ ကောင်းကောင်းမွန်မွန် ပြေးချင်တယ်ဆိုရင်  
အခုလို လုပ်လိုရပါတယ်။

```

def run_prem_program():
    while True:
        try:
            dob = read_date()
            gender = read_gender()
            coverage = read_coverage()

            try:

```

```

        print(retrieve_prem(age_today(dob),
                            gender,
                            coverage))
    except ValueError as dm_err:
        print(f'Domain error: {dm_err}')

    quit_or_cont = input(('Press enter/return to continue.'
                         ' To quit the program, enter quit.'))
    if quit_or_cont == 'quit':
        break
    except Exception as sys_err:
        print(f'Application/System Error: {sys_err}')
        print('Please kindly report to system owner.')

```

အပြင် try: ... except: က အခြား မမျှော်လင့်ထားတဲ့ exception အေးလုံးကို handle လုပ်ပေးပါတယ်။ except Exception ကတော့ သီးသန့် exception တစ်မျိုးအတွက်မဟုတ်ဘဲ ဘယ်အချို့အစားဖြစ်ဖြစ် handle လုပ်ချင်တဲ့အခါ သုံးရတာပါ။ ဒါနဲ့ ပါတ်သက်ပြီး Exception-handling အခန်းမှာ ဆက်လက်လေးလာကြပါမယ်။

နောက်ပိုင်းမှာတော့ retrieve\_prem မှာ ပထား try...except ဖြတ်လိုက်လိုရမယ် (သို့ ဖြတ်သင့်တယ်) ဆိုတာ နားလည်လာပါမယ်။

```

def retrieve_prem(age, gender, coverage):
    age_band = age_to_age_band(age)
    try:
        coverage_idx = COVERAGES.index(coverage)
    except ValueError:
        raise ValueError(f'No such coverage amount, {str(coverage)}!')
    if gender == FEMALE:
        return FEMALE_PREM[coverage_idx][age_band]
    elif gender == MALE:
        return MALE_PREM[coverage_idx][age_band]

```

age\_to\_age\_band မှာ ဖြစ်တဲ့ exception ကို handle မလုပ်ထားတော့ဘူး။ ဒါပေမဲ့ ပရိုဂရမ်တစ်ခုလုံးရဲ့ လုပ်ဆောင်တဲ့ ဘီးဘေးများကတော့ မပြောင်းလဲသွားပါဘူး။ နိုင်အတိုင်းပါပဲ။ ဘာ့ကြောင့်လဲ အပိုင်း (၂) မှာ ဆက်ကြည့်ပါမယ် ...

ပရိုဂရမ် အစအဆုံး လေးလာကြည့်ပါ စမ်းကြည့်ပါ ...

```

from decimal import *
from datetime import *

MALE_PREM = [[Decimal('162.00'), Decimal('224.00'), # 1st row, $250,000
              Decimal('499.00'), Decimal('1375.00')],
             [Decimal('251.00'), Decimal('360.00'), # 2nd row, $500,000
              Decimal('891.00'), Decimal('2567.00')],
             [Decimal('408.00'), Decimal('628.00'),
              Decimal('1681.00'), Decimal('4952.00')],
```

```

        [Decimal('749.00'), Decimal('1190.00'),
         Decimal('3267.00'), Decimal('9660.00')]]
# Premium for female
FEMALE_PREM = [[Decimal('142.00'), Decimal('193.00'),
                 Decimal('392.00'), Decimal('989.00')],
                [Decimal('205.00'), Decimal('307.00'),
                 Decimal('685.00'), Decimal('1781.00')],
                [Decimal('325.00'), Decimal('526.00'),
                 Decimal('1227.00'), Decimal('3375.00')],
                [Decimal('593.00'), Decimal('984.00'),
                 Decimal('2388.00'), Decimal('6758.00')]]


COVERAGES = [Decimal("250_000.00"), Decimal("500_000.00"),
             Decimal("1_000_000.00"), Decimal("2_000_000.00")]


FEMALE = 'F'
MALE = 'M'

MONTHS = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4,
          'May': 5, 'Jun': 6, 'Jul': 7, 'Aug': 8,
          'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}

def retrieve_prem(age, gender, coverage):
    try:
        age_band = age_to_age_band(age)
    except ValueError as age_err:
        raise age_err
    try:
        coverage_idx = COVERAGES.index(coverage)
    except ValueError:
        raise ValueError(f'No such coverage amount, {str(coverage)}!')
    if gender == FEMALE:
        return FEMALE_PREM[coverage_idx][age_band]
    elif gender == MALE:
        return MALE_PREM[coverage_idx][age_band]

def age_to_age_band(age):
    if not (18 <= age <= 60):
        raise ValueError(f"Sorry. Age of {age} yrs not applicable!")
    if age <= 30:
        return 0
    elif age <= 40:
        return 1
    elif age <= 50:

```

```

        return 2
    elif age <= 60:
        return 3

def parse_date_str(dtstr):
    parts = dtstr.replace('/', '-').split('-')
    if parts[0].isdigit():
        return date(int(parts[2]), int(parts[1]), int(parts[0]))
    else:
        return date(int(parts[2]), MONTHS[int(parts[0])], int(parts[1]))

def age_today(dob):
    today = date.today()
    this_bd = dob.replace(year=today.year)
    if today - dob >= this_bd - dob:
        return today.year - dob.year
    else:
        return today.year - dob.year - 1

def read_date():
    while True:
        try:
            return parse_date_str(input('dob? '))
        except Exception as err:
            print("Incorrect date. Please enter the date again.")
            print('Error is probably: ' + str(err))

def read_coverage():
    print("Enter 1/2/3/4 for one of the coverage options: ")
    print("1. $250,000")
    print("2. $500,000")
    print("3. $1,000,000")
    print("4. $2,000,000")
    while True:
        try:
            choice = int(input())
            if 1 <= choice <= 4:
                return COVERAGES[choice - 1]
            else:
                print('Incorrect choice! Accept only 1 to 4!')
        except ValueError as err:
            print('Incorrect choice! Accept only 1 to 4!')

```

```

def read_gender():
    while True:
        gender = input('Enter gender F (Female)/M (Male): ')
        if gender in ['F', 'M']:
            return gender
        else:
            print('Invalid gender! Accept only F or M!')

def run_prem_program():
    while True:
        dob = read_date()
        gender = read_gender()
        coverage = read_coverage()

        try:
            print(retrieve_prem(age_today(dob),
                                gender,
                                coverage))
        except ValueError as dm_err:
            print(f'Domain error: {dm_err}')

        quit_or_cont = input(('Press enter/return to continue.'
                            ' To quit the program, enter quit.'))
        if quit_or_cont == 'quit':
            break

run_prem_program()

```

### ၈.၅ ဖန်ရှင်နှင့် ပရီဂရမိဒီဇိုင်း ဥပမာ (၂) “Colorful House”<sup>၆၇</sup>

အခုတစ်ခါ ပုံ (၈.၁) မှ ရောင်စံအိမ်ပုံလေး ရေးဆွဲကြပါမယ်။ အိမ်အမြင်ဟာ အိမ်အောက်ခြေကနေ အိမ်ခေါင်မှိုး ထိပ်ထိ ဖြစ်တယ်။ အမြင်နဲ့ အကျယ်ဟာ အိမ်အရွယ်အစားကို ဖော်ပြတယ်။ ပုံဆွဲတဲ့အခါ အိမ် အရွယ်အစားနဲ့ အရောင် စိတ်ကြိုက် အလွယ်တကူ ပြောင်းလို့ရသုတေသနတယ်။ အိမ်တည်နေရာကို ဘယ်ဘက် အပေါ်ထောင့်အမှတ် နဲ့ဖော်ပြုမယ် (ပုံတွင်ကြည့်ပါ)။ အိမ်တစ်အိမ်လုံး ကွက်တိဝင်တဲ့ ထောင့်မှုန်စတုဂံ ရဲ့ ( $x, y$ ) လိုလည်း ယူဆနိုင်တယ်။

ခေါင်မှိုးအမြင့် (အပြာရောင် ကြိုက်ပုံ) က အိမ်တစ်ခုလုံး အမြင့် လေးပုံတစ်ပုံ။ ကျွန်ုတ်လေးပုံ သုံးပုံက နံရံအမြင့် (အစိမ်းနှင့်ထောင့်မှုန်စတုဂံ) ဖြစ်တယ်။ နံရံကို အညီအမျှ လေးပိုင်းခွဲပြီး တစ်ပိုင်းစီရဲ့ ဗဟို နဲ့ ပြတောင်းပေါက်ပဲ တစ်ထပ်တည်းကျအောင် နေရာချထားရပါမယ်။

### draw\_window

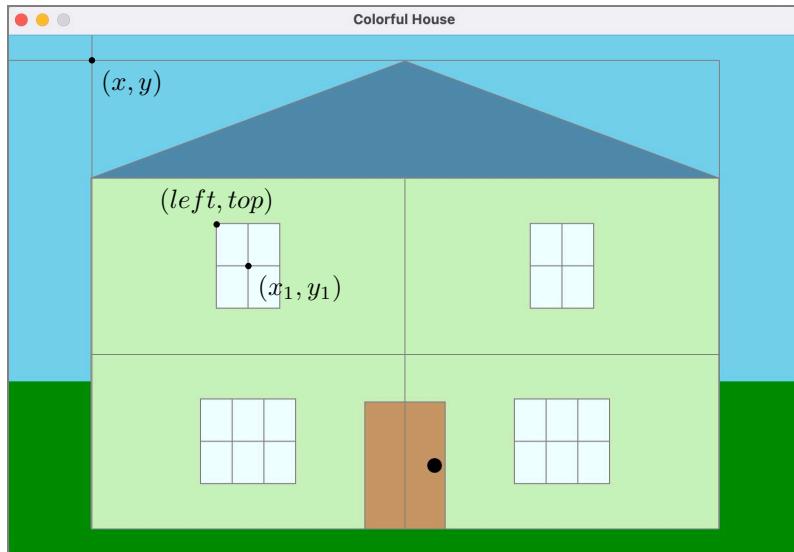
ပြတင်းပေါက်တွေဆဲဖို့ draw\_window ဖန်ရှင် သတ်မှတ်သင့်တယ်။ ပြတင်းပေါက် အရွယ်အစား (မျှန်ချုပ် အရေအတွက်) ကို  $rows \times columns$  နဲ့ ဖော်ပြနိုင်တယ်။ အပေါ်ပြတင်းပေါက် နှစ်ခုက 2  $\times$  2 ဖြစ်ပြီး အောက်နှစ်ခုက 2  $\times$  3 ။ အောက်ဖော်ပြပါ draw\_window မှာ x, y, row, col တို့ဟာ ပြတင်းပေါက် ဗဟိုမှတ်, အရွယ်အစားနဲ့ အရောင် တို့အတွက် ပါရာမိတာတွေဖြစ်တယ်။ နံရဲ ပထမတစ်ပိုင်းရဲ့ ဗဟိုမှတ် ဟာ  $(x_1, y_1)$  ဆိုပါနို့ (ပုံတွင်ကြည့်ပါ)။ ပြတင်းပေါက်ကို အခုလို

```
| draw_window(x1, y1, 2, 2);
```

ဆဲနိုင်ပါတယ်။ နံရဲတစ်ပိုင်းစီအတွက် ဗဟိုမှတ်ရှာတာ သိပ်မခက်ဘူး။ ဒါကြောင့် ဗဟိုမှတ်ကို ပါရာမိတာ အနေနဲ့ ထားတာ အဆင်ပြေတယ်။ (ပြတင်းတစ်ပေါက်ချင်းအတွက် သူ့ရဲ့ ဘယ်ဘက်အပေါ် ထောင့်စွာနဲ့ မှတ်  $(x, y)$  တန်ဖိုးကို တွက်ကြည့်ပါ။ ဗဟိုမှတ်တွက်တာလောက် မလွယ်ကူတာ တွေ့ရလိမ့်မယ်)။

```
# File: colorful_house.py
import arcade
from arcade.color import *

# x, y က ပြတင်းပေါက် အလယ်မှတ်ပါ ပုံထဲမှာ  $(x_1, y_1)$  နဲ့ပြထားတယ်
def draw_window(x, y, row, col):
    pane_width = 30
    pane_height = 40
    window_width = pane_width * col
    window_height = pane_height * row
    top = y - window_height / 2
    left = x - window_width / 2
    for i in range(row):
```



ပုံ ၈.၁ ရောင်ခိုခေါ်လေး

```

for j in range(col):
    arcade.draw_lrtb_rectangle_filled(left + pane_width * j,
                                      left + pane_width * (j+1),
                                      top + pane_height * (i+1),
                                      top + pane_height * i,
                                      AZURE_MIST)
    arcade.draw_lrtb_rectangle_outline(left + pane_width * j,
                                       left + pane_width * (j+1),
                                       top + pane_height * (i+1),
                                       top + pane_height * i,
                                       GRAY)

```

ပြတင်းပေါက် မှန်ချပ်တွေ ဆွဲတာက အနေး (၇) စာမျက်နှာ (၁၀၈) checkerboard လိုပါပဲ။ တက္ကာက ချင်းရဲ့ left, right, top, bottom တွေ loop တစ်ကျော်တိုင်းမှာ မှန်အောင်တွက်နှင့်ရင် ပြတင်းပေါက် ဆွဲလို ရုံးမှုပါ။ မှန်တစ်ချပ် အကျယ်၊ အမြင့် 30, 40 သတ်မှတ်တယ်။ ပြတင်းပေါက်တစ်ခုလုံး အကျယ်၊ အမြင့်ကို

```

window_width = pane_width * col
window_height = pane_height * row

```

ဖော်ပြုလာနဲ့ တွက်ရပါမယ်။ ပြတင်းပေါက် အလယ်မှတ် x, y ကနေ ဘယ်ဘက် အပေါ်ထောင့်စွဲနဲ့မှတ် ကို ရှာမယ်ဆိုရင်

```

top = y - window_height / 2
left = x - window_width / 2

```

(ပုံကိုကြည့်ပြီး  $x_1, y_1$  အမှတ်ကနေ  $left, top$  အမှတ်ကို ဘယ်လိုရှာမလဲ စဉ်းစားကြည့်ပါ)။ မှန်ချပ်တစ် ချပ်ချင်း ကိုထြုဒိန်တွေကို ဦးလိုက် အမှတ်ကို ဘယ်လိုရှာမယ်

<code>left + pane_width * j,</code>	<i># မှန်ချပ်ရဲ့ left x</i>
<code>left + pane_width * (j+1),</code>	<i># မှန်ချပ်ရဲ့ right x</i>
<code>top + pane_height * (i+1),</code>	<i># မှန်ချပ်ရဲ့ bottom y</i>
<code>top + pane_height * i,</code>	<i># မှန်ချပ်ရဲ့ top y</i>

ဥပမာ ( $left, top$ ) = (100, 120) ဆိုပါစို့။ ညာဘက်အောက် မှန်ချပ်အတွက်  $i = 1, j = 1$  ဖြစ်တယ် ( $i \neq j$  တန်ဖိုး သုညာက စတယ်ဆိုတာ အမှတ်ရပါ)။ အဲဒီမှန်ချပ်ရဲ့ ကိုထြုဒိန်တွေက

<code>100 + 30 * 1</code>	$= 130$
<code>100 + 30 * (1 + 1)</code>	$= 160$
<code>120 + 40 * (1 + 1)</code>	$= 200$
<code>120 + 40 * 1</code>	$= 160$

ဖြစ်ပါတယ်။

### draw\_house

`draw_house` ကို အောက်ပါအတိုင်း သတ်မှတ်ထားတယ်။  $left \neq top$  က အိမ်တည်နေရာ  $(x, y)$

လက်ခံတဲ့ ပါရာမီတာတွေပါ။ width နဲ့ height ကဲ အိမ်တစ်ခုလုံးရဲ့ အကျယ်နဲ့ အမြင်အတွက်။

```
# left = x top = y (ပုံစံကြည်ပါ)
def draw_house(left, top, width, height):
    middle = left + width / 2
    wall_top = top + height / 4
    right = left + width
    wall_bottom = top + height

    arcade.draw_lrtb_rectangle_filled(left, right,
                                      wall_bottom, wall_top,
                                      TEA_GREEN)
    arcade.draw_lrtb_rectangle_outline(left, right,
                                      wall_bottom, wall_top,
                                      GRAY)

    arcade.draw_triangle_filled(left, wall_top,
                               right, wall_top,
                               middle, top,
                               AIR_FORCE_BLUE)
    arcade.draw_triangle_outline(left, wall_top,
                               right, wall_top,
                               middle, top,
                               GRAY)

    left_windows_x = left + width/4
    right_windows_x = left + width * 3/4

    wall_height = height * 3/4
    upper_windows_y = wall_top + wall_height/4
    lower_windows_y = wall_top + wall_height/4 * 3

    draw_window(left_windows_x, upper_windows_y, 2, 2)
    draw_window(left_windows_x, lower_windows_y, 2, 3)
    draw_window(right_windows_x, upper_windows_y, 2, 2)
    draw_window(right_windows_x, lower_windows_y, 2, 3)

    door_width = 76
    door_height = 120
    door_top = wall_bottom - door_height
    door_left = left + width/2 - door_width/2
    draw_main_door(door_left, door_top,
                   door_width, door_height,
                   WOOD_BROWN)
```

အိမ်ခေါင်မှူး ထိပ်စွန်းမှတ်က အိမ်ကို အလယ်ကနေ ပိုင်းဖြတ်တဲ့ ဒေါင်လိုက်မျဉ်းပေါ်မှာပါ (ခေါက်ချိုးညီ

မျှော်း)။ ဒီမျှော်းပေါ်က  $x$  အမှတ်ကို middle လို့ ထားမယ်။ သူတွန်ဖိုးက

```
| middle = left + width / 2
```

ဖြစ်တယ်။ နံရုံ အပေါ်အနားရဲ့  $y$  အမှတ်ကို wall\_top လို့ ထားတယ်။ သူတွန်ဖိုးက

```
| wall_top = top + height / 4
```

ဖြစ်မယ်။ top ကို အိမ်အမြင့်ရဲ့ လေးပုံတစ်ပုံ ပေါင်းပေးရမှာပါ။ နံရုံ အောက်ဘက်အနားရဲ့  $y$  အမှတ်ကို wall\_bottom လို့ ထားတယ်။ သူတွန်ဖိုးက

```
| wall_bottom = top + height
```

နံရုံ ညာဘက်အနားရဲ့  $x$  အမှတ်ကို right လို့ ထားတယ်။ သူတွန်ဖိုးက

```
| right = left + width
```

(ရေပြင်ညီမျှော်းပေါ်က အမှတ်အားလုံး  $y$  တန်ဖိုး တူညီတယ်၊ ဒေါင်လိုက်မျှော်းပေါ်က အမှတ်အားလုံး  $x$  တန်ဖိုး တူညီတယ်လို့ သိတယ်ပါ)။

ခေါင်မိုးကို draw\_triangle\_filled နဲ့ အခုလိုဆွဲနိုင်ပါတယ်။ ပြိုက်ရဲ့ ထောင့်စွန်းမှတ်တွေ ထည့်ပေးရမှာပါ။ (ပြိုက်အနားသတ်ကို draw\_triangle\_outline နဲ့ ဆွဲတယ်။ ရွှေ့မျက်နှာ draw\_house မှာ ကြည့်ပါ)။

```
# draw roof
arcade.draw_triangle_filled(left, wall_top,      # ဘယ်ထောင့်စွန်း
                            right, wall_top,    # ညာထောင့်စွန်း
                            middle, top,       # အပေါ်ထောင့်စွန်း
                            AIR_FORCE_BLUE)
```

ပြတင်းပေါက်တွေရဲ့ အလယ်မှတ်တွေကို အောက်ပါအတိုင်း တွက်ထားတယ်။ ဘယ်ဘက် ပြတင်းနှစ် ပေါက် အလယ်မှတ်  $x$  တန်ဖိုး တူတယ်။ ညာဘက်နှစ်ခု အလယ်မှတ်လည်း  $x$  တန်ဖိုး တူတယ်။ အပေါ် ပြတင်းနှစ်ပေါက် အလယ်မှတ်  $y$  တန်ဖိုး တူတယ်။ အောက်နှစ်ခု အလယ်မှတ်လည်း  $y$  တန်ဖိုး တူတယ်။ နံရုံအမြင့်က အိမ်အမြင့်ရဲ့ လေးပုံသုံးပုံ (ခေါင်မိုးက လေးပုံတစ်ပုံ ဖြစ်တဲ့အတွက်)။

```
# center points of the windows
left_windows_x = left + width/4
right_windows_x = left + width * 3/4

wall_height = height * 3/4
upper_windows_y = wall_top + wall_height/4
lower_windows_y = wall_top + wall_height/4 * 3
```

ဒီအမှတ်တွေသိရင် ပြတင်းပေါက်တွေကို ဒီလိုဆွဲရှုပဲ

```
draw_window(left_windows_x, upper_windows_y, 2, 2)
draw_window(left_windows_x, lower_windows_y, 2, 3)
draw_window(right_windows_x, upper_windows_y, 2, 2)
draw_window(right_windows_x, lower_windows_y, 2, 3)
```

အိမ်တံခါးမ ကို အခုလို ဆွဲပါတယ်။ အမြင့်နဲ့ အကျယ်ကို ကြည့်ကောင်းအောင် သင့်တော်သလိုထားနိုင်တယ်။ နေရာအတွက် door\_top, door\_left တွက်တာလည်း မခက်ပါဘူး။ နားလည်မှာပါ။

```
# draw main door
door_width = 76
door_height = 120
door_top = wall_bottom - door_height
door_left = left + width/2 - door_width/2
draw_main_door(door_left, door_top,
                door_width, door_height,
                WOOD_BROWN)
```

အတွက်အချက်တွေ အားလုံး ဒီလောက်ပါပဲ။ အရမ်းမခက်ဘူး ထင်ပါတယ်။ draw\_house ဖန်ရှင်တစ်ခုလုံး ရှေ့စူးမျက်နှာမှာ အစအဆုံး ပြန်လည့်လာကြည့်ပါ။

### **draw\_main\_door**

draw\_main\_door ဖန်ရှင်က ဒီလို ... left နဲ့ top က တံခါးမ အပေါ်ဘယ်ဘက်ထောင့် အမှတ်ပါ။ တံခါးသော့ကို draw\_circle\_filled နဲ့ ဆွဲတယ်။ စက်ဝိုင်း ဗဟိုမှတ်ကို ထည့်ပေးရတယ်။ ဘယ်၊ ညာ၊ အထက်၊ အောက် ဘယ်လိုတွက်ထားလဲ အခုလောက်ဆုံး နားလည်လောက်ပြီလို့ ယူဆတယ်။

```
def draw_main_door(left, top, width, height, color):
    arcade.draw_lrtb_rectangle_filled(left, left + width,
                                      top + height, top,
                                      color)
    arcade.draw_lrtb_rectangle_outline(left, left + width,
                                       top + height, top,
                                       GRAY)

    # draw door lock
    arcade.draw_circle_filled(left + width - 10,  # circle center x
                             top + height/2,      # circle center y
                             7,                      # radius
                             BLACK)
```

အိမ်ပုံဆွဲတဲ့အခါ ခေါက်ချိုးညီ ပေါ်အောင်နဲ့ သင့်တော်တဲ့ အရွယ်အစားဖြစ်အောင် အခုလိုတွက်ထားပါတယ်

```
WIN_WIDTH = 754
WIN_HEIGHT = 492
SIDE_MARGIN = 80
TOP_BTM_MARGIN = 25
HOUSE_WIDTH = WIN_WIDTH - 2 * SIDE_MARGIN
HOUSE_HEIGHT = WIN_HEIGHT - 2 * TOP_BTM_MARGIN
```

ဝင်းဒီး အကျယ်နဲ့ အမြင့် 754, 492 pixels ထားတယ်။ ဝင်းဒီးရဲ့ ဘယ်၊ ညာနဲ့ အထက်၊ အောက် ဘောင်တွေကနေ အိမ်ကို 80, 25 pixels ခွာထားတယ်။

နောက်ဆုံးတော့ Arcade လိုက်ဘရဲ့ လိုအပ်ချက်အတိုင်း အခုလို တစ်ဆင့်ချင်း ပြင်ဆင်ပြီး ရောင်စုံ

၃၆

အိမ်ပံ့လေး ဆွဲလိုပါပြီ

```
arcade.open_window(WIN_WIDTH, WIN_HEIGHT, "Colorful House")
arcade.set_viewport(0, WIN_WIDTH, WIN_HEIGHT, 0)

# Set the background color
arcade.set_background_color(arcade.color.SKY_BLUE)

# Get ready to draw
arcade.start_render()

# იძალვეთ გრძელ აუქნის ფონის ფერს (ტერენის ფერი)
arcade.draw_lrtb_rectangle_filled(0, WIN_WIDTH,
                                  WIN_HEIGHT, WIN_HEIGHT * 2/3,
                                  FOREST_GREEN)
draw_house(SIDE_MARGIN, TOP_BTM_MARGIN, HOUSE_WIDTH, HOUSE_HEIGHT)

# Finish drawing
arcade.finish_render()

# Keep the window up until someone closes it.
arcade.run()
```

မှတ်ချက်။ ။ အခန်း (၇) စာမျက်နှာ (၁၀၅) မှာ Arcade လိုက်ဘရီအကြောင်း မိက်ဆက်ပေးထားပါတယ်။ အဲဒီအပိုင်းဖတ်ထားမှ အခြေငါးပြတ္တတေကို သေချာနားလည်နိုင်မှာပါ။

ရောင်စုအိမ်လေးဆွဲတဲ့ ပရိုဂရမ်ကို တစ်ဆက်တည်း ဆက်စပ် ကြည့်လိုအဆင်ပြအောင် အစအဆုံး ထည့်ပေးထားတယ်။ လေ့လာကြည့်ပါ။

```
arcade.draw_lrtb_rectangle_outline(left + pane_width * j,
                                   left + pane_width * (j+1),
                                   top + pane_height * (i+1),
                                   top + pane_height * i,
                                   GRAY)

def draw_main_door(left, top, width, height, color):
    arcade.draw_lrtb_rectangle_filled(left, left + width,
                                      top + height, top,
                                      color)
    arcade.draw_lrtb_rectangle_outline(left, left + width,
                                      top + height, top,
                                      GRAY)

    # draw door lock
    arcade.draw_circle_filled(left + width - 10,
                             top + height/2,
                             7,                                     # radius
                             BLACK)

def draw_house(left, top, width, height):
    middle = left + width / 2
    wall_top = top + height / 4
    right = left + width
    wall_bottom = top + height

    arcade.draw_lrtb_rectangle_filled(left, right,
                                      wall_bottom, wall_top,
                                      TEA_GREEN)
    arcade.draw_lrtb_rectangle_outline(left, right,
                                      wall_bottom, wall_top,
                                      GRAY)

    arcade.draw_triangle_filled(left, wall_top, right,
                               wall_top, middle, top,
                               AIR_FORCE_BLUE)
    arcade.draw_triangle_outline(left, wall_top,
                               right, wall_top,
                               middle, top,
                               GRAY)

    # center points of the windows
    left_windows_x = left + width/4
    right_windows_x = left + width * 3/4
```

99

```
wall_height = height * 3/4
upper_windows_y = wall_top + wall_height/4
lower_windows_y = wall_top + wall_height/4 * 3

draw_window(left_windows_x, upper_windows_y, 2, 2)
draw_window(left_windows_x, lower_windows_y, 2, 3)
draw_window(right_windows_x, upper_windows_y, 2, 2)
draw_window(right_windows_x, lower_windows_y, 2, 3)

# draw main door
door_width = 76
door_height = 120
door_top = wall_bottom - door_height
door_left = left + width/2 - door_width/2
draw_main_door(door_left, door_top,
               door_width, door_height,
               WOOD_BROWN)

WIN_WIDTH = 754
WIN_HEIGHT = 492
SIDE_MARGIN = 80
TOP_BTM_MARGIN = 25
HOUSE_WIDTH = WIN_WIDTH - 2 * SIDE_MARGIN
HOUSE_HEIGHT = WIN_HEIGHT - 2 * TOP_BTM_MARGIN

# Open up a window.
# From the "arcade" library, use a function called "open_window"
# Set the window title to "Colorful House"
# Set the dimensions (width and height)
arcade.open_window(WIN_WIDTH, WIN_HEIGHT, "Colorful House")
arcade.set_viewport(0, WIN_WIDTH, WIN_HEIGHT, 0)

# Set the background color
arcade.set_background_color(arcade.color.SKY_BLUE)

# Get ready to draw
arcade.start_render()

arcade.draw_lrtb_rectangle_filled(0, WIN_WIDTH,
                                 WIN_HEIGHT, WIN_HEIGHT * 2 / 3,
                                 FOREST_GREEN)
draw_house(SIDE_MARGIN, TOP_BTM_MARGIN, HOUSE_WIDTH, HOUSE_HEIGHT)

# Finish drawing
arcade.finish_render()
```

```
# Keep the window up until someone closes it.
arcade.run()
```

## ပိုကောင်းအောင် ပြုပြင်သင့်တဲ့ နေရာလေးတွေ

ရောင်စုံအိမ်လေး ပရိဂရမ်က အတန်အသင့် အဆင်ပြေနေပါပြီ။ တချို့နေရာလေးတွေ ပိုကောင်းအောင် improve လုပ်လို့ရပါတယ်။ ပြင်ဖိုက သိပ်လည်းမခက်ဘူး။ ထောင့်မှန်စတုဂံတွေကို နှစ်ခါဆွဲနေရတယ်။ အရောင်ဖြည့်စုံ draw\_lrtb\_rectangle\_filled နဲ့တစ်ခါ အနားသတ်အတွက် draw\_lrtb\_rect angle\_outline နဲ့တစ်ခါ။ ဖန်ရှင်မှာ ထည့်ပေးတဲ့ တန်ဖိုးတွေကလည်း အားလုံး တူတူပဲ (အရောင်က လွှဲလို့)။ အခုလို ဖန်ရှင် သတ်မှတ်ထားလိုက်ရင် ပိုအဆင်ပြုသွားမှာပါ

```
def draw_rect_filled_with_outline(left, right, top, bottom, color):
    arcade.draw_lrtb_rectangle_filled(left, right, top, bottom, color)
    arcade.draw_lrtb_rectangle_outline(left, right, top, bottom, GRAY)
```

မိုးခိုးရောင် အနားသတ်နဲ့ rectangle ကို အရောင်အမျိုးမျိုး ဖြည့်ဆွဲလို့ရမှာပါ။ ဥပမာ draw\_window မှာ အခုလို သုံးလို့ရမယ်

```
draw_rect_filled_with_outline(left + pane_width * j,
                               left + pane_width * (j+1),
                               top + pane_height * (i+1),
                               top + pane_height * i,
                               AZURE_MIST)
```

နံရံနဲ့ တံခါးမဆွဲရင်လည်း အလားတူ သုံးရုံပဲ။

## ၈.၆ ပါရာမီတာ ခွေးချယ်သတ်မှတ်ခြင်း

ဖန်ရှင်သတ်မှတ်တဲ့အခါ ‘ဘယ်ဟာတွေက ပါရာမီတာ ဖြစ်သင့်လဲ’၊ ‘ဘယ်ဟာတွေကတော့ရော မဖြစ်သင့်ဘူးလား’ ဝေခဲ့ဆုံးဖြစ်မရ ဖြစ်လေ့ရှိတာဟာ စလေ့လာတဲ့သူတွေရော အတွေ့အကြုံပြီး သူတွေအတွက်ပါ ပုံမှန် ဖြစ်ရှိဖြစ်စဉ် တစ်ခုပါပဲ။ ဖော်မြှုလာ ရှိပြီးသား သချုပ်ဖန်ရှင်တော့အတွက်တော့ သိပ်ခေါင်းစားစရာ မလိုပါဘူး။ စက်ပိုင်းရေးယာရှာရင် အချင်းဝက်၊ ကြိုက်ဆိုရင် အခြေနဲ့ အမြင့်တိုက် ပါရာမီတာ အနေနဲ့ ထားရမယ်ဆိုတယ်။ ဒါပေမဲ့ တချို့နေရာတွေမှာ ထင်သလောက် မရှိရှင်းပါဘူး။

ယော်ယူအားဖြင့် ဖန်ရှင် ခေါ်တဲ့အခါကျတော့မှ တန်ဖိုးထည့်ပေးချင်တဲ့ အရာတွေကို ပါရာမီတာ အနေနဲ့ ထားလေ့ရှိတယ်။ ရောင်စုံအိမ်လေးမှာ draw\_window ဖန်ရှင်ခေါ်တော့မှ ဆွဲမဲ့ ပြတင်းပေါက် တည်နေရာနဲ့ row နဲ့ column အရေအတွက် ထည့်ပေးရမှာပါ။ ဒီတော့မှ လိုချင်တဲ့နေရာမှာ လိုချင်တဲ့ row, column အရေအတွက်နဲ့ ဆွဲလို့ရမယ်။

အခုလက်ရှိ draw\_window ဟာ ပြတင်းပေါက်အရောင် တစ်မျိုးနဲ့ပဲ ဆွဲလိုရနိုင်မယ်။ အကယ်၍ အပေါ်ပြတင်းပေါက် နှစ်ခုနဲ့ အောက်ပြတင်းပေါက် နှစ်ခု အရောင်ကဲ့နေတယ်ဆိုပါစို့။ ဒါဆိုရင်တော့ draw\_window ကို အခုလို ပြင်ပေးရပါမယ်

```
def draw_window(x, y, row, col, color):
    # ... ဒီနေရာက လိုင်းတွေ တချို့ကို ချွဲစွဲထားတယ်
    for i in range(row):
```

```

for j in range(col):
    draw_rect_filled_with_outline(left + pane_width * j,
                                   left + pane_width * (j+1),
                                   top + pane_height * (i+1),
                                   top + pane_height * i,
                                   color)

```

အပေါက် အဖြူရောင်၊ အောက်ကို မီးခါး ဆွဲမယ်ဆိုရင်

```

draw_window(left_windows_x, upper_windows_y, 2, 2, WHITE)
draw_window(right_windows_x, upper_windows_y, 2, 2, WHITE)
draw_window(left_windows_x, lower_windows_y, 2, 3, GRAY)
draw_window(right_windows_x, lower_windows_y, 2, 3, GRAY)

```

ဒီလိုဆွဲလို ရပါတယ်။

တကယ်လို မှန်ချပ်အရွယ်ပါ ဖန်ရှင်ခေါ်တဲ့အခါကျတော့မှ သတ်မှတ်လိုရချင်ရင် မှန်ချပ် အကျယ် အမြင့် အတွက် ပါရာမီတာနှစ်ခု ထပ်ထည့်နိုင်ပါတယ်

```

def draw_window(x, y, row, col, color, pane_width, pane_height):
    # pane_width = 30 ဒီနှစ်ကြောင်း မလိုတော့ဘူး
    # pane_height = 40
    window_width = pane_width * col
    window_height = pane_height * row
    top = y - window_height / 2
    left = x - window_width / 2
    for i in range(row):
        for j in range(col):
            draw_rect_filled_with_outline(left + pane_width * j,
                                           left + pane_width * (j+1),
                                           top + pane_height * (i+1),
                                           top + pane_height * i,
                                           color)

```

ပါရာမီတာတွေ ထပ်ဖွည့်ခြင်းအားဖြင့် ပြတင်းပေါက်ကို အရောင် အမျိုးမျိုးနဲ့ ဆွဲလိုရလာတယ်။ မှန်ချပ် အရွယ် အမျိုးမျိုးနဲ့ ဆွဲလိုရလာတယ်။ ဖန်ရှင်ဟာ ပိုပြီး ဂျွန်ရယ်ကျလာတယ်။ flexible ပိုဖြစ်လာတယ်။ ဒီတော့ တတ်နိုင်သူမျှ ပါရာမီတာတွေ များ များထားရင် ပိုကောင်းတယ် ယူဆရမှာလား။ ပါရာမီတာတွေ များ လာတာနဲ့အမျှ သူတို့ကို ဘာအတွက်ထည့်ရေတာလဲ၊ ဘာတန်ဖိုးထည့်သင့်လဲ စဉ်းစားဖို့လိုလာတယ်။ ဖန်ရှင် ကို အသုံးပြုရတာ ပိုခက်လာတယ်။ ဒါကြောင့် flexibility နဲ့ အသုံးပြုရ လွယ်ကူမှု ပေါ်မှုတည်ပြီး ချင့်ချိန် သတ်မှတ်ရမှာပါ။

ဂျွန်ရယ်ကျတဲ့ flexible ဖြစ်တဲ့ ဖန်ရှင်တစ်ခုကို အခြေခံပြီး specific ဖြစ်တဲ့ ဖန်ရှင်တွေကို သတ်မှတ်ယူလိုရတယ်။ ဒီဖန်ရှင်နှစ်ခုမှာ အပေါက် draw\_window ကို ပြန်သုံးထားပါတယ်။

```

def draw_2x2_pearl_window(x, y):
    draw_window(x, y, 2, 2, PEARL, 30, 40)

```

```

def draw_2x3_gray_window(x, y):

```

```
    draw_window(x, y, 2, 3, LIGHT_GRAY, 30, 40)
```

$2 \times 2$  ပုလဲရောင်နဲ့  $2 \times 3$  မီးခါးရောင် ပြတင်းပေါက်ဆွဲဖို့ပါ။ နေရာအတွက် ပါရာမိတာ နှစ်ခုပဲ ပါတော့ တယ်။ သုံးရတာ ပိုလွယ်သွားတာ သိသိသာတွေ့ရမှာပါ။ ဥပမာ ရောင်စုံအိမ်လေးမှာ အပေါ်ပြတင်းနှစ် ပေါက်ကို ပုလဲရောင်၊ အောက်နှစ်ခုကို မီးခါးရောင် လိုချင်ရင် အခုလို ဆဲရုံပါပဲ

```
draw_2x2_pearl(left_windows_x, upper_windows_y)
draw_2x3_gray(left_windows_x, lower_windows_y)
draw_2x2_pearl(right_windows_x, upper_windows_y)
draw_2x3_gray(right_windows_x, lower_windows_y)
```

အရောင်နဲ့ row, column တော့ ပြောင်းချင်လို့ မရဘူး။ ပုံသေ ဒီနှစ်မျိုးပဲ ရပါမယ်။ ဒါကြောင့် flexibility အရတော့ အားနည်းတယ်ပေါ့။ ဒါပေမဲ့ သုံးရတာ အတော်လေးလွယ်တယ်။ ဂျှိန်ရယ်ကျေတဲ့ flexible ဖြစ်တဲ့ ဖန်ရှင်ကို အခြေခံပြီး specific ဖြစ်တဲ့ ဖန်ရှင် သတ်မှတ်တဲ့ နည်းလမ်းကို ပရိုဂရမ်မာတွေ မကြာခဲာ အသုံးပြုလေ့ရှိတယ်။ ကိုယ်ဝိုင် အသုံးချုတ်တောင် လေ့လာထားသင့်ပါတယ်။ အသုံးတည့် မှုပါ။ နားလည်ဖို့ သိပ်ခက်ခဲတဲ့ ကိစ္စလည်း မဟုတ်ပါဘူး။

## ၈.၃ တန်ဖိုးပြန်ပေးခြင်း၊ မပေးခြင်း

တချို့ ဖန်ရှင်တွေမှာ တန်ဖိုးပြန်ပေးသင့် (သို့) မပေးသင့် ဆုံးဖြတ်ရ ခက်တတ်ပါတယ်။ ဒါနဲ့ပါတ်သက် ပြီး ပထမဆုံး သိတားသင့်တာကတော့ တန်ဖိုးပြန်ပေးခြင်း (သို့) မပေးခြင်းရဲ့ အကျိုးဆက်က ဘယ်လို ရှိမလဲ ဆိုတာပါ။

```
def greet_guest_nort(guest_name):
    print(f'Hello, {guest_name}.')

def greet_guest_rt(guest_name):
    return f'Hello, {guest_name}.'
```

ဒီဖန်ရှင်နှစ်ခုက သဘောတရား ဆင်တူပါတယ်။ ကွာခြားတာဆိုလို့ ပထမတစ်ခုက တစ်ခါတည်း print နဲ့ စာသားကို output ထုတ်ပေးပြီး နောက်တစ်ခုကတော့ စာသားကိုပဲ return ပြန်ပေးတယ်။ ဒုတိယ တစ်ခုနဲ့ စခရင်မှာထုတ်ချက်ရင်

```
print(greet_guest_rt("Joe"))
```

ဖန်ရှင်ခေါ်တဲ့သူက print လုပ်ပေးရပါမယ်။ သုံးတဲ့သူအနေနဲ့တော့ အလုပ်ပိုတာပေါ့။ ဒါပေမဲ့ ဖန်ရှင်က ပြန်ရတဲ့ တန်ဖိုးကို လိုအပ်သလို ဆက်လက် အသုံးပြုခိုင်တယ်။ ဥပမာ greet လုပ်ပြီးရင် နေကောင်းလား ဆက်မေးချင်တယ်၊ ဒါမှုမဟုတ် စာသားထဲက အကွား/စကားလုံး အတေးထိုးမယ် ဆိုပါစို့

```
print(greet_guest_rt("Joe") + '\nHow are you doing?')
print(greet_guest_rt('Kathy').replace('.!', '!!!!'))
```

ဖန်ရှင်ခေါ်တဲ့သူက ပြန်ရတဲ့ တန်ဖိုးကို လိုအပ်သလို ဆက်လက် အသုံးပြုလို့ ရပါမယ်။ တန်ဖိုးပြန်မရရင် တော့ ဒီလိုလုပ်လို့ ရမှာ မဟုတ်ပါဘူး။

နောက်တစ်ခုက တန်ဖိုးပြန်ပေးရင် မှန်/မမှန်စစ် (test) လုပ်ရတာ ပိုလွယ်တယ်။ ဒါက စာမျက်နှာ (၁၀၄) က ခရစ်မတ် သစ်ပင်လေးကို ဖန်ရှင်ရေးထားတာပါ

## ၁၅၂

```
def christmas_tree(leaf_rows, trunk_rows):
    tree = []
    for r in range(leaf_rows):
        tree_row = ' ' * (leaf_rows - 1 - r)
        tree_row += '*' * (r * 2 + 1)
        tree.append(tree_row)
    trunk_width = 3
    for r in range(trunk_rows):
        tree_row = ' ' * (leaf_rows - 2)
        tree_row += '*' * trunk_width
        tree.append(tree_row)

    return tree
```

Row တစ်ခုချင်းကို string အနေနဲ့ list ထဲမှာ ထည့်ပြီး return ပြန်ပေးထားတယ်။ Output ထိုတ်မယ်ဆိုရင်

```
my_tree = christmas_tree(8, 4)
for row in my_tree:
    print(row)
```

ဒီဖန်ရှင် မှန်/မမှန် test လုပ်ဖို့ အခုလို ဖန်ရှင်တစ်ခု ရေးထားနိုင်ပါတယ်။

```
def test_christmas_tree():
    test_tree = [
        '      *',
        '     ***',
        '    *****',
        '   *****',
        '*****',
        ' ***',
        ' ***',
        ' ***',
    ]
    result_tree = christmas_tree(5, 3)
    if result_tree != test_tree:
        print("Error: expected and actual result don't match!")
    else:
        print("Expected and actual result do match!")
```

အကြောင်းတစ်ခုခုကြောင့် `christmas_tree` ရဲ့ အတွင်းပိုင်း တည်ဆောက်ထားပုံကို (ပိုမြန်အောင် ဒါမှုမဟုတ် ပိုမြန်အောင် လွယ်ကူအောင်) ပြင်ရေးမယ်ဆိုရင် ရလဒ် နို့အတိုင်း ဖြစ်မဖြစ် ပြန်စစ်ဖို့ လိုပါ တယ်။ တကယ်လို့ `christmas_tree` က တန်ဖိုးပြန်မပေးဘဲ တစ်ခါတည်း `print` ထုတ်လိုက်ရင် စခရင်္ာ ပေါ်လာတဲ့ output ကိုကြည့်ပြီး လူကိုယ်တိုင် စစ်ဆေးမှပဲ မှန်/မမှန် သိနိုင်မှာပါ။ ပရိုဂရမ်ရေးပြီး အခုလို automated testing (လူက test မလုပ်ဘဲ ပရိုဂရမ်နဲ့ပဲ test လုပ်တာကို ဆိုလို) လုပ်ဖို့ မဖြစ်နိုင်ဘူး။

အခုဖော်ပြခဲသလို testing လုပ်တာဟာ *unit testing* ရဲ့ အခြေခံသဘောတရားပါပဲ။ ဖန်ရှင်တစ်ခု (သို့) မော်ဒူးတစ်ခုကို သူချဉ်းသီးခြား မှန်/မမှန် test လုပ်တာကို unit testing လို့ ခေါ်တာပါ။ တန်ဖိုးပြန်ပေးတဲ့ ဖန်ရှင်တွေဟာ unit test လုပ်ရ လွယ်ကူတာ တွေ့်င်ပါတယ်။ ဆော်ဖဲ့တည်ဆောက်ရာမှာ *unit testing* ဟာ အရေးကြီးတဲ့ အခန်းကဏ္ဍတစ်ခု အနေနဲ့ ပါဝင်တယ်။ ကျယ်ပြန်တဲ့ နယ်ပယ်ဖြစ်ပြီး သီးခြားလေးလုပ် လိုအပ်ပါတယ်။ ဒီစာအပ်မှုတော့ ဖန်ရှင်တန်ဖိုးပြန်ပေးခြင်နဲ့ ဆက်စပ်နေတဲ့ အတွက် အခြေခံလောက်လေးကိုပဲ ဗဟိုသုတအနေနဲ့ ထည့်သွင်းဖော်ပြတာပါ။

## ၈.၈ return

`return` ဟာ ဖန်ရှင်ကနေ သူကိုခေါ်တဲ့သူ (သို့) ခေါ်ထားတဲ့နေရာဆီ ပြန်ရောက်သွားစေတယ်လို့ သိထားပြီ။ သတိပြုမိသင့်တဲ့ အသေးစိတ်အချက် တချို့ကို ဆက်ကြည့်ရအောင်။ `get_sign` နဲ့ `print_sign` ကို cascading if မသုံးဘဲ အခုလိုရေးလိုရပါတယ်

```
def get_sign(r):
    if r > 0:
        return 'positive'
    if r < 0:
        return 'negative'

    return 'zero/nosign'

def print_sign(r):
    if r > 0:
        print('positive')
        return
    if r < 0:
        print('negative')
        return

    print('zero/nosign')
    return # ဒါမိပါလည်းရတယ်
```

ရှိုးရိုး if သုံးထားပေမဲ့ ကွန်ဒီရှင်မှန်ရင် `return` ဖြစ်သွားမှုလိုလို အောက်ကို မရောက်တော့ဘူး။ ဒီအတွက် cascading if အလုပ်လုပ်ပုံလိုပဲ ဖြစ်သွားတယ်။ အပေါ်မှာ မှန်ရင်အောက်ကို မရောက်လာနိုင်ဘူး။ အောက်က if တွေကို ရောက်လာတာဟာ အပေါ်က ကွန်ဒီရှင်တွေ မှားလိုပဲ။ (အခုလိုမျိုးရေးသင့်တယ်လို့ မဆိုလိုပါ။ `return` ရဲ့ သဘောတရားကို သတိပြုမိအောင် ဥပမာပြတာပါ။)

ဖန်ရှင်မှာ loop ထဲကနေ `return` လုပ်လိုက်ရင် loop ကို `break` လိုက်သလို ဖန်ရှင်ခေါ်တဲ့နေရာဆီကိုလည်း ပြန်ရောက်သွားမှုပါ။

```
def read_int(prompt):
    while True:
        valin = input(prompt)
        try:
            return int(valin)
        except ValueError as err:
```

```

        if valin == 'quit':
            return
        print('Non-integer data!')

num = read_int('Enter number: ')
print(num)

```

### မတော်တဆုံး None return ဖြစ်ခြင်း

အောက်ပါ ပကတိတန်ဖိုး ဖန်ရှင်မှာ n တန်ဖိုး သုညအတွက် မစစ်မိထားဘူး။

```

def my_abs(n):
    if n > 0:
        return n
    elif n < 0:
        return -n

```

အကယ်၍ print(my\_abs(0)) နဲ့ ထုတ်ကြည့်ရင် None လို့ ပြပါလိမ့်မယ်။ n သုညဖြစ်တဲ့အတွက် if...elif ထဲက return နှစ်ခုလုံး မလုပ်ဘူး။ ဒီတော့ if...elif အပြီး နောက်တစ်ကြောင်း ကို ရောက်လာတဲ့ သဘောပဲ။ ဖန်ရှင်ဘေးလောက်အဆုံးထိ ရောက်လာပြီးရင် ဒေါ်ခဲ့တဲ့နေရာကို ပြန်သွား ဖို့ return (နောက်မှာ တန်ဖိုးမပါ) လုပ်တယ်လို့ ယူဆရမှာပါ။ တန်ဖိုးမပါတဲ့ return က None ကို ပြန်ပေးပါတယ်။ ဒီလောက်ဆိုရင် ဘာလို့ print(my\_abs(0)) က None ရတာလဲ နားလည်းမှာပါ။ ဒီ အမှားကို ပြင်ထားတဲ့ ဥပမာနှစ်ခုကို ပြထားပါတယ်။ ဒုတိယတစ်ခုက အတွေ့အကြံရှိတဲ့ Python ပရီ ဂရမ်မာတစ်ယောက် ရေးလေ့ရှိတဲ့နည်း။ Pythonic ပိုဖြစ်တယ် (Python ပိုဆန်တယ်) ပေါ့။

```

def my_abs(n):
    if n > 0:
        return n
    elif n < 0:
        return -n
    else:
        return 0

def my_abs(n):
    if n < 0:
        return -n
    return n

```



# အခန်း ၉

## From Classes to Objects (ကလပ်စုများမှ အော်ဂျက်များသို့)

အခန်း (၆) မှာ အော်ဂျက်တရီးနဲ့ မိတ်ဆက်ပေးခဲ့တယ်။ `date`, `list`, `dict` စသည်ဖြင့်။ အော်ဂျက်တွေကို ကလပ်စုတစ်ခုကနေ တည်ဆောက်ယူရတာပါ။ `date` အော်ဂျက်တွေကို `date` ကလပ်စု `Fraction` အော်ဂျက်တွေကို `Fraction` ကလပ်စုတော် ဖန်တီးယူတယ်။ အော်ဂျက်တွေကို `class` တစ်ခုရဲ့ `instance` လိုလည်းခေါ်တယ်။ `Fraction` အော်ဂျက်တွေဟာ `Fraction` ကလပ်စု `instance` လိုလည်း ပြောလေ့ရှိတယ်။ အော်ဂျက်တွေဟာ `class` `instance` တွေဖြစ်တယ်။ (အကြမ်းဖျဉ်းအားဖြင့် အော်ဂျက်နဲ့ `instance` တူတယ်ပြောလိုရပေမဲ့ အနက်အဓိပါယ်အရ အနိတ် ကွာဟနျက်တွေ ရှိတာကိုလည်း inheritance အကြောင်း လေ့လာတဲ့အခါ တွေ့ရပါယ်)။

ဒီအခန်းမှာ ကလပ်စုအကြောင်းကို လေ့လာကြမှာပါ။ ကိုယ်ပိုင် ကလပ်စုသတ်မှတ်ပြီး အဲဒီကလပ်စု အော်ဂျက်တွေ တည်ဆောက် ယူနိုင်တော့မှာဖြစ်တယ်။ အခြားသူတွေရေးထားပေးတဲ့ လိုက်ဘရိဖန်ရှင်တွေကို အသုံးပြုရာကနေ နောက်ပိုင်းမှာ ကိုယ်ပိုင် ဖန်ရှင်တော့ သတ်မှတ်လာနိုင်တာဟာ အဆင့် တစ်ဆင့် ပုံမြေင့်လာသလိုပါပဲ။ ကိုယ်ပိုင်ကလပ်စု ဒီဇိုင်းပြုလုပ် ဖန်တီးနိုင်လာတာဟာလည်း နောက်တစ်ဆင့် ထပ်မံ့ပိုင်လာတယ်လို့ ဆုံးရမှာပါ။ ပရိုဂရမ်တွေကို အမှားနည်းအောင်၊ ပြင်ဆင်ရလွယ်ကူအောင် စနစ်တကျ ဒီဇိုင်းလုပ် တည်ဆောက်လို့ရလာမှာ ဖြစ်ပါတယ်။

ကလပ်စုကို abstraction mechanism တစ်ဖုံးလိုလည်း ရှိမြင်နိုင်တယ်။ Abstraction လုပ်တယ်ဆိတ် ဘယ်လိုတည်ဆောက်ထားလဲ သိစရာမလိုဘဲ အသုံးပြုလိုရစေတာကို ဆုံးလိုတာ။ ဖန်ရှင်တွေနဲ့ abstraction လုပ်တာကို တွေ့ခဲ့ကြပြီးပါပြီ။ ကလပ်စုတွေဟာလည်း abstraction အတွက် အထောက်အကူဗြို့ နည်းလမ်းတစ်မျိုး (mechanism) ပါပဲ။ အများကြီး ပုံးဖွဲ့မှုများထက်တဲ့ နည်းလမ်းလို့ ဆုံးရမှာပါ။ အစွမ်းထက်လာတာနဲ့အမျှ သဘောတရားအရရော အသုံးချုတဲ့အပိုင်းမှာပါ ပိုပြီးရှုပ်ထွေးခက်ခဲနိုင်ပါတယ်။ ဒီဇိုင်းပိုင်းအတွက် ကျယ်ကျယ်ပြန်ပြန့် ဆက်လက်လေ့လာဖို့ လိုအပ်မှာဖြစ်ပြီး အတွေ့အကြုံပါ ပေါင်းစပ်ယူရမှာပါ။ စာချည်းဖတ်ရှုရဲ့ မရရှိနိုးဘူး (စာရေးသူ ကိုယ်တွေ့ အတွေ့အကြုံအရ ကိုယ်ပိုင်အမြင်ကို ပြောခြင်းသာ)။ ဒါတွေက နောက်ပိုင်း ဆက်လက်လေ့လာဖို့အတွက်ပေါ့။ အခုတော့ အများကြီးမပြောတော့ဘဲ ကလပ်စုတွေအကြောင်း စလိုက်ကြရအောင် ...

### ၉.၁ Account Class (အကောင့် ကလပ်စု)

ဘဏ်အကောင့်တွေကို `Account` အော်ဂျက်နဲ့ ဖော်ပြုမယ်ဆိုပါမြို့။ အကောင့်နံပါတ်၊ ပိုင်ရှင်၊ လက်ကျွန် ငွေ (balance) စတဲ့ အချက်အလက်တွေဟာ အပြင်က တကယ့် ဘဏ်အကောင့်တစ်ခုအတွက် အရေးပါပါတယ်။ မှတ်သား သိမ်းဆည်းထားရတယ်။ ဒါကြောင့် အပြင်က တကယ့် ဘဏ်အကောင့်ကို ထင်ဟပ်

ဖော်ပြတဲ့ ဆွဲဖို့ အော့ဘ်ဂျက်တွေမှာလည်း ဖော်ပြပါ အချက်အလက်တွေ ပါဝင်သင့်တယ်ဆိုရင် ကျိုး ကြောင်းဆီလျဉ်တယ်ပဲ ယူဆရမှာပါ။ အကောင့်နဲ့ ပါတ်သက်ပြီး အခြား အရေးပါတဲ့ အချက်အလက် တွေ ဒီထက်မက ရှိပါတယ်။ ဥပမာ အကောင့် အမျိုးအစား (saving, current) ၊ ဖွင့်ရက် (open date)၊ ဘဏ်ခွဲအမည်၊ ငွေသွင်း/ငွေထုတ်/ငွေလွှဲမှတ်တမ်း စတဲ့အချက်တွေ ရှိပါတယ်။ တကယ့်လက်တွေ မှာ လိုအပ်မှာဖြစ်ပေမဲ့ အခုံပမာမှာ ရိုးရှင်းရှင်းဖြစ်ဖို့ အဲဒါတွေ ထည့်မစဉ်စားဘဲ ချုန်ခဲ့ရအေ။ က လပ်စ်ရဲ့ သဘောတရားကို ရှင်းပြန့်အတွက် ပထမသုံးချက်နဲ့ လုံလောက်ပါတယ်။

ကေသိမှာ အကောင့်နံပါတ် 0086-6002-1111 နဲ့ လက်ကျွန်းငွေ တစ်သိန်းရှိတဲ့ အကောင့်တစ်ခု ရှိတယ် ဆိုပါစို့။ Account ကလပ်စ်သာ ရှိမယ်ဆိုရင် အဲဒီ ကေသိအကောင့်ကို ကိုယ်စားပြုတဲ့ အော့ဘ်ဂျက် ကို အခုံလို့

```
acc1 = Account('Kathy',
    '0086-6002-1111',
    Decimal("100000.00"))
```

ဖန်တီးယူလို့ ရမှာပါ။ ထိုနည်းတူစွာ ငွေလက်ကျွန်း လေးသိန်းခဲ့၊ နံပါတ် 0086-6002-2211 နဲ့ စနီးအကောင့် အော့ဘ်ဂျက်ကို ဒီလို

```
acc2 = Account('Sandy',
    '0086-6002-2211',
    Decimal("450000.00"))
```

ဖန်တီးယူနိုင်မယ်။ ဒါက Account ကလပ်စ်သာ ရှိခဲ့ရင် အော့ဘ်ဂျက် ဘယ်လို့ ဖန်တီးယူရမလဲ စဉ်းစားကြည့်တာပေါ့။ ကလပ်စ်မရှိတဲ့အတွက် အမှန်တကယ်တော့ မရသေး။

date, list, စတဲ့ အော့ဘ်ဂျက်တွေ အပေါ်မှာ အော်ပရေးရှင်းတွေ လုပ်ဆောင်လို့ရတာ တွေ့ခဲ့တယ်။ Account အော့ဘ်ဂျက်တွေမှာရေး ဘယ်လို့ အော်ပရေးရှင်းတွေ လုပ်ဆောင်လို့ရသင့်လဲ။ အပြင် မှာ အကောင့်တစ်ခုကနေ ငွေထုတ်လို့ရတယ် အကောင့်ထဲကို ငွေသွင်းလို့ရပါတယ်။ Account အော့ဘ်ဂျက်တွေမှာလည်း ဒါတွေလုပ်လို့ရသင့်တာပေါ့

```
acc1.deposit(Decimal("50000.00"));
acc2.withdraw(Decimal("70000.00"));
```

ဒီအော်ပရေးရှင်းတွေက အော့ဘ်ဂျက်တွေအပေါ် ဘယ်လို့သက်ရောက်မှုရှိမလဲ။ ကေသိအကောင့် လက်ကျွန်းငွေက တစ်သိန်းခဲ့ ဖြစ်သွားသင့်တယ်။ စနီးအကောင့်က သုံးသိန်းရှုစ်သောင်း ဖြစ်သင့်တယ်။ ဒီလို့ အော်ပရေးရှင်းတွေဟာလည်း ကလပ်စ်ပေါ်မှာ မှုတည်တယ်။ ကလပ်စ်က ထောက်ပံ့ပေးထားမှပဲ ရမယ်။ ကလပ်စ်သည်သာ အခရာလို့ ပြောရမှာပါ။

ကလပ်စ်တစ်ခုဟာ အော့ဘ်ဂျက်တွေမှာ ပါရှိရမဲ့ အချက်အလက်တွေနဲ့ ငြင်းအော့ဘ်ဂျက်တွေအပေါ်မှာ လုပ်ဆောင်နိုင်တဲ့ အော်ပရေးရှင်းတွေကို သတ်မှတ်ပေးပါတယ်။ ဥပမာ အနေနဲ့ Account ကလပ်စ်ဘယ်လို့သတ်မှတ်ရမလဲ ကြည့်ရအေ။ ...

```
# File: account.py
from decimal import *

class Account:
    def __init__(self, holder, acc_number, balance):
```

```

    self.holder = holder
    self.acc_number = acc_number
    self.balance = balance

    def deposit(self, amt):
        if amt <= Decimal(0.00):
            raise ValueError('Invalid amount for deposit!')
        self.balance += amt

    def withdraw(self, amt):
        if amt > self.balance:
            raise ValueError('Not enough balance!')
        self.balance -= amt

```

‘Account ကလပ်စ် သတ်မှတ်ပါမယ်’ လို့ ပြောဖိုအတွက် class Account: နဲ့ စရပါတယ်။ သူအောက်မှာရှိတာက Account ကလပ်စ်ရဲ့ ဘေးဒီ (body) ပါ။ Body ဆိတ် block ကို ပြောတာပါပဲ။ ကလပ်စ်ဘေးဒီထဲမှာ ဖန်ရှင် သုံးခု သတ်မှတ်ထားတာ တွေ့ရမယ်။ သုံးခုလုံးမှာ ပထမ ပါရာမိတာက self ဖြစ်နေတာကို သတိပြုမှုပါ။ သူ (self) က လက်ရှိအေးသာ့ဂျက် (current object) လို့ ဆိုလို တာဖြစ်ပြီး သိပ်မကြာခင် သူအဓိပ္ပာယ်ကို ရှင်းပြပါမယ်။

\_init\_ ဖန်ရှင်က အေးသာ့ဂျက်မှာ ပါဝင်တဲ့ မေရီရောဘဲလွှာတွေ သတ်မှတ်ပေးတယ်။ ငှဩးတို့ ရဲ့ ကန်းတို့ တန်ဖိုးကိုလည်း ဒီဖန်ရှင်က initialize (စထည်) လုပ်ပေးရတာပါ။ အေးသာ့ဂျက်ရဲ့ ကန်းတို့ အခြေအနေ (initial state) ကို စတင်ပေးတဲ့ ဖန်ရှင်ဖြစ်တာကြောင့် initializer လို့ ခေါ်လေ့ရှိတယ်။ Account ကလပ်စ်ကို ကြည့်ရင် ဒီဖန်ရှင်မှာ self.holder, self.acc\_number, self.balance စဗော်တဲ့ အချက်အလက်တွေနဲ့ သက်ဆိုတဲ့ မေရီရောဘဲလွှာတွေ ကြော်ထားတာ တွေ့ရမှာပါ။ ဒီနေရာမှာ dot အမှတ်အသား အဓိပ္ပာယ်ကို ‘၏/ရဲ့’ လို့ ယူဆရင် ဖတ်ရတာ အဆင်ပြေတယ်။ self.holder ကို ‘လက်ရှိအေးသာ့ဂျက်ရဲ့ holder’၊ self.balance ကို ‘လက်ရှိအေးသာ့ဂျက်ရဲ့ balance’ လို့ ဖတ်နိုင်တယ်။ \_init\_ ရဲ့ ထူးချွေးချက်က သူကို တိုက်ရှိက် ခေါ်သုံးရလေ့မရှိဘူး။ အေးသာ့ဂျက် တည်ဆောက်တဲ့ ဖြစ်စဉ်ရဲ့ နောက်ဆုံးအဆင့်မှာ Python က အလိုအလျောက် ခေါ်ပေးတဲ့ ဖန်ရှင်ဖြစ်တယ်။

Account instance တစ်ခုကို ဖန်တီးယူမယ်ဆိုရင် အခုလို ရေးရမှာပါ

```
Account('Amy', '0086-6002-2233', Decimal('350_000.00'))
```

ဒီအခါမှာ အေးသာ့ဂျက် တည်ဆောက်တဲ့ ဖြစ်စဉ်ကို စတင်လုပ်ဆောင်ပေးမှာ ဖြစ်တယ်။ ထည့်ပေးထားတဲ့ တန်ဖိုးတွေက \_init\_ ဖန်ရှင်အတွက် ဖြစ်တယ်။ self နောက်က ပါရာမိတာတွေအတွက်ပဲ ထည့်ပေးရပြီး self အတွက် တန်ဖိုးထည့်မပေးရပါဘူး (self ဟာ ကလပ်စ်ရေးတဲ့သူအတွက် သီးသန့်ဖြစ်ပြီး ကလပ်စ်ဘေးဒီထဲမှာပဲ အသုံးပြုရတာပါ)။ ဒီတန်ဖိုးတွေက လက်ရှိအေးသာ့ဂျက် မေရီရောဘဲလွှာတွေရဲ့ ကန်းတို့ တန်ဖိုးတွေဖြစ်သွားမှုပါ

```

self.holder = holder
self.acc_number = acc_number
self.balance = balance

```

ဒီသုံးကြောင်းအရ လက်ရှိအေးသာ့ဂျက်ရဲ့ holder က 'Amy' ဖြစ်ပါမယ်။ လက်ရှိအေးသာ့ဂျက် balance နဲ့ acc\_number က Decimal('350\_000.00') နဲ့ '0086-6002-2233' အသီးသီး ဖြစ်ပါမယ်။ ပုံ

## (၉.၁) မှာလို မြင်ကြည့်နိုင်ပါတယ်

holder	Amy
accountNumber	0086-6002-2233
balance	350,000.00

(Account)

## ပုံ ၉.၁ အေမ့် Account အော်ဂျက် (acc3)

ဒီအော်ဂျက်ကို ရည်ညွှန်းအသုံးပြုနိုင်ဖို့ဆိုရင် ထုံးစံအတိုင်း ပေရီရေဘဲလ်နဲ့ အဆိုင်းမနဲ့ လုပ်ထားဖို့ လိုပါတယ်။

```
acc3 = Account('Amy', '0086-6002-2233', Decimal('350_000.00'))

print(acc3.holder)      # Amy
print(acc3.acc_number)  # 0086-6002-2233
print(acc3.balance)    # 350000.00
```

ဒီနေရာမှာ self အကြောင်းကို ရှင်းပြန့် လိုလာပြီ။ လက်ရှိအသုံးပြုနေတဲ့ အေမ့် Account အော်ဂျက်ဟာ 'လက်ရှိအော်ဂျက်' self ပဲ ဖြစ်တယ်။ ဒါကြောင့် အခုက္ခာမှာ acc3 နဲ့ self နှစ်ခုလုံးဟာ 'လက်ရှိအော်ဂျက်' (တစ်ခုတည်း) ပဲ။ acc3.holder နဲ့ self.holder နှစ်ခုလုံးက လက်ရှိအော်ဂျက်ရဲ့ holder ကို ဆိုလိုတာ ဖြစ်တယ်။ holder က 'Amy' ပါ။ ဒီသဘောအတိုင်း acc3.balance ရော့ self.balance ပါ လက်ရှိအော်ဂျက်ရဲ့ balance ကို ဆိုလိုတာ။ Decimal("350\_000.00") ဖြစ်ပါမယ်။ လက်ရှိအော်ဂျက်က ပြောင်းသွားရင်ရော ဘယ်လိုဖြစ်မလဲ။

```
acc1 = Account('Kathy', '0086-6002-1111', Decimal("100000.00"))
```

အခုလက်ရှိ အော်ဂျက်က ကော့သို့ Account အော်ဂျက် ဖြစ်သွားပြီ။ ဒီအချိန်မှာ self.holder က 'Kathy' ပဲ acc1.holder လည်း ဒါပဲဖြစ်မယ်။ acc\_number နဲ့ balance တို့ကိုလည်း အလားတူစဉ်းစားရမှာ ဖြစ်တယ်။ '0086-6002-1111' နဲ့ Decimal("100000.00") ဖြစ်ပါမယ်။ အခုလက်ရှိ အော်ဂျက်ဟာ သီးခြားတည်ရှုနေတဲ့ အော်ဂျက်တစ်ခုဖြစ်ပြီး အခုလို မြင်ကြည့်ရမှာပါ

holder	Kathy
accountNumber	0086-6002-1111
balance	100,000.00

(Account)

## ပုံ ၉.၂ ကော့သို့ Account အော်ဂျက် (acc1)

ဆိုလိုတာက အေမ့် Account အော်ဂျက်မှာ အကောင့်ပိုင်ရှင်၊ နံပါတ်နဲ့ လက်ကျန်ငွေအတွက် သူ

ကိုယ်ပိုင် ဖေရီရေဘဲလ်သုံးခု ရှိနေမှုဖြစ်ပြီး ကေသီ Account အော်ဂျက်ကလည်း သူဟာနဲ့သူ သီးခြား သုံးခု ရှိနေမှုပါ။

အခုလောက်ဆိုရင် self ရဲ့ သဘောကို နားလည်လောက်ပါပြီ။ ကလပ်စ်သတ်မှတ်တဲ့အခါ အော်ဂျက်တစ်ခုစီမှာ သီးခြားကိုယ်ပိုင် ပါရှိမဲ့ ဖေရီရေဘဲလ်တွေကို dot အမှတ်အသားအသုံးပြုပြီး self နဲ့ ညည်နှင့်ရပါတယ်။ deposit နဲ့ withdraw ကို ဆက်ကြည့်ရအောင်။

```
# File: account.py
class Account:
    ... # __init__ စိတ်ထပ်မြှုပ်နည်းချုပ်ထဲတယ်
    def deposit(self, amt):
        if amt <= Decimal(0.00):
            raise ValueError('Invalid amount for deposit!')
        self.balance += amt

    def withdraw(self, amt):
        if amt > self.balance:
            raise ValueError('Not enough balance!')
        self.balance -= amt
```

ဒီဖန်ရှင်တွေမှာလည်း ပထမ ပါရာမီတာက self ဖြစ်နေတာ တွေ့ရမှာပါ။ လက်ရှိအော်ဂျက်ရဲ့ အခြေ အနေအောက်မှာ အလုပ်လုပ်ပေးမဲ့ ဖန်ရှင်ရဲ့ ပထမ ပါရာမီတာက self ဖြစ်ရပါမယ်။ ဒီဖန်ရှင်တွေက အော်ဂျက်အပေါ်မှာ လုပ်ဆောင်လို့ရတဲ့ အော်ပရေးရှင်းတွေပါပဲ။ Account instance တွေအပေါ်မှာ deposit နဲ့ withdraw လုပ်ဆောင်လို့ ရမှာဖြစ်တယ်

```
acc3.withdraw(Decimal('50_000.00'))
acc1.deposit(Decimal('25_000.00'))
```

ပထမတစ်ခုက အော့မှု အကောင်ကနေ ငွေထုတ် (withdraw) လုပ်တာပါ။ သိပြီးဖြစ်တဲ့အတိုင်း acc3 နဲ့ self ဟာ လက်ရှိအော်ဂျက် ဖြစ်တယ်။ withdraw ကို ကြည့်ရင် amt က လက်ရှိအော်ဂျက်ရဲ့ balance ထက် များနေရင် exception raise လုပ်ထားတယ်။ ရှိတဲ့လက်ကျို့ငွေထက် ပိုထုတ်လို့ မရ သင့်ဘူး။

```
self.balance -= amt
```

ကတော့ လက်ရှိအော်ဂျက်ရဲ့ balance ကနေ amt နှုတ်လိုက်တာပါ။ အော့မှု အကောင်မှာ လက်ကျို့ငွေ 300,000.00 ဖြစ်သွားမယ်။ ဒုတိယတစ်ကြောင်းက ကေသီ အကောင့်ကို ငွေသွင်း (deposit) လုပ်တာ။ ဒီတစ်ခါကျတော့ လက်ရှိအော်ဂျက်က ကေသီအကောင့်ပေါ့။ self.balance += amt က ကေသီအကောင့် သို့ သက်ဆိုင်တဲ့ အော်ဂျက်ရဲ့ balance ကို amt ပမာဏ ပေါင်းပေးတာ။ ဒါကြောင့် ကေသီအကောင့်လက်ကျို့ငွေ 125,000.00 ဖြစ်သွားပါမယ်။

```
print(acc3.balance) # 300000.00
print(acc1.balance) # 125000.00
```

holder	Amy	holder	Kathy
accountNumber	0086-6002-2233	accountNumber	0086-6002-1111
balance	<b>300,000.00</b>	balance	<b>125,000.00</b>

**(Account)**

(က) အေမ့် Account အော်ဂျက် (acc3)

(e) ගෙවීම් Account වෙනත් පුරු (acc1)

ସଂ ଟ.୨ After withdraw and deposit on acc3 and acc1 respectively

## Attributes, Methods and Members

အော်ဂျက်မှာပါဝင်တဲ့ အချက်အလက်ကို attribute လိုခေါ်ပြီး အော်ဂျက်ပေါ်မှာ လုပ်ဆောင်နိုင်တဲ့ အော်ပရေးရှင်းကို method (မက်သဒ်) လိုခေါ်တယ်။ အော်ဂျက်တစ်ခုမှာ ပါဝင်တဲ့ attribute တော့နဲ့ method တော့ နစ်မျိုးလုံးကိုခြုံပြီး members တော့လို့ ခေါ်ပါတယ်။

C.J Access Control

Account instance တွေမှာ ပြဿနာတရာ့၊ ရှိနေပါတယ်။ withdraw နဲ့ deposit မက်သင်တွေက ငွေရှိတာထက် ပိုတိတ်လို့မရအောင်၊ ငွေသွင်းရင်လည်း ဖြစ်သင့်တဲ့ပမာဏပဲ သွင်းအောင် ကာကွယ်ထားတာ တွေ့ရမှာပါ။ ဒါပေမဲ့ ခက်တာက အသုံးပြသူတွေက ဒီအကာအကွယ်တွေကို ကျော်ပြီး အခုံလို့လုပ်လို့ရနေပါလိမ့်မယ်

```
acc1.balance = Decimal('50_000.00') #  
acc1.acc_number = None #
```

လက်ကျွန်ုင်ဟော အနှစ်တန်ဖိုး ဖြစ်လိုမရသင့်ဘူး။ ဘဏ်အကောင့်တစ်ခုမှာ အကောင့်နံပါတ်ကလည်း ရှိကိရိရပါမယ်။ ဒီလိုသာ ထင်သလို တန်ဖိုး ထည့်လိုရနေရင် ပြဿနာပဲ။ အခုလိုဆိုရင် acc1 အော့သုဂ္ဂက်ဟာ invalid state (မမှန်ကန်တဲ့ အနေအထား) ဖြစ်သွားမှုပါ။ အသုံးပြုသူက သေချာစဉ်းစားပြီး မဖြစ်သင့်တာ မလုပ်နဲ့ပေါ့လို့ စောဒကတက်စရာတော့ ရှိပါတယ်။ လျှပ်စစ်မီး ပလပ်ပေါက်ထဲ သတ္တုရွောင်းထိုးထည့်လို့ ခါတ်လိုက်ရင် user fault လိုပြောတာက မှားတော့မမှားပါဘူး။ ဒါပေမဲ့လည်း မီးပလပ်ပေါက်ထုတ်လုပ်သူအနေနဲ့ အကာတ်နိုင်ဆုံး အဲဒီလိုလုပ်လို့မရအောင် အန်ရှုယ်နည်းနိုင်သွားနည်းအောင် ဒီဇိုင်း

## ၁၆၂

ပြုလုပ်ထားသင့်ပါတယ်။

အော့သ်ဂျက် attribute တွေကို တိုက်ရှိက် အသံးမပြုစေချင်တဲ့အခါ Python စေလေထုံးစံက attribute နံမည်ရေးမှာ \_ (underscore) ထည့်ပေးပါတယ်။

```
class Account:  
    def __init__(self, holder, acc_number, balance):  
        self._holder = holder  
        self._acc_number = acc_number  
        self._balance = balance
```

Attribute နံမည်တွေကို \_holder, \_acc\_number စသည်ဖြင့် - နဲ့စထားတာ သတိပြုပါ။ ဒီလိုနံမည် တွေဆိုရင် ကလပ်စံအသံးပြုသူ (အတိကျပြောရင် ကလပ်စံကနေ ဖန်တီးယူတဲ့ instance အသံးပြုသူ) အနေနဲ့ 'တိုက်ရှိက် အသံးမပြုရ' လို့ အမြိုက်ရတယ်။ ဒါက Python ပရိုဂရမ်မာ အများစု လက်ခံထားတဲ့ ဓမ္မလုပ်စံတစ်ခုသာ ဖြစ်တယ်။ အားလုံးက - နဲ့စတဲ့ attribute (မက်သာတွေလည်းပါတယ်) တွေဆိုရင် တိုက်ရှိက်ယူမသုံးသင့်ဘူးလို့ နားလည် လက်ခံထားတယ်။ ဒါပေမဲ့ အခြား programming language တွေလို့ လုံးဝသုံးလို့မရအောင် ကန့်သတ်လို့တော့ Python မှာ မရနိုင်ဘူး။ သုံးချင်သပဆိုရင်လည်း သုံးလို့တော့ရတယ်။ ဒါပေမဲ့ ဖြစ်လာမဲ့အကျိုးဆက်ဟာ အသံးပြုသူရဲ့ တာဝန်သာဖြစ်တယ်။ ဒါကြောင့် ဒီလိုလုပ်လို့ ရတယ်

```
acc1._balance = Decimal('-50_000.00')      # မလုပ်သင့်ပါ
```

ဒါပေမဲ့ ထုံးစံကိုဖောက်ဖျက်တာ ဖြစ်တယ်။ မလုပ်သင့်တဲ့ အရာပေါ့။

ဒီလို ဘဏေကြောင့် မသုံးသင့်လ အခြားအကြောင်းအရင်း တစ်ခုလည်း ရှိပါသေးတယ်။ ကလပ်စံပိုင်ရှင် က သူကလပ်စံ implementation details တွေကို အကြောင်းအမျိုးမျိုးကြောင့် ပြင်ဆင် ပြောင်းလဲရလေ့ရှိတယ်။ ဥပမာ ပိုမြန်တဲ့ ဒါမှာဟုတ် မမ်မိုရှိ အစားသက်သာစေမဲ့ နည်းလမ်းတွေကို ပြောင်းလဲ အသံးပြုရနိုင်တယ်။ ဘဏ်အကောင့်တွေမှာ ငွေသွင်းငွေထဲတ် စာရင်းကို အစဉ်အတိုင်း သိမ်းထားဖို့ လိုတယ်ဆုံးပါစိုး။ List တစ်ခုထဲမှာ သွင်းငွေပေမာဏကို အပေါင်းတန်ဖိုး၊ ထုတ်ငွေပေမာဏကို အနှုတ်တန်ဖိုးနဲ့ သိမ်းထားနိုင်တယ်။ တစ်သိန်းသွင်းပြီး နှစ်သောင်းခွဲ ထပ်သွင်းတယ်။ နောက်တော့ သုံးသောင်းခွဲ ပြန်ထုတ်တယ် ဆုံးပါစိုး။ List ထဲမှာ အခုလို ရှိနေရပါမယ်

```
[100000.00, 25000.00, -35000.00]
```

ဒီတန်ဖိုးတွေကို ပေါင်းလိုက်ရင် လက်ရှိလက်ကျန်ငွေ balance ကိုရမှာပါ။ ဒီလိုဆိုရင် သီးသန့် attribute တစ်ခုနဲ့ balance ကို သိမ်းထားဖို့ မလိုအပ်တော့ဘူး။ ငွေသွင်းငွေထဲတ် စာရင်းကနေ တွေက ယူနိုင်တယ်။ အဖက်ဖက်ကနေ စဉ်းစားသုံးသပ်ပြီး ပိုင်ရှင်က သူကလပ်စံကို အခုလို update လုပ်ဖိုး ဆုံးဖြတ်ချက် ချုနိုင်ပါတယ်

```
from decimal import *  
  
class Account:  
    def __init__(self, holder, acc_number, balance):  
        self._holder = holder  
        self._acc_number = acc_number  
        self._transactions = [balance]  
  
    def deposit(self, amt):
```

```

if amt <= Decimal(0.00):
    raise ValueError('Invalid amount for deposit!')
    self._transactions.append(amt)

def withdraw(self, amt):
    if amt > self.balance:
        raise ValueError('Not enough balance!')
    self._transactions.append(-amt)

@property
def balance(self):
    return sum(self._transactions)

```

လောလောဆယ် အသေးစိတ်တွေက သိပ်အရေးမကြိုးဘူး။ `_balance` ဖြတ်လိုက်တာကိုသာ အခိုက အာရုံထားပါ။ ဒီအခါ သူကို တိုက်ရှိက် သုံးထားတဲ့ ကုဒ်အေးလုံး (ဥပမာ `acc1._balance`) ကလပ်စ် ဟူရင်း အသစ်ကို update ယူပြီးသုံးရင် ပြဿနာတက်သွားမှာပါ။ ပိုကောင်းတဲ့ ဟာရှင်းအသစ်ကို သုံးချင်တယ် ဆိုရင် တိုက်ရှိက်သုံးထားတဲ့နေရာတွေ အားလုံးကို လိုက်ပြင်ရပါလိမ့်မယ်။ ကလပ်စ်ပိုင်ရှင်က တမင်တကာ ဖုံးကွဲယ်ထားတဲ့ implementation details တွေကို တိုက်ရှိက်မသုံးရင် ဒီလိုပြဿနာမျိုးတွေ မဖြစ်နိုင်ဘူးပေါ့။

ဒီလောက်ဆိုရင် access control ဘာကြောင့် အရေးကြိုးသလဲ နားလည် သဘောပါက်လောက်ပါပြီ။ ကလပ်စ် ပိုင်ရှင်ရော အသုံးပြုသူတွေ အတွက်ပါ ဂရုစိုက်စဉ်းစားရမဲ့ ကိစ္စဖြစ်ပါတယ်။ ဘယ်သူမ ဆို တရားဝင် အသုံးပြုခွင့် ပေးထားတဲ့ `members` ကို ကလပ်စ်ရဲ့ interface သို့ API (*Application Programming Interface*) လိုခေါ်ပါတယ်။ ကလပ်စ်ပိုင်ရှင်က စဉ်းစားဆင်ခြင် သုံးသပ်ပြီး ကလပ်စ် API က ဘယ်လိုဖြစ်သင့်လဲ တိတိကျကျ သတ်မှတ်ပေးရတာပါ။ ဖုံးကွဲယ်ထားသင့်တဲ့ implementation details တွေကို ပေးသိ/သုံး မိတာဟာ နောင်တစ်ချိန် ကလပ်စ်ကို ပြင်ဆင်မွမ်းမံဖို့ အဟန်အတား ဖြစ်စေပါတယ်။ အသုံးပြုသူတွေကလည်း မသိ/မသုံးစေချင်တဲ့ internal တွေကို အသုံးပြုခြင်းဖြင့် စဉ်းစားလောက်ဖို့ လိုပါတယ်။

## ၉.၃ Composition or Has-a Relationship

အခုတွေခဲ့တဲ့ `Account` instance တွေက ပိုင်ရှင် နံမည်ကိုပဲ attribute အနေနဲ့ သိမ်းတယ်။ နံမည် အပြင် အကောင့် ပိုင်ရှင် မှတ်ပုံတင်နံပါတ် (NRIC)၊ မွေးသက္ကရာဇ်၊ နေရပ်လိပ်စာတိုကိုလည်း ဘဏ်တွေက မှတ်သား သိမ်းဆည်းရပါတယ်။ ပရိုဂရမ်ထဲမှာလည်း ဒီအချက်အလက်တွေကို ထင်ဟပ်ဖော်ပြန့် လိုပါမယ်။ `Account` ကလပ်စ်မှာ ဒီအချက်အလက်တွေအတွက် attribute အသစ်တွေ ထပ်ထည့်ရမလား၊ ဒါမှမဟုတ် ပိုင်ရှင်တစ်ယောက်ချင်းကို အော့ဘုရာ်အနေနဲ့ ဖော်ပြရမလား စဉ်းစားစရာ ဖြစ်လာပါတယ်။ အပြင်မှာရှိတဲ့ ဘဏ်အကောင့်ကို `Account` instance နဲ့ ကိုယ်စားပြုတယ်ဆိုရင် အပြင်က အကောင့်ပိုင်ရှင်ကို `Holder` instance နဲ့ ဖော်ပြတာဟာလည်း ကျိုးကြောင်းဆီလျှော်တယ်လိုပဲ ယူဆရမှာပါ။ အော့ဘုရာ် ဖြစ်သင့်/မသင့် စဉ်းစားဆုံးဖြတ်တဲ့အခါ အခြေခံအကျဆုံး မေးခွန်းတစ်ခုကတော့ မိမိ ဖော်ပြလိုတဲ့ အရာဟာ attributes တွေအပြင် ဘယ်လို methods တွေ ပါဝင်သင့်လဲ ဆိုတာပါ။ Attribute တွေပဲ ရှိပြီး method တစ်ခုမှ မရှိရင် အော့ဘုရာ်အနေနဲ့ မရှုမြင်သင့်ဘူး။ `Holder` instance တွေ အပေါ်မှာ လုပ်ဆောင်လို့ ရသင့်တဲ့ method တာချို့ ခေါင်းထဲပေါ်လာဖို့ သိပ်မခက်သင့်ဘူး။ အခြားလည်း ရှိအံးမှာပါ။

```
|holder1.change_address(new_address)
```

## ၁၆၂

အပြင်မှုလည်း လိပ်စာပြောင်းတာ မကြာခဏဖြစ်လေ့ရှိပါတယ်။ Attributes ရော methods တွေပါ ရှိမယ်ဆိုရင် အော့သ်ဂျက်အနေနဲ့ စဉ်းစားနိုင်ပါတယ်။ ဒါဆိုရင် ကလပ်စံသတ်မှတ် ရပါမယ်

```
class Holder:  
    def __init__(self, name, dob, nric, gender):  
        self._name = name  
        self._dob = dob  
        self._nric = nric  
        self._gender = gender  
  
    # ...
```

ဒီလောက်နဲ့ အော့သ်ဂျက် ဖန်တီးယူလို ရပါပြီ (ကျွန်တဲ့ ပါသင့်တာတွေ ခဏနေရင် ထပ်ဖြည့်ပါမယ်)။ ဒီ စက်ရှင်ရဲ့ အခိုက အကြောင်းအရာကို အရင်ကြည့်ရအောင်။ ဘက်အကောင့်နဲ့ အကောင့်ပိုင်ရှင် ဆက်စပ် နေတောက် ဖော်ပြချင်ရင် ဒီလိုပါ

```
holder1 = Holder('Kathy',  
                  date(1990, 3, 10),  
                  'MaRaNa (N) 1343232',  
                  'F')  
acc1 = Account(holder1, '0086-6002-1111', Decimal('100_000.00'))
```

အကောင့်ပိုင်ရှင် ကေသ့ကို ကိုယ်စားပြုတဲ့ အော့သ်ဂျက် ဖန်တီးယူမယ်။ ပြီးတော့ ကေသ့အကောင့်မှာ အဲ ဒီအော့သ်ဂျက်ကို ထည့်ပေးလိုက်ရမဲ့။ Account မှာ Holder က attribute တစ်ခုအနေနဲ့ ရှိနေမှာပါ။ အော့သ်ဂျက်နှင့် အခုလုံ ဖွံ့ဖြိုးထားတာကို composition လို ခေါ်ပါတယ်။ Composition ဟာ ပိုင်ဆိုင်ခြင်း (သို့) ရှိခြင်း ကိုလည်း ဖော်ပြတယ်။ ‘ဘက်အကောင့်တစ်ခုမှာ အကောင့်ပိုင်ရှင်တစ်ယောက် ရှိတယ်’၊ ‘ကားတစ်စီးမှာ အင်ဂျင်တစ်လုံး ပါရှိတယ်’။ ဒီလိုဆက်စပ်မှုမျိုးကို ဖော်ပြတဲ့ အော့သ်ဂျက်နှစ်ခု ဟာ has-a relationship ရှိတယ်လို ပြောလေ့ရှိတယ်။

အော့သ်ဂျက်တစ်ခုဟာ တစ်ခုထက်ပိုတဲ့ အခြား အော့သ်ဂျက်တွေနဲ့ has-a relation ရှိနိုင်တေပါ့။ ဥပမာ အကောင့်ဖွင့်တဲ့ ဘက်ခွဲကို အော့သ်ဂျက်အနေနဲ့ ယူဆမယ်ဆိုရင်

```
class Branch:  
    def __init__(self, name, address):  
        self._name = name  
        self._address = address  
    # ...  
  
class Account:  
    def __init__(self, holder, acc_number, balance, branch):  
        # ... အခြား attributes တွေရှိမယ် မပြထားတော့ဘာ  
        self._branch = branch
```

Account နဲ့ သူ့ခဲ့ Branch ကို ချိတ်ဆက်မယ်ဆိုရင်

```
# ... holder1 ရှိမယ် မပြထားတော့ဘာ  
branch1 = Branch('Mandalay', '232A, 62 St, Chanayetharsi')  
acc1 = Account(holder1,
```

```
'0086-6002-1111',
Decimal('100_000.00'),
branch1)
```

### Types of Composition and Sharing

ဘက်ခဲ့တစ်ခုတည်းမှာ ဖွင့်ထားတဲ့ ဘက် အကောင့်တွေဟာ Branch အော်ဂျက်တစ်ခုတည်းကိုပဲ sharing လုပ်ထားပြီး အသုံးပြန်ပါတယ်။

```
branch1 = Branch('Mandalay', '232A, 62 St, Chanayetharsi')
acc1 = Account(holder1,
                '0086-6002-1111',
                Decimal('100_000.00'),
                branch1)
acc2 = Account(holder2,
                '0086-6002-1111',
                Decimal('100_000.00'),
                branch1)
```

acc1 နဲ့ acc2 နှစ်ခုလုံးက branch1 အော်ဂျက်တစ်ခုတည်းကိုပဲ မျှသုံးထားတာပါ။ လူတစ်ယောက် တည်းက အကောင့် နှစ်ခု ဖွင့်ထားရင် Holder အော်ဂျက်တစ်ခုတည်းကို မျှသုံးပါမယ်။

```
holder1 = Holder('Kathy',
                  date(1990, 3, 10),
                  'MaRaNa (N) 1343232',
                  'F')
acc1 = Account(holder1, '0086-6002-1111', Decimal('100_000.00'))
acc5 = Account(holder1, '0086-6002-5522', Decimal('500_000.00'))
```

Has-a relationship ကို အမျိုးအစားတွေခဲ့ကြည့်ရင် one-to-one, one-to-many, many-to-many စသည့်ဖြင့်တွေရပါတယ်။ Car နဲ့ Engine ဟာ one-to-one ဖြစ်ပါမယ်။ Student နဲ့ Course (သို့) Actor နဲ့ Film ဆိုရင်တော့ many-to-many relationship ပေါ့။ One-to-many ကို Bank နဲ့ Branch, House နဲ့ Resident တို့အကြား တွေရမှာပါ။

Bank နဲ့ Account တို့အကြား one-to-many relationship ကို ဒီအခန်းနောက်ဆုံးမှာ တွေ့ရပါမယ်။ One-to-one အတွက် Car နဲ့ Engine, many-to-many အတွက် Student နဲ့ Course ကို ဖော်ပြထားတာ လေ့လာကြည်ပါ။

```
class Engine:
    def __init__(self, horsepower):
        self._horsepower = horsepower

class Car:
    def __init__(self, model, engine):
        self._model = model
        self._engine = engine
```

## ၁၆။

```
# Creating an engine object
my_engine = Engine(horsepower=200)

# Creating a car object with the engine
my_car = Car(model="Toyota Camry", engine=my_engine)

class Student:
    def __init__(self, name):
        self.name = name
        self.courses = []

    def enroll(self, course):
        self.courses.append(course)

class Course:
    def __init__(self, name):
        self.name = name
        self.students = []

    def add_student(self, student):
        self.students.append(student)

# Creating student objects
sandar = Student("Sandar")
waiyan = Student("Waiyan")

# Creating course objects
math_course = Course("Mathematics")
science_course = Course("Science")

# Enrolling students in courses
sandar.enroll(math_course)
sandar.enroll(science_course)
waiyan.enroll(math_course)

# Adding students to courses
math_course.add_student(sandar)
math_course.add_student(waiyan)
science_course.add_student(sandar)
```

ကျောင်းသူ/သား တစ်ယောက် တက်ရောက်ဖို့ စာရင်းပေးထားတဲ့ course တွေအတွက် course တစ်ခုကို တက်ရောက်မဲ့ ကျောင်းသူ/သား တွေအတွက် list ကို သုံးထားတာ ဂရုပြုကြည့်ပါ။ Has-a

relationship မှာ many ဘက်ခြမ်းကို ဖော်ပြန့်အတွက် အော့သုဂ္ဂက်တွေ တစ်စုတစ်စည်းတည်း ထားနိုင်တဲ့ list, set, dictionary စတေတွေ သုံးရပါတယ်။

## ၆.၄ Properties

Access control စည်းကမ်းအတိုင်း လိုက်နာရမယ်ဆိုရင် Holder ရဲ့ attributes တွေကို တိုက်ရှိက်မသုံးသင့်ဘူး။ ဒါဆို attributes တွေကို သုံးလိုအပ်ရင် ဘယ်လိုလုပ်ရမလဲ မေးစရာ ရှိပါတယ်။ ကလပ်စိုင်ရင် အနေနဲ့ attribute ကို မက်သဒ်ကနောက်ဆင့် အသုံးပြုလိုရအောင် လမ်းဖွင့်ပေးထားနိုင်ပါတယ်။ မဖြစ်သင့်တာ မဖြစ်အောင် မက်သဒ်မှာ စိစစ်လိုရတယ်။

```
class Holder:
    def __init__(self, name, dob, nric, gender):
        self._name = name
        self._dob = dob
        self._nric = nric
        self._gender = gender

    def get_name(self):
        return self._name

    def set_name(self, name):
        if name.strip() == '':
            raise ValueError('Name cannot be empty!')
        self._name = name.strip()
```

set\_name မှာ နံမည်ကို empty string မဖြစ်အောင် စစ်ထားတယ်။ အသုံးပြုသူအနေနဲ့ နံမည်ကို ယူကြည့်တာ၊ ပြောင်းတာ လုပ်မယ်ဆိုရင် မက်သဒ်ကိုပဲ သုံးရမှုပါ။ ဥပမာ

```
print(holder1.get_name())
holder1.set_name('Amy Lynn')
holder1.set_name('')           # ValueError 張စ်မယ်
```

(- နဲ့ မစရင် API, သုံးလိုရတယ်)။

အခြား attribute တွေလည်း လိုအပ်ရင် အလားတူ မက်သဒ်တွေနဲ့ API ထုတ်ပေးထားလိုရပါတယ်။ ဒီနည်းလမ်းအပြင် Python မှာ managed attributes ဆိုတာလည်း ရှိပါလေးတယ်။ Managed attributes တွေကို properties လိုလည်း ခေါ်တယ်။ Managed attribute သုံးထားတဲ့ ဥပမာကို လေ့လာကြည့်ပါ

```
class Holder:
    def __init__(self, name, dob, nric, gender):
        self._name = name
        self._dob = dob
        self._nric = nric
        self._gender = gender

    @property
```

```

def name(self):
    return self._name

@name.setter
def name(self, name):
    if name.strip() == '':
        raise ValueError('Name cannot be empty!')
    self._name = name

@property
def dob(self):
    return self._dob

@dob.setter
def dob(self, value):
    self._dob = value

# ...

```

@ သက်တန့် စထားတဲ့ @property, @name.setter စသည်ဖြင့် တွေ့ရပါတယ်။ Property decorator လိုအပါတယ်။ ဒါလေးတွေက နောက်ကုတ်မှာတော့ ဖန်ရှင်တစ်ခုပါပဲ။ property() ဖန်ရှင်က သူတို့ရဲ့ နောက်ကုတ်ကနေ အလုပ်လုပ်ပေးမှာပါ။ လောလောဆယ် အသေးစိတ်တွေ ခဏထားလိုက်ပြီး အသုံးပြန်ည်းကို အဓိကထား ကြည့်ရအောင်။ Decorator ရဲ့ အောက်မှာ ရှိနေတာက ပုံမှန်မက်သဒ်တွေ ပါပဲ။ Property decorator က အဓိကဘာလုပ်ပေးလဲ။ မက်သဒ်တစ်ခုကို ပုံမှန် attribute ဖော်ရေး ဘဲလုလို သုံးလို့ရအောင် လုပ်ပေးတာပါ။ ဥပမာ name မက်သဒ်ကို ခေါ်ချင် holder1.name() လို့ ရေးနေစရာ မလိုဘူး။ holder1.name လို့ ရေးရုံပဲ။ တကယ်တမ်း နောက်ကုတ်မှာ အလုပ် လုပ်တာကတော့ မက်သဒ်ကပဲ လုပ်သွားမှာပါ။ name အတွက် @name.setter, dob အတွက်ဆိုရင် @dob.setter စသည်ဖြင့် ဖြစ်ရပါမယ်။

အဆိုင်းမန် လုပ်ရင်လဲ ဒီသဘောပဲ။ holder1.name = 'Amy Lynn' လို့ ရေးရင် @name.setter decorator နဲ့ မက်သဒ်က အလုပ်လုပ်သွားမှာ ဖြစ်တယ်။ holder1.name('Amy Lynn') မက်သဒ်ခေါ်တာနဲ့ သဘောတရား တူမယ်။ ဒါကြောင့် အခုလို စမ်းကြည့်ရင်

```
holder1.name = ''
```

ValueError exception ဖြစ်ပါလိမ့်မယ်။ Property decorator သုံးထားတဲ့အတွက် dob ကိုလည်း attribute အနေနဲ့ သုံးလို့ရပါတယ်။

```
println(holder1.dob)
holder1.dob = date(1990, 3, 11)
```

Python ပရိုဂရမ်မှာတွေက managed attribute အသုံးပြုတာကို Pythonic ပိုဖြစ်တဲ့ နည်းလမ်းလို့ ယူဆကြပါတယ်။ ဒါကြောင့် ရိုးရိုး get\_name, set\_name မက်သဒ်တွေနဲ့ attribute သုံးလေ့မရှိဘူး။ ဘိုင်နာအနေနဲ့ မက်သဒ်ဖြစ်သင့်တော့ကို attribute အနေနဲ့ မရှိမှင်ကြည့်မိမိ သတိပိုသင့်ပါတယ်။ withdraw နဲ့ deposit က attribute မဖြစ်သင့်ဘူး။ ဒါတွေက အော့သုတေသနများ အော့သုတေသနများ မှာ လုပ်ဆောင်လိုကူရတဲ့ အော်ပရေးရှင်းတွေ အနေနဲ့ပဲ ရှိမှင်ရမှာပါ။ Attribute ဆိုတာက အော့သုတေသနများ အချက်အလက်သာ ဖြစ်တယ်။ ဒီအချက်အလက်တွေကိုမှ လိုအပ်ရင် managed at-

tribute အနေနဲ့ အသုံးပြုခွင့် ပေးရတာဖြစ်တယ်။

### Derived Attributes/Properties

Holder ရဲ့ အသက်ကို မေရီရောဲလ်အနေနဲ့ ထားစရာမလိုဘဲ မွေးသက္ကရာဇ်ကနေ တွက်ယူလိုရတယ်။ ဒီလိုမျိုး attribute တွေကို derived attributes လို ခေါ်လှုရှိတယ်။ ငြင်းတို့အတွက် မေရီရောဲလ် ရှိဖို့ မလိုအပ်တာကြောင့် နေရာကုန် သက်သာမယ်။ အသုံးပြုတဲ့ အခါမှ လိုတဲ့ တန်ဖိုးကို တွက်ယူတာမိုလို အချိန် အကုန်အကျရှိမယ်။ တချို့ တကယ့်လက်တွေ့ အခြေအနေတွေမှာ နေရာနဲ့ အချိန် အကုန်အကျ ကို ဆန်းစစ်သုံးသပ်ပြီး တန်ဖိုးအနေနဲ့ သိမ်းထားမလား၊ ဒါမှုမဟုတ် တွက်ချက်ယူမလား ဆုံးဖြတ်ရတတ်ပါတယ်။ Derived attributes တွေက နေရာသက်သာပေမဲ့ အချိန်အကုန်အကျ ရှိတယ်လို့ အခြေခံ အဆင့်မှာ သိထားသင့်ပါတယ်။

```
class Holder:
    def __init__(self, name, dob, nric, gender: str):
        # ...
        self._dob = dob
        # ...

    @property
    def age(self):
        return age_today(self._dob)
```

တမျက်နှာ (၁၆၅) က Account ကလပ်စွဲ balance ကို derived attribute အနေနဲ့ တွေ့ရမှာပါ။ Transaction (ငွေအဝင်အထွက်ကို ဆိုလို) အကြိမ်အရေအတွက် များရင်များသလို balance တွက် ထုတ်တဲ့ အခါ အချိန်အကုန်အကျလည်း များမှာဖြစ်တယ်။

```
class Account:
    def __init__(self, holder, acc_number, balance):
        # ...
        self._transactions = [balance]
        # ...

    @property
    def balance(self):
        return sum(self._transactions)
```

### self အကြောင်း သိကောင်းစေရာ

self နဲ့ ပါတ်သက်ပြီး ထိုက်သင့်သလောက် ဖော်ပြခြားပါပြီ။ ဒီလောက်သိထားရင်ကို ဘိုင်နာအဆင့် အတွက် လုံလောက်ပါတယ်။ ဒါပေမဲ့ စိတ်ဝင်စားလို ပိုသိချင်တဲ့ သူတွေအတွက်ရော သိထားသင့်တဲ့ အတွက်ပါ တချို့ဟာလေးတွေ ထပ်ကြည့်ကြပါမယ်။

အော့ဘ်ဂျက်တစ်ခု စဖန်တီးတဲ့ အခါမှာ မမေ့မိုရီ ရော်ယာထဲမှာ သူအတွက် လိုအပ်တဲ့ နေရာ အရင်ပေးရတယ် (`__init__` အလုပ်မလုပ်ခင် ဒီကိစ္စကို အရင်လုပ်ရတာ)။ နေရာပေးပြီးသွားရင် အဲဒီအော့ဘ်ဂျက်ကို ရည်ညွှန်းလိုရမဲ့ reference ကို ရှုပါတယ် (reference ဆိုတာ အော့ဘ်ဂျက်ကို ရည်ညွှန်းတဲ့ မေရီရောဲလ်ကို ပြောတာပါပဲ)။ ဒီအဆင့်ပြီးရင် အကြမ်းထည်အဆင့် အော့ဘ်ဂျက်တစ်ခု ရဲ့ နေပါပြီ။ ဒီအော့ဘ်ဂျက်ကို initialization ဆက်လုပ်ရပါမယ်။ `__init__` ဖော်ရှင်က initialize လုပ်

ပေးရမဲ့ အော်ဂျက်ကို သိရပါမယ်။ ဒါကြောင့် `__init__` ခေါ်တဲ့အခါ ခုနက referene ကို `self` နေရာမှ Python က အလိုအလျောက် ထည့်ပေးသွားမှာပါ။

အော်ဂျက် အချေထည် (properly initialized object) တည်ဆောက်ပြီးသွားတဲ့အခါ သူ အတွက် reference ကို ပြန်လည်ရရှိမှာပါ။ ဒီ reference နဲ့ အော်ဂျက်ကို ဆက်လက် အသုံးပြုလို ရ တာဖြစ်တယ်။ ဥပမာ

```
| acc4 = Account('Waiyan', ...)
```

ဒါဟာ ပြန်ရတဲ့ reference ကို `acc4` နဲ့ ဖမ်းယူထားလိုက်တဲ့ သဘောပဲ။

`withdraw`, `deposit` မက်သဒ်တွေ ခေါ်တဲ့အခါမှာလည်း `self` နေရာမှ လက်ရှိအော်ဂျက်ကို Python က ထည့်ပေးတာ ဖြစ်တယ်။

```
| acc4.deposit(Decimal('50_000.00'))
```

အခုလို ခေါ်တဲ့အခါ `acc4` ကို `deposit` မက်သဒ် `self` နေရာမှ ထည့်ပေးမှာပါ။ ဒီလို transformation လုပ်ပေးတယ်လို ယူဆနိုင်တယ်

```
| acc4.deposit(acc4, Decimal('50_000.00'))
```

မှတ်ချက်။ ။ ။ ။ အတိအကျ ဒီလိုဖြစ်တယ်လို မဆိုလို။ တကယ့်တကယ်က `Account.deposit(ac4, Decimal('50_000.00'))` အဖြစ် ပြောင်းပေးတာပါ။ ဒါပေမဲ့ ဘိုင်နာအတွက် လက်ရှိ knowledge နဲ့ ဒါကိုနားလည်နိုင်ဖို့က စေလွန်းသေးတယ်။

### Instance Variables and Instance Methods

အော်ဂျက်တစ်ခုခိုက် သီး၏ပိုင်ဆိုင်တဲ့ ဖေရာ့ရောဘဲလွှာကို *instance variable* တွေလိုလည်း ခေါ်ပါတယ်။ *Instance attribute* ပြောရင်လည်း အဓိပ္ပာယ်တူတူပါပဲ။ အော်ဂျက်တစ်ခုခိုက် သီး၏ပိုင်ဆိုင်တဲ့ attribute တွေပေါ့။

*Instance method* တွေကတော့ လက်ရှိအော်ဂျက်ရဲ့ အခြေအနေအောက်မှာ အလုပ်လုပ်တဲ့ မက သဒ်တွေဖြစ်ပြီး instance variable တွေကို အသုံးပြုလေ့ရှိတယ် (`acc1.withdraw(amt)` မှာ `withdraw` ဟာ လက်ရှိအော်ဂျက် `acc1` အခြေအနေအောက်မှာ အလုပ်လုပ်တယ်)။ *Instance method* ဟာ လက်ရှိအော်ဂျက်နဲ့ တွဲဖက်လုပ်ဆောင်ရတာ ဖြစ်တာကြောင့် `self` ပါရမိတာ ပေါ့လို မရဘူး။

## ၉.၅ Class Attributes and Methods

*Class attribute* ဆိုတာ ကလပ်စ်တစ်ခုရဲ့ *instance* တွေအားလုံးနဲ့ ဆိုင်တဲ့ ဘုံးသုံး attribute တွေ ကို ဆိုလိုတာပါ။ *Class attribute* တွေကို ကလပ်စ်ဘော်ဒီ top level မှာ ကြော်လျှပါမယ် (မက သဒ်တွေရဲ့ အပြင်ဘက်၊ ကလပ်စ်အောက်မှာ တိုက်ရှိကြရမှာ၊ မက်သဒ်ထဲမှာ မဖြစ်ရဘူး)။ ဘုံးသုံးဆိုတဲ့ အတိုင်း လက်ရှိအော်ဂျက်နဲ့ပဲ ဆိုင်တာမဟုတ်တော့ဘူး။ ဒါကြောင့်မို့လို attribute ကြော်တဲ့အခါ `self.attribute_name` ပုံစံ မကြော်ရဘူး။

```
# File: class_attr_eg.py
class SomeClass:
    # class attributes
```

```

MAX_ID = 9999
cur_id = 0

def __init__(self):
    self._id = SomeClass.cur_id + 1
    SomeClass.cur_id += 1

@property
def id(self):
    return self._id

# create 3 instance for testing
smc1 = SomeClass()
smc2 = SomeClass()
smc3 = SomeClass()

# အေးလုံး 3 ထွက်တယ်
print(SomeClass.cur_id)          # ကလပ်စံ နံမည်နဲ့ သုံးတာ
print(smc1.cur_id)              # ကလပ်စံ instance သုံးတာ
print(smc2.cur_id)
print(smc3.cur_id)

print(smc3.MAX_ID)
print(SomeClass.MAX_ID)

```

MAX\_ID နဲ့ cur\_id က class attribute တွေပါ။ အသုံးပြုတဲ့အခါ *ClassName.attribute\_name* ပုံစံနဲ့ သုံးရမှာပါ။ ဒါပေမဲ့ ကလပ်စံ instance ကနေ သုံးလိုလည်း ရပါတယ်။ Instance တွေကို ကိုယ်စားပြုတဲ့ smc1, smc2 စတာတွေနဲ့ အသုံးပြုလိုရတာ တွေရပါမယ်။

`__init__` ထဲမှာ လက်ရှိအော်ဂျက်ရဲ့ `_id` ကို `cur_id` အပေါင်း တစ် ထည့်ထားတာ တွေရပါမယ်။ ပြီးတော့ `cur_id` တန်ဖိုးကိုလည်း တစ်တိုးလိုက်တယ်။ ပထမဆုံး SomeClass instance ခဲ့`id` က တစ် ဖြစ်ပါမယ်။ အော်ဘ်ဂျက်တစ်ခု ဆောက်လိုက်တိုင်း `cur_id` တန်ဖိုးက တစ်တိုးသွားမှာပါ။ အခုံး ဥပမာမှာ အော်ဘ်ဂျက် သုံးခုယူထားတဲ့အတွက် သုံး ဖြစ်သွားပါမယ်။

Attribute တစ်ခုတည်းကိုပဲ အော်ဘ်ဂျက်အားလုံးက မျှသုံးထားတာ ဖြစ်တဲ့အတွက် `print` ထဲတော်ကြည့်တဲ့အခါ ဘယ်အော်ဂျက်နဲ့ပြစ်ဖြစ် 3 ပဲထွက်တာ တွေရမှာပါ။ ဒါဟာ အော်ဘ်ဂျက်တစ်ခုစီက သီးသန်ပိုင်ဆိုင်တဲ့ attribute နဲ့ အဓိကကွာခြားချက်ပါပဲ။ ဒီအချက်ကို ကဲ့ကဲ့ပြားပြား နားလည်ဖို့ အရေးကြီးပါတယ်။ `id` ကို ထဲတော်ကြည့်ပါ

```

print(smc1.id)      # 1 ထွက်မှာပါ
print(smc2.id)      # 2
print(smc3.id)      # 3

```

Class attribute တွေကို အော်ဂျက် (ပိုတိကျအောင်ပြောရင် အော်ဂျက်ကို ကိုယ်စားပြုတဲ့ ပေါ်ရော်လု) နဲ့ ရော ကလပ်နံပည်နဲ့ပါ အသုံးပြုလို ရပေမဲ့ instance attribute တွေကိုတော့ ကလပ်နံမည်နဲ့ အသုံးပြုလို မရှိနိုင်ပါဘူး။ SomeClass ဥပမာမှာ အခုလို စမ်းကြည့်ရင်

```
print(SomeClass.id)
print(SomeClass._id)
```

ထူးဆောင်ရွက်တဲ့ တစ်ခုက <property object at 0x10dbf4bd0> ထုတ်နေပါတယ် (နောက်ဆုံးနံပါတ်က တစ်ခါနဲ့ တစ်ခါ တူမှာမဟုတ်ပါ)။ ဒုတိယတစ်ခုက attribute မရှိဘူးဆိုတဲ့ အယ်ရှုပြတယ်။

`var_name.attribute_name` ပုံစံနဲ့ class attribute အသုံးပြုတဲ့ အခါးလက်ရှိအော်ဂျက်ရဲ့ attribute လား၊ class attribute လား မကွဲပြားတော့ဘူး။ ဥပမာ

```
>>> dt1 = date(1990, 11, 30)
>>> print(dt1.min)
0001-01-01
```

ဒါကို ဖတ်တဲ့သူအနေနဲ့ `min` က class attribute လားဆိုတာ ရုတ်တရှုံး မသိနိုင်ပါဘူး။ `date` ရဲ့ documentation မှာ ကြည့်မှ `min` ဟာ class attribute လို့ ဖော်ပြထားတာ တွေ့ရမှာပါ။ ကလပ်နံမည်နဲ့ `date.min` ဆိုရင်တော့ class attribute ပဲ ဖြစ်ရပါမယ်။

### Class attribute နှင့် `self`

ကလပ် ဘော်ဒီထဲမှာ ငါးကလပ် ကိုယ်တိုင်ရဲ့ class attribute တွေကို `self.attribute_name` နဲ့ ရည်ညွှန်း အသုံးပြုနိုင်ပါတယ်။ ဒါပေမဲ့ attribute တန်ဖိုးကို သုံးလိုပဲ ရုံးဖြစ်ပြီး အဆိုင်းမန်လုပ်လို့ မရပါဘူး။ တကယ်လို့ အဆိုင်းမန် လုပ်ရင် လက်ရှိအော်ဂျက်မှာ (class attribute နဲ့ နံမည်တူ) ပေါ်ရော်လဲ အသစ်တစ်ခု ရှိသွားမှာဖြစ်တယ်။

```
class SomeClass:
    # class attributes
    MAX_ID = 9999
    cur_id = 0

    def __init__(self):
        # class attribute
        self._id = self.cur_id + 1
        # လက်ရှိအော်ဂျက်မှာ cur_id ကြော်လို့ ဖြစ်သွားမှာ၊ class attribute
        # ကို အဆိုင်းမန် လုပ်တာမဟုတ်ဘူး
        self.cur_id = self._id
```

## Class Methods

Class attribute နဲ့ ဆက်စပ်နေတဲ့ class method တွေကတော့ မက်သာဒ်ကို ခေါ်ဖို့အတွက် အော့သံ ဂျက်ရှိစရာ မလိုဘူး။ တစ်နည်းအားဖြင့် class method တွေက ကလပ်စံတစ်ခုရဲ့ အခြေအနေအောက် မှာ လုပ်ဆောင်တာ။ အဲဒီ class instance ရဲ့ အခြေအနေအောက်မှာ လုပ်ဆောင်တာ မဟုတ်ဘူး။ မက် သာဒ်ခေါ်တဲ့အခါ *ClassName.method\_name()* ပုံစံနဲ့ ခေါ်လိုရတယ်။

```
class Joker:
    # class attributes
    CLASS_NICK_NAME = 'NaiveIdiot'
    joker_ids = []
    last_id = 0

    # class method
    @classmethod
    def get_info(cls):
        return {'CLS_NICK_NAME': cls.CLASS_NICK_NAME,
                'NXT_JOKER_ID': cls.last_id + 1,
                'CUR_JOKERS_IDS': cls.joker_ids}

    def __init__(self, name):
        self._id = Joker.last_id + 1
        Joker.joker_ids.append(self._id)
        Joker.last_id = self._id
        self._name = name

jk1 = Joker('Deadly Bee')
jk2 = Joker('Happy Hi')

print(Joker.get_info())
```

Output:

```
{'CLS_NICK_NAME': 'NaiveIdiot', 'NXT_JOKER_ID': 3, 'CUR_JOKERS_IDS': [1, 2]}
```

get\_info က ကလပ် မက်သာဒ် ဖြစ်ပါတယ်။ @classmethod decorator က မက်သာဒ် တစ်ခုကို ‘ကလပ် မက်သာဒ် ဖြစ်ပါတယ်လို့’ ကြော်လှုံးတဲ့ အဓိပ္ပာယ်ပါ။ ပုံမှန် မက်သာဒ်ဆိုရင် ပထမဆုံး ပါရေမီတာ က လက်ရှိအော့သံဂျက်ကို ရည်ညွှန်းတဲ့ self ဖြစ်တယ်။ ကလပ် မက်သာဒ် ဆိုရင်တော့ အဲဒီအား ပထမဆုံး ပါရေမီတာက ‘လက်ရှိကလပ်’ ကို ရည်ညွှန်းတဲ့ cls ဖြစ်ရမှာပါ။ cls ပါရေမီတာနဲ့ class attribute တွေကို အသုံးပြုလို့ ရတယ်။ Joker ကလပ် get\_info မှာ class attribute တွေထုံး ထားတာကို တွေ့ရမှာပါ။ Class method ခေါ်တဲ့အခါ cls နေရမှာ လက်ရှိအသုံးပြနေတဲ့ ကလပ် ဝင်လာမှာပါ။ Python က အလိုအလျောက် ထည့်ပေးတာဖြစ်တယ်။

## ၉.၆ Static Methods

Static method တွေကတော့ ကလပ်ထဲမှာ သတ်မှတ်ထားပေမဲ့ လက်ရှိအော့သံ self ကိုရေ့ လက်ရှိကလပ် cls ကိုပါ အသုံးပြုစို့ မလိုအပ်တဲ့ မက်သာဒ်တွေပါ။ @staticmethod decorator နဲ့

## Built-in Decorators

Python မှာ `@property`, `@staticmethod`, `@classmethod` စတဲ့ built-in ပုံပြီးသား decorator တွေပါရှိတယ်။ Decorator အမျိုးအစားအလိုက် လိုအပ်တဲ့ အသွေးပြောင်းလဲမှု (transformation) တွေ Python က လုပ်ပေးသွားမှာ ဖြစ်တယ်။ ဥပမာ `@property` decorator က မက သုတေသနတဲ့ attribute လို့ သုံးလို့ရအောင် လုပ်ပေးတယ်။ `@staticmethod` နဲ့ `@classmethod` ကလည်း မက်သံတစ်ခုကို class method နဲ့ static method အနေနဲ့ အသုံးပြုလို့ရအောင် လိုအပ်တဲ့ ပြောင်းလဲမှုတွေ လုပ်ပေးမှာ ဖြစ်တယ်။

Decorator တွေရဲ့ နောက်ကွယ်မှာ အလုပ်လုပ်ပေးတာက ဖန်ရှင်တွေပါပဲ။ ကိုယ်ပိုင် decorator သတ်မှတ်လို့လည်း ရပါတယ်။ အတွေ့အကြား ပရီဂရမ်မာ ဖြစ်လာတော့မှ လေ့လာကြလေ့ရှိပြီး ဒီ စာအပ်မှာတော့ ငှါးတိုက် သီးသန့်ဖော်ပြုမှာ မဟုတ်ပါဘူး။

`__init__` ရတယ်။ ရိုးရိုး top level ဖန်ရှင်လို့ပဲ `cls` (သို့) `self` ပါရာမီတာ မလိုဘူး။ ငွေလွှာ ကိစ္စအတွက် `transfer` မက်သံကို static method အနေနဲ့ ဥပမာ ပြထားတာကြည့်ပါ။

```
class Account:
    def __init__(self, holder, acc_number, balance, branch):
        # ...
        self._balance = balance
        # ...
    # ...
    @staticmethod
    def transfer(from_acc, to_acc, amt):
        if amt > from_acc._balance:
            raise ValueError('Not enough balance!')
        from_acc._balance -= amt
        to_acc._balance += amt
```

ဒီ မက်သံ ခေါ်တဲ့အခါ class method လို့ပဲ ကလပ်စံမည့်နဲ့ ခေါ်လို့ရသလို instance နဲ့ ခေါ်ချင်လည်း ရတယ်။

```
Account.transfer(acc1, acc2, Decimal('50_000.00'))
acc1.transfer(acc2, Decimal('50_000.00'))
```

ကလပ်စံမည့်နဲ့ ခေါ်တာ ပို့ရှင်းပါတယ်။

မှတ်ချက်။ `transfer` မက်သံကို instance method အနေနဲ့ပဲ ရေးထားတာကို လာမဲ့ အသုံးချုပ်မှာ တွေ့ရပါမယ်။ Instance method ဆိုရင် ခေါ်တဲ့အခါ

```
acc1.transfer(acc2, Decimal('50_000.00'))
```

ဖြစ်ပါမယ်။

Static method ဟာ ဖိုင်တစ်ခုရဲ့ top level မှာ `__init__` တဲ့ ပုံမှန်ဖန်ရှင်တွေလို့ပဲ `cls` (သို့) `self` မပါတဲ့အတွက် ကလပ်စံအပြင်ကို ထုတ်လိုက်ရင် ရုံးရုံးဖန်ရှင် ဖြစ်သွားမှာပါ။

```
# @staticmethod decorator လည်း မလိုတော့ဘူး
```

```
def transfer(from_acc, to_acc, amt):
    if amt > from_acc._balance:
        raise ValueError('Not enough balance!')
    from_acc._balance -= amt
    to_acc._balance += amt

class Account:
    def __init__(self, holder, acc_number, balance, branch):
        # ...
        self._balance = balance
        # ...
    # transfer ମେ ଗଲାଯିବାକୁ ଅର୍ଥଦିଙ୍କରିବାକୁ
```

ဒါပေမဲ့ ငွေလွှာတယ်ဆိတာ ဘက်အကောင့်နဲ့ သက်ဆိုင်တဲ့ ကိစ္စဖြစ်တဲ့အတွက် Account ကလပ်စဲမှု static method အနေနဲ့ ရှိတာက ပိုပြီး သဘာဝကျပါတယ်။

မက်သဒ္ဓနှင့် attribute အမျိုးအစားများ အကြား အပြန်အလုန် အသုံးပြန်မှ

Instance method ም instance/class attribute ቁ instance/class/static method ተቃዋሚ  
အသုံးပූለንድሬትယ်။ Class method ቁ static method በတော့ instance attribute/method  
ቴቃዋሚ ወုံးလိုက်မှာ မဟုတ်ပါဘူး။ ၌အက်ပါ ယေားနှစ်ခုမှာ အပျိုးအစားအလိုက် အသုံးပූለንድ့ ရ/မရ ይခြေား  
ပြထားတာကို ကြည့်ပါ။

Method Type	Class Attribute	Instance Attribute
Static	Yes (class name)	No
Class	Yes ( <code>cls</code> /class name)	No
Instance	Yes ( <code>self</code> /class name)	Yes ( <code>self</code> )

ຕະຫຼາດ ປ.၁ ພົມວິທີ່ອານາໄສ attribute ອັນໃຈ້ວ່າ ອົງກະຕາວ ເພື່ອປົກລົງ

Method Type	Static Method	Class Method	Instance Method
Static	Yes (class name)	Yes (class name)	No
Class	Yes ( <code>cls</code> /class name)	Yes ( <code>cls</code> /class name)	No
Instance	Yes ( <code>self</code> /class name)	Yes ( <code>self</code> /class name)	Yes ( <code>self</code> )

ତେବେଳ ଉ.ପ୍ ମନ୍ଦିରକୁ ଆଶୀର୍ବାଦ ଦିଲ୍ଲି ମାତ୍ରରେ ଆବଶ୍ୟକ ହେବାକୁ ଅନୁରୋଧ କରିଛି।

```
| # File: accessibility_test.py
```

20

```
# can be used inside the class
top_level_var = 'top level var'

# can be called inside the class
def top_level_fun():
    pass

class AccessibilityTest:

    class_attr = 'class attr'

    def __init__(self):
        self.instance_attr = 'instance attr'
        self.instance_attr2 = 'instance attr2'

    @staticmethod
    def static_method():
        # print(self.instance_attr)  # cannot use
        print(AccessibilityTest.class_attr)
        # can only call static and class method types
        AccessibilityTest.static_method2()
        AccessibilityTest.class_method2()
        # self.instance_method()      # cannot use

    @staticmethod
    def static_method2():
        pass

    @classmethod
    def class_method(cls):
        # print(self.instance_attr)  # cannot use
        print(cls.class_attr)
        print(AccessibilityTest.class_attr)
        # can only call static and class method types
        AccessibilityTest.static_method()
        cls.class_method2()
        AccessibilityTest.class_method2()
        # self.instance_method()      # cannot use

    @classmethod
    def class_method2(cls):
        pass

    def instance_method(self):
```

```

print(self.instance_attr)
print(AccessibilityTest.class_attr)
# can call all type of methods
AccessibilityTest.static_method()
AccessibilityTest.class_method()
self.instance_method2()

def instance_method2(self):
    print(self.instance_attr2)

# just test
avar1 = AccessibilityTest()
AccessibilityTest.static_method()
print()
AccessibilityTest.class_method()
print()
avar1.instance_method()

```

## ၉.၇ Object-Oriented Programming

ကလပ်နဲ့ အော်ဂျက်တွေ အကြောင်း အထိုက်အလျောက် လေ့လာပြီးနောက် ငင်းတိုကို ပရိုကရမ် တည် ဆောက်ရာမှာ အသုံးချတတ်အောင် ဆက်လက် လေ့လာကြရပါမယ်။ လက်တွေရေး လေ့ကျင့်ဖို့ လိုပါ တယ်။ အော်ဂျက်သောတရား အခြေပြု ပရိုကရမ်ရေးသား တည်ဆောက်တာကို *Object-oriented Programming* (OOP) လိုပေါ်ပါတယ်။ ဒီနည်းနဲ့ တည်ဆောက်ထားတဲ့ ပရိုကရမ်တွေကို object-oriented program/software လို့ ဆိုင်ပါတယ်။ စလေ့လာသူတွေ အတာက် အထောက်အကူဖြစ်စေ မဲ့ ရုံးရှင်းတဲ့ နမူနာ object-oriented program လေးတစ်ခုကို လေ့လာကြည့်ပါမယ်။

### A Simple Banking Application

S Bank ဆိုတဲ့ ဘဏ်အတွက် ပရိုကရမ်တစ်ခု ရေးပေးရမယ်လို့ စိတ်ကူးကြည့်ပါ။ ဘဏ်အကောင့် အသစ် ဖွင့်တဲ့အခါ အောက်ပါ အချက်အလက်တွေ လိုအပ်တယ်။ S Bank က လောလောဆယ် ရန်ကုန်၊ မန္တလေး နဲ့ နေပြည်တော် ဘဏ်ခဲ့ သုံးခုပဲ ရှိတယ်။

#### Open Account

Name: Kathy

NRIC: 12/ThaGaKa (N) 283772

Gender (F/M): F

Date of Birth: 12/12/1990

Address Line 1: 45A, Marlar St., 16 Ward

Address Line 2: South Okkalapa

City: Yangon

State/Division: Yanon

Initial deposit: 350000.00

Branch: Yangon

Please confirm (Y/N): Y

Account No: 100000000012

Successfully created

အကောင့်နံပါတ်က ကယ်န်း (၁၂) လုံး ဖြစ်ရမယ်။ 100000000001 က ပထမဆုံးဖွင့်တဲ့ အကောင့်နံပါတ် ဖြစ်ပြီး အကောင့်သစ် တစ်ခုတိုင်းအတွက် နံပါတ်စဉ် တိုး သွားရမှာပါ။

အကောင့်ထဲကို ငွေသွင်းမယ်ဆိုရင် အကောင့်နံပါတ်နဲ့ သွင်းမဲ့ ငွေပမာဏ ထည့်ပေးရပါမယ်။ အကောင့်နံပါတ် ထည့်ပေးတာ မှားခဲ့ရင် အမြားအကောင့်တစ်ခုထဲကို ငွေသွင်းမိနိုင်တယ်။ အကောင့်ပိုင်ရှင် နံမည်ကို ပြပေးပြီး ကွန်ဖန်းလုပ်ရင် အမှားဖြစ်နိုင်ခြား ပိုပြီးနည်းသွားမှာပါ။ ငွေထုတ်ရင်လည်း ဒီလိုပဲ ကွန်ဖန်းလုပ်သင့်တယ်။

### **Deposit**

Account No: 100000000012

Amount: 250000

Please check and confirm the following details:

Holder: Kathy

Account No: 100000000012

Amount to deposit: 250000

Please confirm (Y/N): Y

...

Account transaction successfully completed.

### **Withdrawal**

Account No: 100000000012

Amount: 50000

Please check and confirm the following details:

Holder: Kathy

Account No: 100000000012

Amount to withdraw: 50000

Please confirm (Y/N): Y

...

Account transaction successfully completed.

ငွေလွှဲမယ်ဆိုရင် လွှဲမဲ့အကောင့်၊ လက်ခံမဲ့အကောင့် နှစ်ခုလိုမယ်။ လွှဲမဲ့အကောင့်နဲ့ လက်ခံအကောင့်တစ်ခုတည်း ဖြစ်မနေသင့်ပါဘူး။

### **Transfer**

From Account No: 100000000012

To Account No: 100000000015

Amount: 50000

Please check and confirm the following details:

From

Holder: Kathy

Account No: 100000000012

To

Holder: Sandar

Account No: 100000000015

Amount: 50000

Please confirm (Y/N): Y

...

Account transaction successfully completed.

ဒီပရိုကရမ်လေးက အပြင်မှာ တကယ်သုံးလို့မရပေမဲ့ တတ်နိုင်သမျှ ယူလိုကျအောင်၊ အပြင်နဲ့ နီးစပ်အောင် စဉ်းစားပြီး ရေးသင့်ပါတယ်။ ဥပမာ အကောင်နံပါတ်မှားရင် ဘယ်လိုပြပေးသင့်သလဲ၊ ထုတ်မဲ့ ငင်ပမာဏက လက်ကျန်ငွေထက် ပိုများနေတဲ့အခါ ဘယ်လိုဖြစ်သင့်လဲ စတာတွေကို သေသေချာချာ စဉ်းစားပေးသင့်တယ်။ အခြေခံအဆင့် ဥပမာမို့လို့ အလွန်အကျိုး ရှုပ်ထွေးလို့တော့လည်း မရသေးဘူးပေါ့။ ဘိုင်နာအဆင့်နဲ့ သင့်တော်လောက်မဲ့ ပုံစံမျိုးလေးနဲ့ တစ်ပိုင်းချင်း တည်ဆောက်သွားရအောင်။

ပရိုကရမ်တစ်ခု စတည်ဆောက်ဖို့ ကြီးစားတဲ့အခါ ဘယ်ကစရမယ်မှန်းမသိ အစရှာလို့မရ ဖြစ်နေတတ်ပါတယ်။ ဒီလိုအခြေအနေမျိုးမှာ problem decomposition ကို အသုံးချရမှာပါ။ ဖြေရှင်းမဲ့ ကိစ္စကို အပိုင်းတွေ ခွဲကြည့်ပြီး ဘယ်တစ်ခုကို အရင်ဆုံး တည်ဆောက်ရမလဲ ရွေးချယ်ရပါမယ်။ ဖော်ပြထားချက်တွေအရ ဘက်ပရိုကရမ်က အကောင့်ဖွင့်တာ၊ ငွေသွေး၊ ငွေထုတ်နဲ့ ငွေလွှဲ ကိစ္စတွေ ဆောင်ရွက်ပေးရမှာပါ။ အကောင့်ဖွင့်တဲ့ အပိုင်းကို ပထမဆုံး တည်ဆောက်ပါမယ်။ ဘာလို့ ဒါနဲ့စတင်မယ်လို့ ဆုံးဖြတ်တာလဲ ခိုင်လုတဲ့ အကြောင်းပြချက် သိပ်တော့ကြီးကြီးမားမားရယ် မရှိပါဘူး။ (ငွေသွေး/ထုတ်တွေ လုပ်နိုင်ဖို့က အကောင်းအရင် ဖွင့်ထားမှ လုပ်လို့ရမှာဖြစ်လို့ ဒီကနေ စသင့်တယ်လို့တော့ မြင်မိတယ်)။ ဒါပေမဲ့ ငွေသွေးတဲ့အပိုင်းနဲ့ စမယ်ဆုံးရင်လည်း ရတာပါပဲ။

Object-oriented Programming နည်းနဲ့ ပရိုကရမ်ရေးတဲ့အခါ ဖော်ပြထားချက်ထဲမှာ ပါဝင်တဲ့အကြောင်းအရာတွေကို လေ့လာသုံးသပ်ပြီး ဘယ်အရာတွေကို အော်ဂျက်တွေအနေနဲ့ ဖော်ပြရမလဲ ဆုံးဖြတ်ရတယ်။ အကောင့်၊ အကောင့် ပိုလှုပ်၊ လိပ်စာ၊ ဘက် နဲ့ ဘက်ခွဲ တို့ကို အော်ဂျက်အနေနဲ့ ရှုမြင်နိုင်တယ်။ ဒါကြောင့် Account, Holder, Address, Bank နဲ့ Branch ကလပ်စွဲတွေ သတ်မှတ်ပါမယ်။ Account ကို အရင်ကြည့်ရအောင် ...

## Account Class

Account ကလပ်စွဲက ဥပမာတွေမှာလည်း တွေ့ခဲ့ကြပြီးပါပြီ။ ဘယ်ပရိုကရမ်အတွက် လိုအပ်တာ တရာ့ကို ထပ်ဖြည့်ထားပါတယ်။

```
# File: bank_app.py
class Account:
    _next_acc_no = 100_000_000_001

    def __init__(self, holder, balance, branch):
        self._holder = holder
        self._acc_number = Account._next_acc_no
        Account._next_acc_no += 1
        self._balance = balance
        self._branch = branch

    def deposit(self, amt):
        if amt <= Decimal(0.00):
```

```

        raise ValueError('Invalid amount!')
    self._balance += amt

    def withdraw(self, amt):
        if amt <= Decimal(0.00):
            raise ValueError('Invalid amount!')
        if amt > self._balance:
            raise ValueError('Not enough balance!')
        self._balance -= amt

    def transfer(self, to_acc, amt):
        if self.acc_number == to_acc._acc_number:
            raise ValueError('From-account and to-account should not be '
                            'the same!')
        if amt <= Decimal(0.00):
            raise ValueError('Invalid amount!')
        if amt > self._balance:
            raise ValueError('Not enough balance!')
        self._balance -= amt
        to_acc._balance += amt

    @property
    def holder(self):
        return self._holder

    @property
    def acc_number(self):
        return self._acc_number

    @property
    def balance(self):
        return self._balance

    @property
    def branch(self):
        return self._branch

```

ရှေ့မှောက်ပြခဲ့တဲ့ လိုအပ်ချက်အရ အကောင့်နံပါတ်မှာ ကဏ္ဍး (၁၂) လုံး ရှိရမယ်။ ပထမဆုံး ဖွင့်တဲ့အကောင့် နံပါတ်က 1000000000001 ဖြစ်ရမယ်။ နောက်ထပ် အကောင့်သစ်တစ်ခု ဖွင့်တိုင်း နံပါတ်စဉ် တစ်ခုချင်း တိုးသွားရမှာပါ။ ဒီလိုအပ်ချက်အတွက် class attribute ကို အသုံးပြုနိုင်ပါတယ်

`_next_acc_no = 100_000_000_001`

Initializer ထဲမှာ ဒီ attribute ကို ဘယ်လို သုံးထားလဲ လေ့လာကြည့်ပါ။ လက်ရှိအော်ဘူက်ရဲ့ အကောင့်နံပါတ်အဖြစ် သုံးပြီးတာနဲ့ နောက်ဖန်တီးမဲ့ အော်ဘူက်အတွက် အသင့်ဖြစ်အောင် တစ်တိုးပေး ထားတာ တွေ့ရမှာပါ။ မက်သဒ်တွေနဲ့ instance attribute တွေကတော့ ရှေ့က ဥပမာတွေမှာ တွေ့ပြီး

ဖြစ်တဲ့အတွက် သိပ်ထွေထွေထူးထူး ရှင်းပြန့် မလိုလောက်တော့ဘူး။ transfer မက်သဒ်ကတော့ နည်းနည်းထူးခြားတော့ပါနေတယ်

```
def transfer(self, to_acc, amt):
    # ... (တရာ့လိုင်းတွေ မပြထား)
    self._balance -= amt
    to_acc._balance += amt # မသုံးစေချင်တာကို ယူသုံးထားတာ ဖြစ်မနေဘူးလား
```

self.\_balance က ပြဿနာမရှိပေမဲ့ to\_acc ရဲ့ \_balance ကို တိုက်ရှိက် ယူသုံးထားတာ access control စည်းကမ်းဖောက်သလို ဖြစ်မနေဘူးလား မေးစရာရှိပါတယ်။ အခု ဒီလိုမျိုး သုံးထားတာက Account ကလပ်စံထဲမှာပါ။ to\_acc ကလည်း အခြား Account အော့ဘ်ဂျက် တစ်ခုဖြစ်တယ်။ လက်ရှိကလပ်စံသတ်မှတ်သူဟာ အဲဒီကလပ်စံအမျိုးအစား အော့ဘ်ဂျက်တွေရဲ့ attribute တွေကို တိုက်ရှိက်သုံးတာက ပြဿနာမရှိဘူးလို့ ယူဆပါမယ်။ အကြောင်းအရင်းက ဒီလိုပါ။ ပိုင်ရှင်ဟာ သူကလပ်စံ attribute (ပုံတိကျအောင် ပြောရင် သူကလပ်စံကနေ ဖန်တီးယူလိုက်တဲ့ အော့ဘ်ဂျက်ရဲ့ attribute) တွေကို အခြားသူတွေကိုပဲ တိုက်ရှိက် အသုံးမပြုစေချင်တာ။ သူကိုယ်တိုင်အနေနဲ့တော့ သုံးလို့ရှုသင့်ပါတယ်။ ဘာဖြစ်လိုလဲဆိုတော့ သူကိုယ်တိုင်က ပိုင်ရှင် ဖြစ်တဲ့အတွက် ကလပ်စံကို ပြင်ဆင်ပြောင်းလဲတဲ့အခါ ဒီလိုတိုက်ရှိက်ယူသုံးထားတဲ့အပေါ် ပြဿနာ ရှိ/မရှိ သူအနေနဲ့ သုံးသပ်နိုင်ပါတယ် (သူရေးထားတဲ့ ကလပ်စံပဲလေ၊ ဘာလုပ်ရင် ဘာဖြစ်မယ် သူသိမှာပေါ့)။ ဒီအတွက် တိုက်ရှိက်ယူသုံးထားတဲ့ နေရာတွေကိုလည်းလိုအပ်သလို အလိုက်သင့် ပြင်ဆင်ပြောင်းလဲ ပေးနိုင်ပါတယ်။ ဒီလောက်ဆုံးရှင် ကလပ်စံပိုင်ရှင် အနေနဲ့ အဲဒီကလပ်စံ အော့ဘ်ဂျက်တွေရဲ့ attribute တွေကို တိုက်ရှိက် အသုံးပြုလို့ ရာသင့်တယ်ဆိုတဲ့ အကြောင်းပြုချက်ဟာ ကျိုးကြောင်းဆီလျှော့ရှိတယ်လို့ စာဖတ်သူ လက်ခံနိုင်မယ် မျှော်လင့်ပါတယ်။

ကလပ်စံတစ်ခုမှာ ပါဝင်သင့်တဲ့ မက်သဒ်နဲ့ attribute တွေကို တစ်ခါတည်းနဲ့ အားလုံးပြီးပြည့်စုံအောင် ပြိုတင်စဉ်းစားထားဖို့ မလိုအပ်ပါဘူး။ စဉ်းစားလို့ရှုပြာတွေနဲ့ပဲ စသတ်မှတ်ထားနိုင်တယ်။ ပရိုဂရမ်တည်ဆောက်ရင်းနဲ့ လိုအပ်တာတွေ ထပ်ပြီး တွေ့လာ၊ မြင်လာတဲ့ အခါကျမှ ပြင်ဆင်ဖြည့်စွက်လို့ရှုပါတယ်။ ကလပ်စံတစ်ခုမှာပဲ လည်နေပြီး ရှုံးမဆက်နိုင်တာဟာ စလေ့လာသူ ဖြစ်လေ့ရှိတဲ့ အမှားတစ်ခုပါ။

## Address and Holder Classes

Address နဲ့ Holder ကို အခုလို သတ်မှတ်ထားပါတယ်။ Account နဲ့ နှီးယူဥ်ရင် အတော်လေး ရှိုးရှင်းတာ တွေ့ရှုမှာပါ။

```
class Address:
    def __init__(self, line_1, line_2, city, division):
        self._address_line_1 = line_1
        self._address_line_2 = line_2
        self._city = city
        self._division = division

    @property
    def address_line_1(self):
        return self._address_line_1

    @property
    def address_line_2(self):
        return self._address_line_2
```

```

@property
def city(self):
    return self._city

@property
def division(self):
    return self._division

class Holder:
    def __init__(self, name, dob, nric, gender, address):
        if name.strip() == '':
            raise ValueError('Name cannot be empty!')
        if age_today(dob) < 18:
            raise ValueError('Too young to open a bank account!')
        if nric.strip() == '':
            raise ValueError('Name cannot be empty!')
        if not (gender.upper() == 'F'
                or gender.upper() == 'M'):
            raise ValueError('Unknown gender!')
        self._name = name
        self._dob = dob
        self._nric = nric
        self._gender = gender.upper()
        self._address = address

    @property
    def age(self):
        return age_today(self._dob)

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        if name.strip() == '':
            raise ValueError('Name cannot be empty!')
        self._name = name

    @property
    def dob(self):
        return self._dob

    @dob.setter
    def dob(self, value):

```

```

    self._dob = value

@property
def gender(self):
    return 'FEMALE' if self._gender == 'F' else 'MALE'

```

Holder ရဲ age property မှာ age\_today ဖန်ရှင် သုံးထားတယ်။ Insurance premium ပရီ ကရမ်ဗုံးလည်း ဒီဖန်ရှင်ကို သုံးထားတယ်။ ပရီကရမ်တွေမှာ ဒီလိုမျိုး မကြာခက် ပြန်အသုံးပြုရတဲ့ ဖန်ရှင် တွေကို မော်ဒူး (သို့) လိုက်ဘရိတစ်ခုအနေနဲ့ ထုတ်ထားလို့ရတယ်။ လိုတဲ့အခါ import လုပ်ပြီး သုံးရုပံ့။ ပြန်ရေးဖို့ မလိုတော့ဘူး။ မော်ဒူးအကြောင်းကို နောက်အခန်းတွေမှာ ဆက်လက်လေ့လာကြမှုပါ။ လောလောဆယ်တော့ ဘဏ်ပရီကရမ်ဖိုင် (bank\_app.py) ရဲ top level မှာ ပြန်ရေးပါမယ်။ (Copy and paste လုပ်လည်းရတယ်။)

```

# File: bank_app.py
# top level ဖန်ရှင်ပါ ကုန်ဖိုင်ရဲ အောက်မှာ တိုက်ရိုက်ရှိပါတယ် ကလပ်စံထဲမှာမဟုတ်
def age_today(dob):
    today = date.today()
    this_bd = dob.replace(year=today.year)
    if today - dob >= this_bd - dob:
        return today.year - dob.year
    else:
        return today.year - dob.year - 1

```

## Branch and Bank Classes

Branch ကလပ်ကတော့ ရှင်းပါတယ်။ နံမည်နဲ့ လိပ်စာပဲ ပါတယ်။ ပရီကရမ် လိုအပ်ချက်အရ နောက်ပိုင်းထပ်ဖြည့်ရတာတွေတော့ ရှိလာနိုင်ပါတယ်။ ဥပမာ ဘဏ်ခဲ့ စဖိုင်တဲ့ ရက်နိုင်တဲ့ ဝန်ဆောင်မှုတွေ စသည်ဖြင့်။ အခုတော့ ဒါတွေ မလိုသေးပါဘူး။ ဖော်ပြထားချက်ကို ပြန်ကြည့်ရင် ဘဏ်ခဲ့လိပ်စာကိုလည်း သုံးဖို့ မလိုသေးတာ တွေ့ရမှုပါ။ အခုပုရီကရမ် လိုအပ်ချက်အရဆုံးရင် ဘဏ်ခဲ့ကို သီးခြား အော့သုတေသနနဲ့ မယူဆလည်း ရတယ်။ ဘဏ်ခဲ့နံမည်ကို စာသားအနေနဲ့ သိမ်းထားရင် ရပါ။ ဒါပေမဲ့ ဘဏ်ခဲ့ဟာ အပြင်မှာတကယ်ရှိတဲ့ အရာဖြစ်တယ်။ နံမည်အပြင် ဘဏ်ခဲ့နဲ့ပါတယ်ကတဲ့ အခြား အချက်အလက်တွေကိုလည်း နောက်ပိုင်းမှာ လိုအပ်လာနိုင်တယ်လို့ မျှော်မှုန်းရပါတယ်။ ဒါကြောင့် အော့သုတေသနဖြစ်သင့်တယ်လို့ ယူဆပါမယ်။

```

class Branch:
    def __init__(self, name, address):
        self._name = name
        self._address = address

@property
def name(self):
    return self._name

```

မှတ်ချက်။ ။ ဒီနေရာမှာ ဘဏ်ခဲ့သည့် အော့သုတေသန ဖြစ်ကိုဖြစ်ရမယ့်လို့ တစ်ထံချမယူဆဘဲ လိုအပ်ချက်ပေါ်မှုတည်ပြီး ဆုံးဖြတ်နိုင်တယ် ဆိုတာကို အဓိကပြောချင်တယ်ပါ။ မလိုအပ်ဘူး ယူဆရင် ဘဏ်ခဲ့နံမည်ကို စာသားအနေနဲ့ ထားပြီး နောင်လိုအပ်လာတော့မှ ကလပ်စံ သတ်မှတ်လည်း ရတယ်။

Bank ကလပ်စိတ်ရအောင်။ ဘဏ်မှာ အကောင့်တွေ ရှိပါမယ်။ အကောင့် ဖွင့်ထားတဲ့ ကာစ တပ်မာတွေ ရှိပါမယ်။ ဘဏ်ခဲ့သုံးခုရှိမယ်။ ဆက်စပ်မှုက has-a relationship ပါ။

```
class Bank:
    def __init__(self, name):
        self._name = name
        self._accounts = []
        self._acc_holders = []
        self._branches = [Branch('Yangon', Address('50A, Theinphyu St.',
                                                    'Botataung Township',
                                                    'Yangon', 'Yangon')),
                          Branch('Mandalay', Address('65B, 63 St.',
                                                    'Chanayetharsi',
                                                    'Mandalay',
                                                    'Mandalay')),
                          Branch('Naypyidaw', Address('332C, Main Rd.',
                                                    'Ottarathiri Township',
                                                    'Naypyidaw',
                                                    'Naypyidaw Union'))]

    def open_acc(self, holder, init_bal, branch_name):
        branch = self.retrieve_branch(branch_name)
        account = Account(holder, init_bal, branch)
        self._accounts.append(account)
        self._acc_holders.append(holder)
        return account

    def withdraw(self, acc_no, amt):
        acc = self.retrieve_account(acc_no)
        acc.withdraw(amt)

    def deposit(self, acc_no, amt):
        acc = self.retrieve_account(acc_no)
        acc.deposit(amt)

    def transfer(self, from_acc_no, to_acc_no, amt):
        from_acc = self.retrieve_account(from_acc_no)
        to_acc = self.retrieve_account(to_acc_no)
        from_acc.transfer(to_acc, amt)

    def retrieve_account(self, acc_no):
        for acc in self._accounts:
            if acc.acc_number == acc_no:
                return acc
        raise ValueError(f'Account with {acc_no} not found!')
```

```
def retrieve_branch(self, name):
    for branch in self._branches:
        if branch.name == name:
            return branch
    raise ValueError(f'{name} branch not found!')
```

ဘဏ်ခွဲသုံးခုကို ပြိုပြီး ထည့်ထားတယ်။ လက်ရှိမှာ သုံးခဲ့ရှိပေမဲ့ ဘဏ်ခွဲ အသစ်တွေ ထပ်ဖွင့်နှင့်တယ်။ ပရိုဂရမ် နောက်ပိုင်း ဗားရှင်းတွေမှာ ဘဏ်ခွဲအသစ် ထပ်ထည့်နိုင်ဖို့ လိုအပ်ချက်ရှိလာမယ်လို့ မျှော်မှန်း ရပါတယ်။ အားလုံး သိပြီးဖြစ်တဲ့အတိုင်း ပရိုဂရမ်တွေက သူတိုကို အသုံးပြုနေတဲ့ ကာလတစ်လျှောက် အမြဲပိုင်ဆင်ဖြည့်စွက်တာ၊ ပြုပိုင်ထိန်းသိမ်းတာတွေ လုပ်နေရမှာပါ။ ဗားရှင်းအသစ်တွေ မကြာခဏ ဖူ-  
date လုပ်ကြရပါတယ်။ တကယ့် အသုံးချ ဆော်ဖွဲ့တွေ တည်ဆောက်တဲ့အခါ ပြုပိုင်ထိန်းသိမ်းရ လွယ်ကူ အောင်၊ ဒီချေအသစ်တွေ ထပ်ထည့်ဖို့ မခက်ခဲအောင် ဒီဇိုင်းပြုလုပ်ထားရပါတယ်။ အခြေခံ ပရိုဂရမ်းမင်း တတ်ကျမ်းပြီးတဲ့နောက်မှာ ဒီလို ဒီဇိုင်းပိုင်းဆိုင်ရာတွေ နားလည်အသုံးချတတ်လာအောင် ဆက်လက် လေ့ လာဖို့ လိုအပ်ပါလိမ့်မယ်။

Account မှာ ပါတဲ့ မက်သဒ်တွေကို Bank မှာလည်း ပြန်တွေ့ရပါတယ်။ transfer ကို ကြည့်ရင် အခုလိုရေးထားတာ တွေ့ရတယ်

```
def transfer(self, from_acc_no, to_acc_no, amt):
    from_acc = self.retrieve_account(from_acc_no)
    to_acc = self.retrieve_account(to_acc_no)
    from_acc.transfer(to_acc, amt)
```

အခု Bank ကလပ်စ် transfer က အကောင့် နံပါတ်နှစ်ခုနဲ့ ငွေပမာဏကို လက်ခံထားတယ်။ အဲဒီနံပါတ် နဲ့ အကောင့်နှစ်ခုကို ဘဏ်ခွဲ အကောင့်တွေအားလုံးထဲကနေ ဆွဲထုတ်တယ် (retrieve\_account မက် သဒ်ကို ကြည့်ပါ။)။ ရလာတဲ့ အော့ဘ်ဂျက်ရဲ့ transfer ကို တစ်ဆင့်ပြန်ခေါ်ထားတယ်။ ဘာလို ဒီလို အဆင့်ဆင့် လုပ်နေရလဲ မေးစရာရှိလာပါတယ်။ တစ်ဆင့်တည်းနဲ့ရော ရအောင်လုပ်လို့ မရရှိနိုင်ဘူးလား။ ဘာလို အခုလို နှစ်ဆင့်လုပ်သင့်လဲ design principle တာ၌နဲ့ ရှင်းပြုပို့ ရနိုင်ပေမဲ့ အခြေခံအဆင့် အတွက် အချိန်မကျသေးဘူးလို့ ယူဆရပါတယ်။ လက်ရှိအဆင့်မှာ အခုပြထားတဲ့ နည်းလမ်းအပြင် မိမိ စဉ်းစားလိုရတဲ့ အခြားနည်းလမ်းနဲ့ ပြောင်းလဲ စမ်းသပ်ရေးကြည့်ပါ။ ဘယ်နည်းလမ်းက ရေးရတာ (သို့) နားလည်းရတာ ပိုလွယ်လဲ ပိုရှိရှင်းလဲ စဉ်းစားချင့်ချိန်ကြည့်ခြင်းအားဖြင့် အကျိုးရှိနိုင်တယ်။ ပိုပြီးတော့ ထိုးထွင်းသိမြင်လာဖို့လည်း အထောက်အကူး ဖြစ်နိုင်ပါတယ်။

မှတ်ချက်။ ။ Bank ဟာ ငါးရဲ့ အကောင့် အော့ဘ်ဂျက်တွေ အားလုံးကို စုစုည်းသိမ်းထားတယ်။ ဒါကြောင့် အကောင့်နံပါတ်နဲ့ အကောင့်တွေကို ရှာတဲ့အလုပ်ကို Bank ကပဲ လုပ်ဆောင်ပေးတာ သာဘ ဝကျတယ်။ ငွေလွှဲတဲ့အခါ အကောင့်နှစ်ခုရဲ့ လက်ကျွန်းငွေကို ပြောင်းလဲစေမှာပါ။ လက်ကျွန်းငွေ balance က Account နဲ့ဆိုင်တဲ့ အချက်အလက် ဖြစ်တာကြောင့် transfer ဟာ Account မှာပဲရှိသင့်တယ်လို့ ယူဆနိုင်တယ် (တစ်နည်းအားဖြင့် အောက်ပါတော်းတွေကို ပိုင်ဆိုင်ထားတဲ့ ကလပ်စ်နဲ့ ငါးစေတာအပေါ် မြှို့လုပ်ဆောင် ရမဲ့ မက်သဒ်ကို တပေါ်စ်းတစည်းတည်း ရှိစေချင်တာ)။ ငွေလွှဲကိစ္စဟာ ဘဏ်နဲ့ ငါးပိုင်ဆိုင်ထားတဲ့ အကောင့်နှစ်ခု ပူးပေါင်းလုပ်ဆောင်ရတဲ့ အလုပ်လို့ ရှိမြဲ့ကြည့်နိုင်ပါတယ်။

## BankApp Class

Input/Output နဲ့ ဆိုင်တာတွေကို လုပ်ငန်းပိုင်းဆိုင်ရာတွေနဲ့ ခွဲဥြားရေးသင့်တယ်လို့ အတွေ့အကြိုး ပရိုဂရမ်တည်ဆောက်သူတွေ အားလုံးက လက်ခံထားကြပါတယ်။ လုပ်ငန်းပိုင်းဆိုင်ရာကို domain လို့ ခေါ်ပါတယ်။ ဘဏ်လုပ်ငန်းမှာဆိုရင် အကောင့်ဖွင့်ခြင်း၊ ငွေ သွင်း/လွှဲ/ထုတ် လုပ်ခြင်း၊ အတိုးပေးခြင်း

ချေးငွေထုတ်ပေးခြင်း စတာတွေဟာ banking domain နဲ့ သက်ဆိုင်တဲ့ အရာတွေပါ။ ရှေ့မှာဖော်ပြခဲ့တဲ့ ကလပ်စံတွေဟာ banking domain နဲ့ သက်ဆိုင်ပါတယ်။

Input/Output ဆိုတာ ပရီဂရမ်နဲ့ ပြင်ပလောက ဆက်သွယ်လုပ်ဆောင်တဲ့ ကိစ္စတွေလို့ အကြမ်း ဖျဉ်း ပြောလို့ရပါတယ်။ ကီးဘုဒ်ကနေ ရိုက်ထည့်လိုက်တာကို ပရီဂရမ်က ဖတ်တာဟာ input ဖြစ်တယ်။ မော်နှစ်တာမှာ စာသား/ရုပ်ပုံ ပြေားတာဟာ output ဥပမာ တစ်ခုဖြစ်တယ်။ Input/output အပိုင်းမှာ ပါဝင်တဲ့ အခြား ဥပမာတွေလည်း အများကြီးရှုပါတယ်။

BankApp က input/output ကို အဓိက တာဝန်ယူလုပ်ဆောင်ပေးတဲ့ ကလပ်စံပါ။ ဒီကလပ်စံက နေ အော်ဂျက်တွေ ဖန်တီးယူဖို့ မလိုတဲ့ အတွက် initializer နဲ့ instance attribute/method တွေ မပါတော့ဘူး။ ကလပ်စံအနေနဲ့ ထားပြီး ဘာလို့ အော်ဂျက်ဖန်တီးဖို့ မလိုအပ်တာလဲ။ BankApp ဟာ ဘဏ်ပရီဂရမ်တစ်ခုလုံးကို ကိုယ်စားပြုတယ်။ ပရီဂရမ်ကတစ်ခုပဲ ရှိမှာပါ။ ဒီတော့ အော်ဂျက်တစ်ခုချင်း စီ ပိုင်ဆိုင်ထားရမဲ့ attribute တွေလည်း သူမှာ မရှိဘူး။ ဒါကြောင့် ကလပ်စံအနေနဲ့ ထားပြီး အသုံးပြုလိုရတယ်။

Class attribute နှစ်ခု တွေရတယ်။ ‘S Bank’ အတွက် Bank အော်ဂျက်တစ်ခုနဲ့ ဘဏ်ခဲ့နံမည်တွေ ပါတဲ့ list တစ်ခုတို့ ဖြစ်တယ်။ ဘဏ်ခဲ့နံမည် list ကို read\_branch မှာ သုံးထားတယ်။ ဒီ မက်သင်က ဘဏ်ခဲ့နံမည် မှားနေရင် ပြန်ထည့်ခိုင်းမှာပါ။

```
class BankApp:

    _bank = Bank('S Bank')
    _branches = ['Yangon', 'Mandalay', 'Naypyidaw']

    @classmethod
    def open_account(cls):
        name = read_nonempty_text('Name: ')
        nric = read_nonempty_text('NRIC: ')
        gender = read_gender('Gender (F/M): ')
        dob = read_date('Date of Birth: ')

        address_line1 = read_nonempty_text('Address Line 1: ')
        address_line2 = read_nonempty_text('Address Line 2: ')
        city = read_nonempty_text('City: ')
        division = read_nonempty_text('State/Division: ')

        init_deposit = read_decimal('Initial deposit: ')

        address = Address(address_line1, address_line2, city, division)
        holder = Holder(name, dob, nric, gender, address)
        branch = cls.read_branch()
        has_confirmed = read_confirmation('Please confirm (Y/N): ')
        if has_confirmed:
            try:
                acc = cls._bank.open_acc(holder, init_deposit, branch)
                print(f'A new account with account number {acc.acc_number} '
                      f'successfully created')
            except Exception as e:
                print(f'An error occurred: {e}')
        else:
            print('Operation canceled by user')

    @classmethod
    def read_branch(cls):
        return cls._branches[cls._branch_index]

    @classmethod
    def _set_branch_index(cls, index):
        cls._branch_index = index
```

```

        except ValueError as err:
            print(err)
        except RuntimeError as err:
            print(err)

@classmethod
def read_branch(cls):
    while True:
        branch_name = read_nonempty_text('Branch: ')
        if branch_name in cls._branches:
            return branch_name
        print(f'{branch_name} not found!')

```

open\_account သည် class method ပါ။ read\_nonempty\_text က စာသား input ဖတ်ပေးတဲ့ ဖန်ရှင်ဖြစ်တယ်။ bank\_app.py ဖို့ top level မှ သတ်မှတ်ထားတာပါ။ အခြား read\_decimal, read\_gender စတဲ့ input ဖတ်ပေးတဲ့ ဖန်ရှင်တွေကိုလည်း top level မှပဲ သတ်မှတ်ထားတယ်။ ဘဏ်ခွဲ နံမည်ဖတ်တဲ့ read\_branch ကိုတော့ BankApp class method အနေနဲ့ သတ်မှတ်တယ်။ ဘဏ်ခွဲ နံမည်များရင် ပြန်ထည့်ပေးခိုင်း ရပါမယ်။ ဘဏ်ခွဲနံမည်ဖတ်တာက ဘဏ်ပရီဂရမ်နဲ့ပဲ ဆက်စပ် အသုံးပြုရမှာပါ။ ဒုံးအပြင် ဘဏ်ခွဲနံမည်တွေကိုလည်း BankApp class attribute အနေနဲ့ ထားသင့်တယ်။ BankApp ကလပ်စံဟာ ဆက်စပ်မှရှိတဲ့ ဖန်ရှင်တွေနဲ့ ဒေတာတွေကို စုစည်းပေးထားတာပါ။ ဒီက လပ်စံထဲမှာ ပါရှိတာတွေကို top level ကို ထုတ်ရင်လည်း ရပေးမဲ့ အခုံလိုဖွံ့စည်းထားခြင်းအားဖြင့် ဘဏ်ပရီဂရမ်နဲ့ပဲ ဆက်စပ်အသုံးပြုရတဲ့ အရာတွေကို ပိုပြီး နီးနီးကပ်ကပ် တပေါင်းတစည်းတည်း ရှိစေတယ်။ Top level ဖန်ရှင်တွေကတော့ အခြား ပရီဂရမ်တွေမှာလည်း အသုံးတည့်မယ်။ နောက်ပိုင်းမှာ မော်ဒါး တစ်ခုအနေနဲ့ ထုတ်ထားရင် ပို့ကောင်းတယ်။

### **deposit, withdraw, transfer**

Input/output အတွက် deposit, withdraw နဲ့ transfer တို့ဟာလည်း class method တွေ ပါ။ open\_account နားလည်ရင် ဒီမက်သံတွေကိုလည်း နားလည်မှာပါ။ သဘောပေါက်အောင် လေ့လာ ကြည့်ဖို့ တိုက်တွန်းပါတယ်။

```

@classmethod
def withdraw(cls):
    acc_no = read_nonempty_text('Account no: ')
    try:
        acc = cls._bank.retrieve_account(int(acc_no))
    except ValueError as err:
        print(err)
        return
    amt = read_decimal('Amount: ')
    print('Please check and confirm the following details:')
    print(f'Holder: {acc.holder.name}')
    print(f'Account no: {acc.acc_number}')
    print(f'Amount to withdraw: {amt}')
    has_confirmed = read_confirmation('Please confirm (Y/N): ')
    if has_confirmed:

```

```

    try:
        cls._bank.withdraw(int(acc_no), amt)
        print('Operation successfully completed.')
    except ValueError as err:
        print(err)

@classmethod
def deposit(cls):
    acc_no = read_nonempty_text('Account no: ')
    try:
        acc = cls._bank.retrieve_account(int(acc_no))
    except ValueError as err:
        print(err)
    return
    amt = read_decimal('Amount: ')
    print('Please check and confirm the following details:')
    print(f'Holder: {acc.holder.name}')
    print(f'Account no: {acc.acc_number}')
    print(f'Amount to deposit: {amt}')
    has_confirmed = read_confirmation('Please confirm (Y/N): ')
    if has_confirmed:
        try:
            cls._bank.deposit(int(acc_no), amt)
            print('Operation successfully completed.')
        except ValueError as err:
            print(err)

@classmethod
def transfer(cls):
    from_acc_no = read_nonempty_text('From account number: ')
    to_acc_no = read_nonempty_text('To account number: ')
    try:
        from_acc = cls._bank.retrieve_account(int(from_acc_no))
        to_acc = cls._bank.retrieve_account(int(to_acc_no))
    except ValueError as err:
        print(err)
    return
    amt = read_decimal('Amount: ')
    print('Please check and confirm the following details:')
    print('From:')
    print(f'    Holder: {from_acc.holder.name}')
    print(f'    Account no: {from_acc.acc_number}')
    print('To:')
    print(f'    Holder: {to_acc.holder.name}')
    print(f'    Account no: {to_acc.acc_number}')
    print(f'Amount to deposit: {amt}')
    has_confirmed = read_confirmation('Please confirm (Y/N): ')

```

```

if has_confirmed:
    try:
        cls._bank.transfer(int(from_acc.acc_number),
                           int(to_acc.acc_number),
                           amt)
        print('Operation successfully completed.')
    except ValueError as err:
        print(err)

```

## Application Loop

အခုနောက်ဆုံးမှာ အကောင့်ဖွင့်၊ ငွေသွင်း ငွေထုတ်၊ ငွေလွှဲ ဘယ် transaction လုပ်မလဲ ရွေးလိုက်အောင် application loop ကို ရေးရပါမယ်။

```

@classmethod
def app_loop(cls):
    print('O (to open a new account)')
    print('D (to deposit)')
    print('W (to withdraw)')
    print('T (to transfer)')
    while True:
        menu_choice = input('Please enter O/D/W/T to start '
                            'a transaction: ')
        if menu_choice == 'O':
            cls.open_account()
        elif menu_choice == 'D':
            cls.deposit()
        elif menu_choice == 'W':
            cls.withdraw()
        elif menu_choice == 'T':
            cls.transfer()
        else:
            print(f'{menu_choice} is not valid!')

```

ပရိုဂရမ်က ဒီ loop ကနေ စအလုပ်လုပ်ရမှာဆိုတော့ အခုလို ခေါ်ပေးရုပ်ပဲ

```
| BankApp.app_loop()
```

ဒီလိုပြပေးပါလိမ့်မယ်။ O/D/W/T ထဲက တစ်ခုရွေးရင် လုပ်ဆောင်မဲ့ ကိစ္စအလိုက် လိုအပ်တဲ့ input တွေ ဆက်တောင်းမှပါ။ နောက်ဆုံးမှာ Y/N ကွန်ဖန်းလုပ်ရပါမယ်။

O (to open a new account)
D (to open a new account)
W (to open a new account)
T (to open a new account)

Please enter O/D/W/T to start a transaction:

## Top Level Common Functions

Top level ဖန်ရှင်တွေကို လေ့လာလိုအောင် ဖော်ပြပေးထားပါတယ်။ အလားတူ ဖန်ရှင်တွေကို အခန်း(၈) မှာ အသေးစိတ် ရှင်းပြထားတယ်။

```
FEMALE = 'F'
MALE = 'M'

MONTHS = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4,
          'May': 5, 'Jun': 6, 'Jul': 7, 'Aug': 8,
          'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}

def age_today(dob):
    today = date.today()
    this_bd = dob.replace(year=today.year)
    if today - dob >= this_bd - dob:
        return today.year - dob.year
    else:
        return today.year - dob.year - 1

def parse_date_str(dt_str):
    parts = dt_str.replace('/', '-').split('-')
    if parts[0].isdigit():
        return date(int(parts[2]), int(parts[1]), int(parts[0]))
    else:
        return date(int(parts[2]), MONTHS[int(parts[0])], int(parts[1]))

def read_date(prompt):
    while True:
        try:
            return parse_date_str(input(prompt))
        except Exception as err:
            print("Incorrect date format. Please enter the date again.")
            print('Error is probably: ' + str(err))

def read_gender(prompt):
    while True:
        gender = input(prompt)
        if gender in ['F', 'M']:
            return gender
        else:
            print('Invalid gender! Accept only F or M!')
```

```

def read_confirmation(prompt):
    while True:
        choice = input(prompt)
        if choice == 'Y':
            return True
        elif choice == 'N':
            return False
        else:
            print('Invalid gender! Accept only Y or N!')
    
```

```

def read_decimal(prompt):
    while True:
        value = input(prompt)
        try:
            return Decimal(value)
        except InvalidOperation:
            print('Invalid number!')
    
```

```

def read_nonempty_text(prompt):
    while True:
        txt = input(prompt)
        if txt.strip() != '':
            return txt
    print(f'{prompt} cannot be empty!')
    
```

### ၃၃။

အခြေဖြစ်တဲ့ ပရိုကရမ်ကို လေ့လာခြင်းအားဖြင့် အော့သံကျက်နဲ့ ကလပ်စွေ့တွေကို ပရိုကရမ်တစ်ခု တည်ဆောက်ရမှာ ဘယ်လိုအသုံးချဖို့ရလဲ သဘောပေါက် နားလည်မယ်လို့ မျှော်လင့်ပါတယ်။ ဒီပရိုကရမ်မှာ အခြားဖို့ချေတွေ ထပ်ဖြည့်ကြတ်ပါ။ ဥပမာ လက်ကျန်ငွေ စစ်လို့ရအောင်၊ ဘဏ်ခွဲ အသစ်ထည်လို့ရအောင် ချုတ်ကြည်ရင် ပိုပြီး နားလည်လာမှာပါ။

# အခန်း ၁၀

## Inheritance and Polymorphism

*Inheritance* ဟာ ရှိထားပြီး ကလပ်စာစုံတဲ့ attribute နဲ့ မက်သဒ္ဓတွေကို အဗြားကလပ်စာနဲ့ ဆက်ခံယူဖို့ဖြစ်နိုင်တယ်။ ကလပ်စာစုံတဲ့ ချို့ယူရမဲ့ ကုဒ်တွေကို ထပ်ခါထပ်ခါ ရေးစရာ မလိုဘဲ ပြန်လည် အသုံးပြုလိုရစေတဲ့ နည်းလိုလည်း ယူဆနိုင်တယ်။ ဆက်ခံရရှိတဲ့ attribute နဲ့ မက်သဒ္ဓတွေအပြင် အသစ် ထပ်ထည့်တာ၊ နဂါးရှိရင်းကို ပြင်ဆင်တာလည်း ကလပ်စုံအသစ်က လုပ်ဆောင်လိုရမှာ ဖြစ်တယ်။ Object-oriented programming မှာ inheritance ဟာ အရေးအပါဆုံး သဘောတရားတစ်ခု ဆိုရင်လည်း မမှုံးဘူး။ ဂိမ်းရေးတဲ့ လိုက်ဘရိတွေ၊ Graphical User Interface (GUI) အတွက် လိုက်ဘရိတွေ အသုံးပြုမယ်ဆိုရင် inheritance သဘောတရားက မသိမဖြစ်ပါဘူး။ ဒီအခန်းအတွက် inheritance အသုံးချု ဥပမာအနေနဲ့ ‘Breakout’ လိုခေါ်တဲ့ ဂိမ်းလေးတစ်ခုကို Arcade လိုက်ဘရိနဲ့ ရေးထားတာကို နောက်ပိုင်းမှာ လေ့လာကြရမှာပါ။

ရှုံးအခန်းမှာ ဖော်ပြုခဲ့တဲ့ has-a အပြင် object-oriented programming မှာ အရေးပါတဲ့ နောက် relationship တစ်မျိုးက *is-a* ဖြစ်ပါတယ်။ Is-a relationship ရှိတဲ့ အရာတွေကို inheritance နဲ့ ထင်ဟပ်ဖော်ပြုင်တယ်။ ဘတ်စုံကားသည် ကားဖြစ်သည်။ ကားသည် ယာဉ်ဖြစ်သည်။ ယာဉ်ကားနဲ့ ဘတ်စုံကားတို့အကြား ဆက်စပ်မှုဟာ *is-a* relationship ဖြစ်တယ်။ Vehicle, Car နဲ့ Bus ကလပ်တွေဟာ အစဉ်အလိုက် inherit လုပ်ယူနိုင်တာကို တွေ့ရမှာပါ။

### ၁၀.၁ Inheritance ဥပမာ ‘Account Class Hierarchy’<sup>9</sup>

ဘဏ်အကောင် အမျိုးအစား အမျိုးမျိုး ရှိပါတယ်။ အတိုးအနှစ်း၊ တစ်နေ့တာ လုပ်ဆောင်နိုင်တဲ့ transaction အကြိမ်အရေအတွက်၊ အနည်းဆုံး ထားရှုံးမဲ့ ငွေပေးကြား၊ လက်ရှိ လက်ကျန်ငွေထက် ပိုထုတ်လို ရ/မရ (overdraft) စတဲ့အချက်တွေ အကောင့်တစ်မျိုးနဲ့တ်မျိုး မတူကြပါဘူး။

ဥပမာ savings account (ငွေစွာအကောင်) ဆိုရင် ဘဏ်တိုးရမယ်။ တစ်နေ့တာ transaction အကြိမ်အရေအတွက်ရော ထုတ်ယူနိုင်တဲ့ ငွေပေးကြား အကောင့်အသတ်ရှိတယ်။ ဒါကြောင့် ဒီအကောင် အမျိုးအစားက လုပ်နေးသုံးအတွက် အဆင်မပြောဘူး။ Current account ကတော့ စီးပွားရေး လုပ်ငန်းတွေအတွက် အဓိက ရည်ရွယ်တယ်။ Transaction အကောင့်အသတ်မရှိဘူး။ လိုသလောက် ထတ်ယူလို ရတယ်။ Overdraft လိုခေါ်တဲ့ ကိုယ့်မှုရှိတာထက် (အကောင့်အသတ်တော့ရှိတာပေါ့) ပိုထုတ်လို ရတယ်။ Current account ကို ဘဏ်က အတိုးပေးလေ့မရှိဘူး။

SavingAccount နဲ့ CurrentAccount ကလပ်စုံကို အောက်ပါအတိုင်း သတ်မှတ်နိုင်ပါတယ်။ ကလပ်စုံတဲ့ အပိုင်းယုံကြည့်ရင် တူညီတဲ့အပိုင်းတွေ ပါဝင်နေတာ တွေ့ရမှာပါ။ နှစ်ခုလုံးမှာ deposit မက သင်က တူတူပါဘူး။ \_holder, \_balance, \_acc\_number မေးရိုရော့လဲတွေ ပါဝင်တာချင်းလည်း တူ

တယ်။ ကွားချက်တချို့ကိုလည်း တွေ့ရပါတယ်။ withdraw မက်သင် နှစ်ခု မတူကြဘူး။ \_txn\_cnt\_tot, \_txn\_amt\_tot နဲ့ provide\_interest တို့ကို SavingAccount မှာပဲ တွေ့ရတယ်။ တစ်ဖက် မှာလည်း \_overdraft\_amt နဲ့ is\_overdrafted တို့ကိုတော့ CurrentAccount မှာ တွေ့မှုဖြစ်ပြီး SavingAccount မှာ မပါပြန်ဘူး။

```
class SavingAccount:
    def __init__(self, holder, acc_number, balance):
        self._holder = holder
        self._acc_number = acc_number
        self._balance = balance
        self._txn_cnt_tot = 0
        self._txn_amt_tot = Decimal(0.00)

    def deposit(self, amt):
        if amt <= Decimal(0.00):
            raise ValueError('Invalid amount for deposit!')
        self._balance += amt

    def withdraw(self, amt):
        if self._txn_amt_tot >= Decimal(500_000.00):
            raise ValueError('Daily withdraw limit exceeds!')
        if self._txn_cnt_tot >= 5:
            raise ValueError('Daily transaction count exceeds!')
        if amt > self._balance:
            raise ValueError('Not enough balance!')
        self._balance -= amt

    def provide_interest(self):
        """complex logic for interest calculation"""
        pass

class CurrentAccount:
    def __init__(self, holder, acc_number, balance):
        self._holder = holder
        self._acc_number = acc_number
        self._balance = balance
        self._overdraft_amt = Decimal(1_000_000.00)

    def deposit(self, amt):
        if amt <= Decimal(0.00):
            raise ValueError('Invalid amount for deposit!')
        self._balance += amt

    def withdraw(self, amt):
        if amt > (self._balance + self._overdraft_amt):
            raise ValueError('Not enough balance!')
```

```

    self._balance -= amt

    def is_overdrafted(self):
        return self._overdraft_amt < Decimal(0.00)

```

ဒီကလပ်နှစ်ခုမှာ ထပ်နေတဲ့ တူညီတဲ့ကုဒ်တွေကို တစ်ခါပဲ ရေးဖို့လိုမယ်ဆိုရင် ပိုပြီးအဆင်ပြမှာ ပါ။ ဒီလိုအခြေအနေမျိုးမှာ inheritance က ဘယ်လို အထောက်အကူပြုလဲ ကြည့်ရအောင်။ ထပ်နေတဲ့ တူညီတဲ့ကုဒ်တွေကို ဘုတုတ်လိုက်ပြီး ကလပ်စာစ်ခု သတ်မှတ်ပါမယ်။

```

from decimal import Decimal

class Account:
    def __init__(self, holder, acc_number, balance):
        self._holder = holder
        self._acc_number = acc_number
        self._balance = balance

    def deposit(self, amt):
        if amt <= Decimal(0.00):
            raise ValueError('Invalid amount for deposit!')
        self._balance += amt

```

CurrentAccount နဲ့ SavingAccount က တူညီတဲ့ အပိုင်းတွေ Account ကလပ်မှာ ခွဲထုတ်ထား တာ သတိပြုကြည့်ပါ။ ဒီအပိုင်းတွေကို inherit လုပ်ယူပါမယ်။

```

class CurrentAccount(Account):
    def __init__(self, holder, acc_number, balance, overdraft_amt):
        super().__init__(holder, acc_number, balance)
        self._overdraft_amt = overdraft_amt

    def withdraw(self, amt):
        if amt > (self._balance + self._overdraft_amt):
            raise ValueError('Not enough balance!')
        self._balance -= amt

    def is_overdrafted(self):
        return self._balance < Decimal(0.00)

```

CurrentAccount က Account ကို inherit လုပ်မှုဖြစ်တော်ကြာင့် ကလပ်စာတ်မှတ်တဲ့အပါ အခုလို ရေးရပါမယ်

```
class CurrentAccount(Account):
```

Account ကို superclass လိုပေါ်ပြီး သူဆီကနေ ဆက်ခံယူမဲ့ CurrentAccount ကို subclass လို ပေါ်ပေါ်တယ်။ ပေးမဲ့ကလပ်စာ superclass ၊ ယူမဲ့ ကလပ်စာ subclass ပေါ့။ parent နဲ့ child လို လည်းကောင်းတယ်။

Subclass initialize လုပ်တဲ့အခါ superclass အပိုင်းကိုလည်း initialize လုပ်ဖို့လိုတယ်။ ဒါ

အတွက် superclass `__init__` ကို အခုလို ခေါ်ရပါတယ်

```
# call __init__ method of the superclass
super().__init__(holder, acc_number, balance)
```

`_holder`, `_acc_number`, `_balance` instance variable တွေက ဆက်ခံရရှိထားတာပါ။ ငှါးတို့ ကို initialize လုပ်ဖို့အတွက် superclass initializer ကို ခေါ်ရပါမယ်။ Subclass က တိုက်ရှိရှိ မလုပ်ရပါဘူး။ Subclass ကိုယ်ပိုင် instance variable တွေကိုတော့ ပုံမှန်အတိုင်း initialize လုပ် နိုင်တယ်။

```
self._overdraft_amt = overdraft_amt
```

deposit မက်သဒ်ကို superclass ကနေ CurrentAccount က ဆက်ခံရရှိမယ်။ withdraw နဲ့ is\_overdrafted က ကိုယ်ပိုင်ရှိမယ်။ ဒီမက်သဒ်နှင့်ခုမှာ `self._balance` ကို သုံးထားတာ တွေ့ရမှာ ပါ။ ဆက်ခံရရှိတဲ့ instance variable တွေကို subclass မက်သဒ်ထဲမှာ `self` နဲ့ ရည်ညွှန်းအသုံးပြု လို့ ရတယ်။

SavingAccount ကိုအောက်ပါအတိုင်း သတ်မှတ်ပါတယ်။ တစ်နောက် transaction အကြိမ် အရေအတွက်နဲ့ ထုတ်ယူတဲ့ ပမာဏအတွက် `_txn_cnt_tot` နဲ့ `_txn_amt_tot` instance variable တွေ ထပ်ဖြည့်ထားတယ်။ (`_txn_cnt_tot` နဲ့ `_txn_amt_tot` ကို တစ်နောက်ကျန်ဆုံးတိုင်း သုညဖြစ် အောင် ပြန်လုပ်ထားဖို့ လုပ်ပါဝိမှုမယ်။ အခုံပမား ဒီကိစ္စအတွက် ထည့်မစဉ်းစားပါဘူး)။

```
class SavingAccount(Account):
    def __init__(self, holder, acc_number, balance):
        super().__init__(holder, acc_number, balance)
        self._txn_cnt_tot = 0
        self._txn_amt_tot = Decimal(0.00)

    def withdraw(self, amt):
        if self._txn_amt_tot >= Decimal(200_000.00):
            raise ValueError('Daily withdraw limit exceeds!')
        if self._txn_cnt_tot >= 30:
            raise ValueError('Daily transaction count exceeds!')
        if amt > self._balance:
            raise ValueError('Not enough balance!')
        self._balance -= amt
        self._txn_cnt_tot += 1
        self._txn_amt_tot += amt

    def provide_interest(self):
        """complex logic for interest calculation"""
        pass
```

Saving နဲ့ current အပြင် သတ်မှတ်ကာလအတွင်း မထုတ်ဘဲထားရတဲ့ fixed deposit ၊ အသေး စား လုပ်ငန်းတွေအတွက် SME စတဲ့ အခြားအကောင် အမျိုးအစားတွေလည်း ရှိတယ်။ ဒီအကောင့်တွေ နဲ့ ဆက်ဆိုင်တဲ့ ကလပ်စံတွေကလည်း Account ကို inherit လုပ်နိုင်ပါတယ်။

## ၁၀.၂ Overriding

Superclass ကနေ ဆက်ခံရရှိထားတဲ့ မက်သဒ်ကို subclass မှာ ပြင်ဆင်သတ်မှတ်တာကို *method overriding* လို့ ခေါ်ပါတယ်။ Deposit လုပ်တဲ့အခါ current account က အနည်းဆုံး ဆယ်သိန်း ဖြစ်မှ လက်ခံတယ်ဆိုပါစို့။ ဆက်ခံ ရထားတဲ့ မူလ deposit မက်သဒ်က ဒီလိုအပ်ချက်နဲ့ မကိုက်ညီဘူး။ အခုလို အခြေအနေမျိုးမှာ ဆက်ခံယူမဲ့ CurrentAccount ဟာ deposit မက်သဒ်ကို လက်ရှိလိုအပ်ချက်နဲ့ ကိုက်ညီအောင် override လုပ်နိုင်ပါတယ်။

```
class CurrentAccount(Account):
    def __init__(self, holder, acc_number, balance):
        super().__init__(holder, acc_number, balance)
        super._holder = holder
        self._overdraft_amt = Decimal(1_000_000.00)

    # override deposit method
    def deposit(self, amt):
        if amt <= Decimal(1_000_000.00):
            raise ValueError('Deposit at least 1,000,000 to current acc!')
        self._balance += amt

    # other methods
```

မက်သဒ်တစ်ခုကို ဆက်ခံရရှိတဲ့အတိုင်း အသုံးပြုလို အဆင်မပြေတဲ့အခါ subclass က override လုပ်နိုင်ပါတယ်။ Subclass အများစုက ဆက်ခံရရှိတဲ့အတိုင်း အသုံးပြုလိုရပြီး တချို့ အနည်းစုကပဲ override ရတဲ့အခါ ကုဒ်တွေထပ်နေတာ သိသိသာသာ လျော့သွားမှုပါ။

ဆက်ခံရရှိတဲ့အတိုင်း အသုံးပြုလိုရနိုင်ပေမဲ့ optimization အတွက် override လုပ်ရတာလည်း ရှိတယ်။ Polygon ပါတ်လည်နားရာတဲ့ နည်းလမ်းကို rectangle အတွက်လည်း သုံးလို့ရပေမဲ့ rectangle အတွက် သီးသန့်ဖော်မြှုပ်လာ  $2 \times (width + height)$  က ပိုရှင်းလင်းပြီး တွက်ချက်ရပိုမြန်မှာ အမှန်ပါပဲ။ ဒါကြောင့် စွမ်းဆောင်ရည်အတွက် Rectangle ကလပ်စုံမှာ perimeter ကို override လုပ်ဖို့ ဆုံးဖြတ်နိုင်ပါတယ်။

```
import math

class Polygon:
    def __init__(self, vertices):
        self.vertices = vertices

    def perimeter(self):
        perimeter = 0
        num_vertices = len(self.vertices)
        for i in range(num_vertices):
            x1, y1 = self.vertices[i]
            x2, y2 = self.vertices[(i + 1) % num_vertices]
            perimeter += math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
        return perimeter
```

```

class Rectangle(Polygon):
    def __init__(self, width, height):
        # Define vertices for a rectangle assuming bottom-left corner at (0, 0)
        vertices = [(0, 0), (width, 0), (width, height), (0, height)]
        super().__init__(vertices)
        self.width = width
        self.height = height

    def perimeter(self):
        # Overriding the perimeter method for performance
        return 2 * (self.width + self.height)

```

Inheritance နဲ့ method overriding ကို ပရိုဂရမ်တစ်ခုရဲ့ စထရှက်ချာကို ထိန်းကွာပေးဖို့ အသုံးပြုတာကိုလည်း တရာ့လိုက်ဘရီတွေမှာ တွေ့ရတယ်။ ဥပမာ Arcade လိုက်ဘရီနဲ့ ဂိမ်းရေးတဲ့အခါ သူမှာပါတဲ့ ကလပ်စွေတွေကို inherit လုပ်ပြီး ကိုယ်လိုချင်တဲ့ ပုံစံနဲ့ အလုပ်လုပ်အောင် မက်သံတွေကို လုံအပ်သလို override လုပ်ပေးရတယ်။ ဒီအခန်း နောက်ပိုင်းမှာ Arcade လိုက်ဘရီနဲ့ ဂိမ်းလေးတစ်ခု တည်ဆောက်တဲ့အခါ တွေ့ရမှာပါ။

## ၁၀.၃ Multilevel Inheritance

ကလပ်စွေ A ကို B က ဆက်ခံထားမယ်။ တစ်ခါ B ကို C က ထပ်ဆင့် ဆက်ခံယူမယ်။ ဒီလို အဆင့်ဆင့် inherit လုပ်တာကို multilevel inheritance လိုခေါ်တယ်။ ကလပ်စွေ C ဟာ B နဲ့ A နှစ်ခုလုံးရဲ့ attribute နဲ့ မက်သံတွေကို ရရှိမှာပါ (A မှာ ရို့တာတွေကို B ကနေတစ်ဆင့် ရတာ)။ ရှေ့က ဥပမာ မှာ inheritance level နှစ်ခုတွေရှုတယ်။ အပေါ်အဆင့်မှာ Account, အောက်တစ်ဆင့်မှာ သူကိုဆက်ခံယူထားတဲ့ SavingAccount နဲ့ CurrentAccount ရှိတယ်။ SavingAccount နဲ့ CurrentAccount ကို ဆက်ခံထားတဲ့ ကလပ်စွေ နောက်တစ်ဆင့်ရှိလာရင် သုံးဆင့်ဖြစ်သွားမှာပါ။

```

class SavingAccount(Account):
    def __init__(self, holder, acc_number, balance):
        super().__init__(holder, acc_number, balance)
        self._txn_cnt_tot = 0
        self._txn_amt_tot = Decimal(0.00)

    def provide_interest(self):
        """complex logic for interest calculation"""
        pass

class SuperSavingAccount(SavingAccount):
    def __init__(self, holder, acc_number, balance):
        super().__init__(holder, acc_number, balance)

    def withdraw(self, amt):
        if self._txn_amt_tot >= Decimal(200_000.00):
            raise ValueError('Daily withdraw limit exceeds!')
        if self._txn_cnt_tot >= 30:
            raise ValueError('Daily transaction count exceeds!')

```

```

if amt > self._balance:
    raise ValueError('Not enough balance!')
self._balance -= amt
self._txn_cnt_tot += 1
self._txn_amt_tot += amt

class FamilySavingAccount(SavingAccount):
    def __init__(self, holder, acc_number, balance):
        super().__init__(holder, acc_number, balance)

    def withdraw(self, amt):
        if self._txn_amt_tot >= Decimal(600_000.00):
            raise ValueError('Daily withdraw limit exceeds!')
        if self._txn_cnt_tot >= 30:
            raise ValueError('Daily transaction count exceeds!')
        if amt > self._balance:
            raise ValueError('Not enough balance!')
        self._balance -= amt
        self._txn_cnt_tot += 1
        self._txn_amt_tot += amt

```

## ၁၁.၅ Class Hierarchy and UML

【ဖြည့်စွက်ရန်】

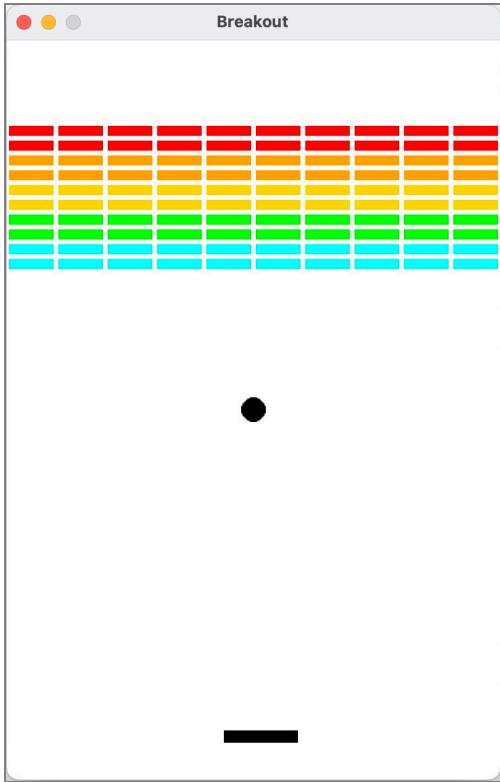
## ၁၁.၆ Is-A Relationship and Inheritance

【ဖြည့်စွက်ရန်】

## ၁၁.၇ အသုံးချုပ်မာ (၁) Breakout Game

ဒီအခန်းအတွက် အသုံးချုပ်မာက Breakout လိုခေါ်တဲ့ ရှိုးရှင်းတဲ့ ဂိမ်းလေးတစ်ခု တည်ဆောက်ပါမယ်။ Breakout ဂိမ်းဟာ ကမ္ဘာကျော် Apple ကုမ္ပဏီ ပူးတွဲတည်ထောင်သူ Steve Wozniak ဒီခိုင်းလုပ်ခဲ့တဲ့ ကန္တဝိုင်းတစ်ခု ဖြစ်ပါတယ်။ ဗားရှင်းတွေအမျိုးမျိုး ရှိုပေါ့ အစိကအနှစ်သာရကတော့ ရွှေနေတဲ့ ဘေးလုံးကို paddle ပြားနဲ့ လိုက်ဖမ်းပြီး အစိအရှိရှိနေတဲ့ အုတ်ခဲတွေ အားလုံးကို ကုန်တဲ့အထိ ဖျက်ရတာပါ။ ဂိမ်းစစ်ချင်း အနေအထားကို ပုံ (၁၀.၁) မှာ တွေ့ရပါမယ်။ အပေါ်ပိုင်းမှာက အုတ်နံရုံပြီး ဘေးလုံးက အောက်ကိုကျေလာမယ်။ paddle ပြားနဲ့ လိုက်ဖမ်းရမယ်။ ဘေးလုံးက paddle, အုတ်ခဲ၊ ဘယ်/ညာ/အပေါ် ဘောင်တွေနဲ့ ဝင်တိုက်ရင် ပြန်ကန်ထွက်တယ်။ အုတ်ခဲတွေက ဘေးလုံးနဲ့ တိုက်မိရင် ပျက်သွားမယ်။ ဘေးလုံးကို မဖမ်းလိုက်နိုင်လို့ paddle ပြားအောက်ဘက် ကျသွားရင် ရှုံးမယ် (အောက်ဘက်ကျသွားရင် ပြန်ကန်မထွက်ဘူး)။ အုတ်ခဲတွေ ကုန်တဲ့ထိ ဘေးလုံးမကျသွားအောင် ထိန်းထားနိုင်ရင် အနိုင်ရမှာဖြစ်တယ်။

ဒီဂိမ်းရေးဖို့ Arcade လိုက်ဘရိုက် အသုံးပြုမှာပါ။ Arcade လိုက်ဘရိုဟာ inheritance ကို အခြေခံထားတယ်။ Window, View, Sprite စဲတဲ့ ကလပ်စွေတွေကို လိုက်ဘရိုကပေးထားတယ်။ ဂိမ်းတစ်ခုရေးတဲ့အခါ ဒီကလပ်စွေတွေကို inherit လုပ်ပြီး လိုအပ်တဲ့ မက်သွ်တွေကို override လုပ်ပေးရှုံးပဲ။



ပုံ ၁၀.၁

လိုက်ဘရိက ပရိုဂရမ်ကို အကြမ်းထည် စထရက်ချာ ချထားပေးတယ်။ အဲဒီ စထရက်ချာထဲမှာ ကိုယ်တိုင် စိတ်ကြိုက် ဖန်တီးချင်တဲ့ နေရာတွေကို ပြင်ဆင်ဖြည့်စက်လိုက်အောင် လုပ်ပေးထားတာလို ယူဆနိုင်တယ်။ ကြိုတင်သိထားဖို့ လိုအပ်တာတွေ တစ်ဆင့်ချင်း အရင်ကြည့်ရအောင်။

### **arcade.Window**

Window ကလပ်စာ့ ကရပ်စစ် window တစ်ခုကို ကိုယ်စားပြုတယ်။ ကရပ်ဖော်ရုပ်ပုံတွေ ဆွဲဖိန့် အန်နီ မေးရှင်း လုပ်ဖို့ လိုအပ်တာတွေ ဒီကလပ်စာ့ ထောက်ပံ့ပေးထားတယ်။ ဒါအပြင် မောက်စာ့၊ ကီးဘုဒ် စတဲ့ input device တွေနဲ့ ဂိမ်းကို ကွန်ထရိုးလုပ်လိုက်ရအောင် နည်းလမ်းတွေလည်း ပါတယ်။

Arcade ပရိုဂရမ်တစ်ခုကို run တဲ့အခါ Window မှာ ပါတဲ့ on\_draw မက်သဒ်ကို တစ်စက်နှင့် အကြိမ် ပြောက်ဆယ် ခေါ်ပေးတယ်။ သူနှင့်အတိုင်း on\_draw မက်သဒ်က ဘာမှ လုပ်ဆောင်မပေးဘူး။ ကိုယ့်လိုချင်တဲ့ ကရပ်စစ်ပုံ ဖော်ဖို့ Window ကို inherit လုပ်ပြီး override လုပ်ပေးရမယ်။ အပြာရောင် ဘောလုံးတစ်ခု အလယ်ဗဟိုမှာ ဆွဲမယ်ဆိုပါစို့ ...

```
# File: arcade_blue_circle.py
import arcade
from arcade.color import *

WIN_WIDTH = 640
WIN_HEIGHT = 480
```

```

class MyGame(arcade.Window):

    def __init__(self, width, height, title):
        super().__init__(width, height, title)
        arcade.set_background_color(arcade.color.ALMOND)

    def on_draw(self):
        self.clear()
        arcade.draw_circle_filled(WIN_WIDTH / 2, WIN_HEIGHT / 2, 20, BLUE)

window = MyGame(WIN_WIDTH, WIN_HEIGHT, "Show Blue Circle")
arcade.run()

```

on\_draw ကို အခဲလို override လုပ်နိုင်ပါတယ်။ ဒီပရိုဂရမ် run တဲ့အခါ on\_draw ကို တစ်စက်နံတိုင်း တစ်စက်နံတိုင်းမှာ အကြိမ်ခြောက်ဆယ် (60 frames per second) တောက်လျှောက် ခေါ်နေမှုပါ။ ပရိုဂရမ် window မပိတ်မချင်းပေါ့။ ဒီလိုဖြစ်အောင် Arcade လိုက်ဘရီက လုပ်ပေးထားတာ။ ဒီအတွက် သီးခြားရေးဖို့ မလိုဘူး။ Window ကလပ်စံကို inherit လုပ်ပြီး on\_draw ကို override လုပ်ပေးရင် ရပြီ။ clear မက်သဒ်ကိုလည်း superclass Window ကနေ ဆက်ခံရထားတာ။ Window ပေါ်မှာ ရှိတဲ့ ဂရပ်ဖော်အားလုံးကို ရှင်းပေးပြီး သတ်မှတ်ထားတဲ့ နောက်ခံရောင် ဖြည့်ပေးတဲ့ မက်သဒ်ပါ။ အန်နီမေးရှင်း အတွက် ဒီမက်သဒ်က အရေးကြီးတယ်ဆိုတာ တွေ့ရပါမယ်။

မောက်စ် (သို့) ကိုဘုံးနဲ့ ဘေးလုံးကို ရွှေလိုရအောင်လည်း လုပ်လိုရတယ်။ ခက်ခဲတဲ့ ကိစ္စတွေကို Window က အဆင်သင့် လုပ်ထားပေးတာပါ။ ဒါဟာ အထူးအဆန်း ဖြစ်နေနိုင်ပါတယ်။ ဘယ်လိုများလုပ်ထားလဲ အကြိမ်ဖျဉ်း သဘောတရား နားလည်ချင်ရင် ဒီဥပမာကို လေ့လာကြည့်ပါ။

```

class ConsolePrinter:

    def __init__(self, times):
        self._times = times

    def do_before(self):
        print("Checking if everything is ready.")

    def do_after(self):
        print("Doing cleaning up.")

    def do_my_task(self):
        self.do_before()
        for i in range(self._times):
            self.my_task()
        self.do_after()

    # do nothing
    def my_task(self):
        pass

```

```

class MyConsolePrinter(ConsolePrinter):
    def __init__(self, times, text):
        super().__init__(times)
        self._text = text

    # to print as you want
    def my_task(self):
        print(self._text)

printer = MyConsolePrinter(15, 'Hello')
printer.do_my_task()

```

ConsolePrinter မှာ my\_task ကလဲလို့ ကျန်မက်သခ်တွေအားလုံး အပြည့်အစုံရေးထားတာ တွေရမှာ ပါ။ do\_my\_task က my\_task ကိုခေါ်ထားတယ်။ MyConsolePrinter က ထုတ်ပေးချင်တဲ့ စာသား အတွက် my\_task ကို override လုပ်ထားတာကို ရရှိပါ။ Arcade လိုက်ဘရီ ကလပ်စွဲကလည်း ဒီသဘောတရားပဲ။ ဂိမ်းတစ်ခုအတွက် လိုအပ်ချက်တွေကို ပြင်ဆင်ပေးထားတယ်။ ပုံစံချေပေးထားတယ်။ အသုံးပြုသူက ဒီကလပ်စွဲတွေကို inherit လုပ်ပြီး ဂိမ်းရဲ့ လိုအပ်ချက်အတိုင်း ဖြစ်အောင် မက်သခ်တွေကို override လုပ်ပြီး စိတ်ကြိုက်ပုံဖော်ယူရတာ။

### အနိမ်မေးရင်း

စောင့်က ပရိုဂရမ်မှာ စက်စိုင်း ညာဘက်ဘက်အပေါ်ကို ရွှေသွားအောင် အန်နီမေးရှင်း လုပ်မယ်ဆိုပါ စိုး။ on\_draw လုပ်တဲ့ အခါတိုင်းမှာ လက်ရှိ x နဲ့ y တန်ဖိုးကို နည်းနည်းချင်း တိုးပေးနိုင်ပါတယ်။ နမူနာ အနော့ x ကို 2 pixels နဲ့ y ကို 1 pixel တိုးပေးပါမယ်။

```

# File: arcade_moving_circle.py
import arcade
from arcade.color import *

WIN_WIDTH = 640
WIN_HEIGHT = 480
RADIUS = 15

class MyGame(arcade.Window):

    def __init__(self, width, height, title):
        super().__init__(width, height, title)
        self._circle_x = WIN_WIDTH / 2
        self._circle_y = WIN_HEIGHT / 2
        arcade.set_background_color(ALMOND)

    def on_draw(self):
        self.clear()
        self._circle_x += 2      # move 2 pixels to the right

```

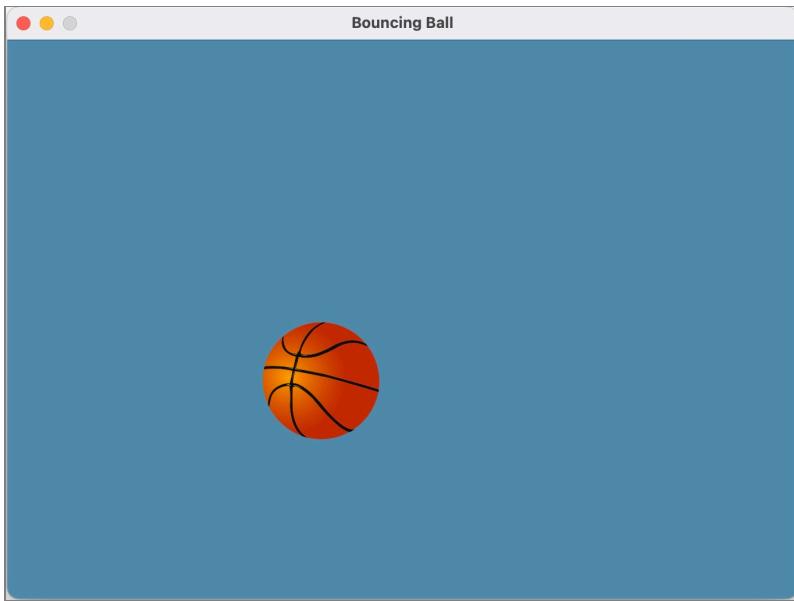
၂၁၂

```
    self._circle_y += 1      # move 1 pixels upwards
    arcade.draw_circle_filled(self._circle_x,
                               self._circle_y,
                               RADIUS,
                               BLUE)

window = MyGame(WIN_WIDTH, WIN_HEIGHT, "Moving Circle")
arcade.run()
```

## Sprite

Sprite ကလပ်က ရုပ်ပဲ (image) တွေကို ဖော်ပြနိုင်တယ်။ ရုပ်ပဲအတွက် .jpg, .png, .bmp အစရိတဲ့ image file ဖော်မထဲအမျိုးမျိုး ထောက်ပံ့ပေးတယ်။ ဂိမ်း ကတ်ကောင်တော့ အရာဝတ္ထုတွေ ပုံဖော်စိုင်စွဲ အခိုက်ကျတဲ့ ကလပ်စပါ။ တစ်ခုနဲ့တစ်ခု ဝင်တိုက်တာ (collision) ၊ အပေါ်/အောက် (သို့) ဘယ်/ညာ လှည့်တာ၊ ပစ္စည်းတွေကို အုပ်စုလိုက် ရွှေ့လျှော်စွာ စတာတွေကို အလွယ်တကူ လုပ်ဆောင်လို့ရအောင် Sprite ကလပ်က ထောက်ပံ့ပေးထားတယ်။ Window ကောင် တစ်ဖက်ဖက်နဲ့ ဝင်တိုက်ရင် ဘေးလုံး က ပြန်ကန်ထဲက်တဲ့ အန်နီမေးရှင်းဥပမာကို ကြည့်ရအောင် ...



ပုံ ၁၀.၂ Ball Sprite ဥပမာ

```
# File: arcade_bouncing_ball.py
import arcade
SCREEN_WIDTH = 680
SCREEN_HEIGHT = 480

class Ball(arcade.Sprite):
```

```

def update(self):
    # rotate the sprite
    self.angle += 1
    # move the sprite
    self.center_x += self.change_x
    self.center_y += self.change_y

    if self.left < 0:
        self.change_x *= -1

    if self.right > SCREEN_WIDTH:
        self.change_x *= -1

    if self.bottom < 0:
        self.change_y *= -1

    if self.top > SCREEN_HEIGHT:
        self.change_y *= -1

```

center\_x, center\_y ကဲ Sprite ပဟိုမှတ် တည်နေရာ။ change\_x, change\_y က အန်စီမံချက်များ ရွှေ့ပေးရမဲ့ x နဲ့ y ပမာဏ။ Sprite အမြန်ဆုံးနဲ့ ဦးတည်ရာကို ဒီနှစ်ခု ရဲ့ တန်ဖိုးနဲ့ ထိန်းလိုပေးနိုင်တယ်။ left, right, top, bottom attribute တွေကတေသူ Sprite ရဲ့ ဘယ်/ညာ x တန်ဖိုးနဲ့ အထက်/အောက် y တန်ဖိုးတွေကို ဖော်ပြတယ်။ ဒါ ပေရှိရောကဲလဲတွေ အားလုံး ကို superclass Sprite ကနေ ဆက်ခံရရှိထားတာ။

ဘယ် (သို့) ညာဘက် ဘောင်နဲ့ ထိရင် change\_x ကို အပေါင်းအနှစ် ပြောင်းပြန် လုပ်ပေးရပါမယ်။ အထက် (သို့) အောက် ဘောင်နဲ့ ထိရင် change\_y ကို ပြောင်းပြန် လုပ်ပေးရပါမယ်။ ပုံ (၁၀.၃) မှာ ကြည့်ပါ။

Ball ကလပ်စုံ \_\_init\_\_ မက်သွေ့ မပါဘူး။ (Subclass မှာ \_\_init\_\_ မပါရင် အော့ဂျက်ဖော်တဲ့ အကိုယ်စားအခါ ဆက်ခံရရှိထားတဲ့ superclass \_\_init\_\_ ကိုပဲခေါ်ပါတယ်။)

ဒါ Sprite ကို ဘယ်လို အသုံးပြုလဲ ဆက်ကြည့်ရအောင်

```

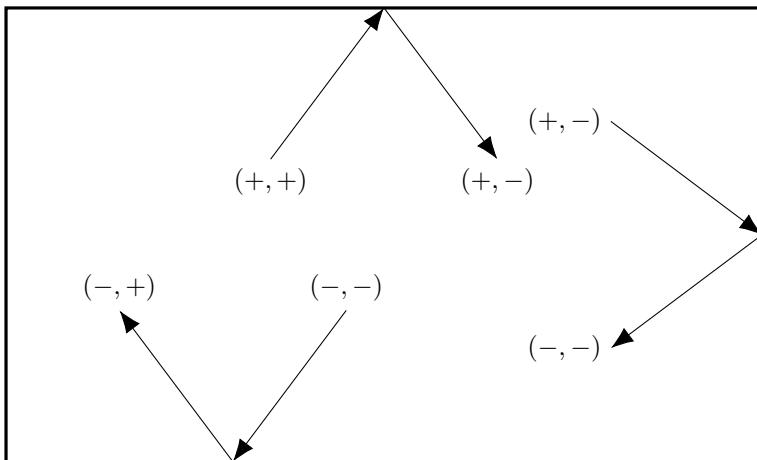
# File: arcade_bouncing_ball.py
class MyGame(arcade.Window):

    def __init__(self, width, height, title):
        super().__init__(width, height, title)

        self._ball = Ball('ball.png', 0.2)

        self._ball.center_x = SCREEN_WIDTH / 2
        self._ball.center_y = SCREEN_HEIGHT / 2
        self._ball.change_x = 2
        self._ball.change_y = 1
        arcade.set_background_color(arcade.color.AIR_FORCE_BLUE)

```



ပုံ ၁၀.၃ `change_x` နှင့် `change_y` လက္ခဏာ ပြောင်းလဲပုံ

```

def on_draw(self):
    self.clear()
    self._ball.draw()

def on_update(self, delta_time):
    self._ball.update()
  
```

```

window = MyGame(SCREEN_WIDTH, SCREEN_HEIGHT, "Bouncing Ball")
arcade.run()
  
```

Sprite အော်ဂျက် ဖန်တီးတဲ့အခါ ရုပ်ပုံစိုင် နံမည်နဲ့ ချုံ/ချဲ တန်ဖိုး (scale factor) ထည့်ပေး ရပါတယ်။ နှင့် ball.png ဖိုင် ဆယ်ပုံနှစ်ပုံအရွယ်အစားနဲ့ ဘေးလုံးပုံ အချင့် အချင့်

```

self._ball = Ball('ball.png', 0.2)
  
```

ဖန်တီးတယ်။ ရုပ်ပုံစိုင်က ကုဒ်ပိုင်နဲ့ ဖို့ပါတစ်ခုထဲမှာ ရှိရပါမယ်။ ဒါက အလွယ်နည်းကို ပြောတာပါ။ အခြား ဖို့ပါမှာ ထားလို့ရပေမဲ့ နည်းနည်း ပို့ရှုပ်လို့

Sprite ရွှေလျားတာနဲ့ collision ဖြစ်တဲ့ ကိစ္စတွေအတွက် အဓိက ဂိမ်းလေ့ဂျွဲ့ကို on\_update မက်သဒ်မှာ ရေးပေးရမယ်။ ဒါလည်းပဲ တကောက်က ဆက်ခံထားတဲ့ on\_update ကို override လုပ်တာပါ။ ဒီမက်သဒ်ကိုလည်း တစ်စက်နဲ့ အကြိမ်ခြောက်ဆယ် အလိုအလျောက် ခေါ်ပေးပါတယ်။ ဒါပေမဲ့ on\_draw နဲ့က ရည်ရွယ်ချက်ချင်း မတူပါဘူး။ on\_draw က စခရင်ကိုပဲ refresh လုပ်ပေးတာ။ တစ်နည်းအားဖြင့် window ပေါ်မှာ ဂရပ်ဖစ်ပုံပဲ ဖော်ပေးတာ။ ဂိမ်းလေ့ဂျွဲ့ကို သက်ဆိုင်တာတွဲ မလုပ်ဘူး။ on\_update ကျတော့ ပုံဆွဲတဲ့ ကိစ္စတွေကို မလုပ်ဘူး။ ဂိမ်းလေ့ဂျွဲ့သီးသန့် လုပ်ဆောင်တယ်။ ဒီမက်သဒ်နှစ်ခု လုပ်ဆောင် ပေးတဲ့ တာဝန်ကို ခွဲခြားထားရပါမယ်။ ဂိမ်းမှာပါဝင်တဲ့ Sprite အားလုံးရဲ့ state ကို update လုပ်ပေး ခြင်းဟာ on\_update ရဲ့ အဓိကတာဝန်တွေထဲက တစ်ခုအပါအဝင် ဖြစ်တယ်။ အချောမာမှာ ဘေးလုံးရဲ့ state ကို

```

self._ball.update()
  
```

ခေါ်ပြီး update လုပ်ပေးထားတယ်။

`on_update` မက်သဒ် `delta_time` ပါရာမိတာကို ရှင်းပြန့် ကျဉ်ပါသေးတယ်။ `on_update` ကို ခေါ်တဲ့အခါ နောက်ဆုံးခါ ခဲ့တဲ့ အချိန်နဲ့ အချိန်ကြား ကွဲဟချက်ကို `delta_time` အနေနဲ့ ထည့်ပေးတယ်။ Arcade လိုက်ဘရဲ့ နောက်ကွယ်က အလုပ်လုပ်ပုံ သဘောတရားအရ ထည့်ထားတဲ့ ပါ ရာမိတာဖြစ်ပြီး လိုက်ဘရဲ့ သုံးတဲ့သူအနေနဲ့ အသေးစိတ်သိဖို့ မလိုအပ်ပါဘူး။ `on_update` ကို override လုပ်တဲ့အခါ `delta_time` ပါရာမိတာ မကျဉ်ခဲ့ရင် ရပါပြီ။

Velocity, `change_x` နှင့် `change_y` ဆက်စပ်ပုံ



ပုံ ၁၁၄

မူလ  $p_1$  အမှတ်ကနေ ညာဘက်ကို  $\Delta x$ , အထက်ကို  $\Delta y$  (နှစ်ခုလုံး အပေါင်းတန်ဖိုး) ရွှေ့ရင်  $p_2$  အမှတ်ကို ရောက်သွားမှာပါ (ဘယ် ပုံ)။ မူလ  $p_1$  အမှတ်ကနေ ညာဘက်ကို  $\Delta x$  (အပေါင်း)၊ အောက် တည့်တည့်ကို  $\Delta y$  (အနှစ်) ရွှေ့ရင်  $p_2$  အမှတ်ကို ရောက်သွားမှာပါ (ညာ ပုံ)။ Sprite မှာ `change_x`, `change_y` ဟာ  $\Delta x, \Delta y$  ပါပဲ။  $p_1$  က  $(x_1, y_1)$ ,  $p_2$  က  $(x_2, y_2)$  ဖြစ်တယ်ဆိုရင်

$$\begin{aligned} x_2 &= x_1 + \Delta x \\ y_2 &= y_1 + \Delta y \end{aligned}$$

Frames per second (တစ်စက်နဲ့ `update/refresh` လုပ်တဲ့ အကြိမ်) နဲ့ velocity ဆက်စပ် နေမယ်ဆိုတာ သိသာပါတယ်။  $\Delta x = 2, \Delta y = 1$  ဆိုပါစို့

$$\begin{aligned} \Delta s &= \sqrt{\Delta x^2 + \Delta y^2} \\ &= \sqrt{2^2 + 1^2} \\ &= \sqrt{5} \\ &\approx 2.24 \end{aligned}$$

တစ်စက်နဲ့ အကြိမ်ခြောက်ဆယ် `update` လုပ်မယ်ဆိုရင် တစ်စက်နဲ့အတွင်းမှာ စုစုပေါင်း အကွာအဝေး  $60 \times 2.24 = 134.4$  pixels ရွှေ့မှာပါ။ မျှော်းဖြောင့်အတိုင်း ရွှေ့မှာ။ ဦးတည်ရာက  $\Delta x, \Delta y$  အပေါင်း/အနှစ် ပေါ့ မူတည်ပါမယ်။

## Event Handling

မောက်စဲ၊ ကီးဘုံး၊ joystick စတဲ့ input device တွေနဲ့ ဂိမ်းကို တိန်းချုပ်ဖို့အတွက်လည်း Window က လပ်စဲက လွှာယ်အောင်လုပ်ထားတယ်။ `on_mouse_motion`, `on_mouse_press`, `on_key_press` စတဲ့ မက်သဒ်တွေကို override လုပ်ရုပ်ပါ။ အဖြစ်အပျက် တစ်စုံတစ်ခု (*event*) ကို ပရိုဂရမ်က တွဲပြန် လုပ်ဆောင်ပေးတာကို `event handling` လို့ ခေါ်တယ်။ ကီးဘုံး၊ ကီးနှုပ်/လွတ် လိုက်တာ၊ မောက်စဲ

## ၂၀၃

ကလစ်နှင့်/လွှတ် လိုက်တာ၊ မောက်စ်ရွှေတာ စတာတွေဟာ event ဥပမာတချို့ ဖြစ်တယ်။ ဒါတွေကို ပရိုဂရမ်က တို့ပြန်လုပ်ဆောင်ချင်တဲ့အခါ event handling ကို သုံးရပါတယ်။

ဒါက ဘေးလုံးကို မောက်စ်နဲ့ ရွှေတဲ့ နမူနာပါ။ မောက်စ်ကို နှင့်ထားပြီး ရွှေရတာ (dragging) မဟုတ်ဘူး။ ဒီတိုင်း ပိုင်တာရွှေတဲ့ နောက်ကို ဘေးလုံးက လိုက်နေမှာပါ။

```
# File: arcade_m_move.py
class MyGame(arcade.Window):

    def __init__(self, width, height, title):
        super().__init__(width, height, title)
        self._ball = arcade.Sprite('ball.png', 0.2)
        self._ball.center_x = SCREEN_WIDTH / 2
        self._ball.center_y = SCREEN_HEIGHT / 2
        arcade.set_background_color(arcade.color.AIR_FORCE_BLUE)

    def on_draw(self):
        self.clear()
        self._ball.draw()

    def on_mouse_motion(self, x: int, y: int, dx: int, dy: int):
        self._ball.center_x = x
        self._ball.center_y = y
```

မောက်စ်ကို ရွှေရင် မောက်စ်ပိုင်တာရွှေနေသ၍ on\_mouse\_motion ကို တစ်ခါပြီးတစ်ခါ အဆက်မပြတ် ခေါ်နေမှာပါ။ မောက်စ် ရပ်လိုက်ရင် ဒီမက်သဒ်ခေါ်တာလည်း ရပ်သွားမယ်။ ဒီလိုဖြစ်အောင် Window က လုပ်ပေးထားတာ။ x နဲ့ y က လက်ရှိ မောက်စ်ပိုင်တာရဲ့ တည်နေရာ။ dx က ဒီမက်သဒ်ကို နောက်ဆုံး ခေါ်ခဲ့တဲ့ အချိန်နဲ့ အခုံခေါ်တဲ့ အချိန်အတွင်း ပြောင်းလဲသွားတဲ့ x ကွာဟာချက်။ dy က ဒီမက်သဒ်ကို နောက်ဆုံး ခေါ်ခဲ့တဲ့ အချိန်နဲ့ အခုံခေါ်တဲ့ အချိန်အတွင်း ပြောင်းလဲသွားတဲ့ y ကွာဟာချက်။ နောက်ဆုံး ခေါ်ခဲ့တဲ့က မောက်စ်က  $(x_1, y_1)$  မှာ ရှိခဲ့တယ် အခုံခေါ်တဲ့အချိန်  $(x_2, y_2)$  မှာ ဆိုပါစို့။  $dx = x_2 - x_1$ ,  $dy = y_2 - y_1$  ဖြစ်တယ်။

မောက်စ် ကလစ်နှင့်တဲ့အခါ တစ်ခုခု လုပ်မယ်ဆိုရင် on\_mouse\_press ကို override လုပ်ရပါမယ်။ ဒီမက်သဒ်ကိုတော့ ကလစ်နှင့်တော့မှပဲ ခေါ်မှာပါ။ မောက်စ် ဘယ်ဘက် ကလစ်နှင့်လိုက်တဲ့နေရာကို ဘေးလုံး (ချက်ချင်း)ရောက်စေချင်ရင် အခုံလို့ ...

```
# File: arcade_m_press.py
def on_mouse_press(self, x, y, button, modifiers):
    if button == arcade.MOUSE_BUTTON_LEFT:
        self._ball.center_x = x
        self._ball.center_y = y
```

ကီးဘုဒ်နဲ့ ထိန်းချင်ရင် on\_key\_press ကို override လုပ်ရပါမယ်။ arcade\_k\_press.py မှာ လေ့လာကြည့်ပါ။ မောက်စ် နှင့်ထားပြီး ဘေးလုံးကိုရွှေ (dragging) တာကို arcade\_m\_drag.py ဖို့ မှာ ကြည့်နိုင်ပါတယ်။

## arcade.View

arcade.View ဟဲ Window နဲ့ သဘောတရား အတော်လေးဆင်တူပါတယ်။ Window လိုပဲ စခရင် မှာ ဂရပ်ဖစ်ပုံဖော်ထိန္ဒြာ View ကို သုံးလို့ရတယ်။ Event handling အတွက် override လုပ်ရတာတွေ ကလည်း Window နဲ့ တူတူပါပဲ။ ဒါပေမဲ့ View က သူချဉ်း မရပ်တည်နိုင်ပါဘူး။ View ကို ပြပေးဖို့ အတွက် Window တစ်ခုလိုပါတယ်။ Window တစ်ခုတည်းဟာ View အမျိုးမျိုးကို အလုပ်ကျ လျှော့ပြုလိုရတယ်။ ဂိမ်းတစ်ခုမှာ welcome screen, game over screen, pause screen စသည်ဖြင့် ပြပေး ဖို့လိုပါတယ်။ ဒီလို လိုအပ်ချက်မျိုးအတွက်ဆိုရင် Arcade မှာ View ကို အသုံးပြုရမှာပါ။ နှုန္ဓာကြည့်ရင် ပို့ရှင်းသွားပါလိမ့်မယ်။

အောက်ပါကြပ်မှာက ပထမ စစ်ချင်း Click to Start! လို့ ပြနေမှာပါ။ ကလစ်နိုင်လိုက်ရင် ဘေးလုံး က စရွှေ့ပါမယ်။ ကလစ်ထပ်နိုင်လိုက်ရင် ဘေးလုံးရွှေနေတာ ပျောက်သွားပြီး The End စသားပြုပါတယ်။ MsgView က စသားပြုပေးဖို့အတွက်။ MainView က ဘေးလုံး အနှစ်မေးရှင်းအတွက်။ နှစ်ခုလုံး arcade.Window အစား arcade.View ကို inherit လုပ်ထားတာ သတိပြုပါ။

```
# File: arcade_view_switch.py
import arcade
from arcade.color import *

WIN_WIDTH = 400
WIN_HEIGHT = 600

class MainView(arcade.View):
    def __init__(self):
        super().__init__()
        self._circle_x = WIN_WIDTH // 2    # move 2 pixels to the right
        self._circle_y = WIN_HEIGHT // 2   # move 1 pixels upwards
        self._next_view = None

    def on_draw(self):
        self.clear()
        self._circle_x += 1.3  # move 2 pixels to the right
        self._circle_y += 2    # move 1 pixels upwards
        arcade.draw_circle_filled(self._circle_x,
                                  self._circle_y,
                                  20,
                                  BLUE)

    def on_mouse_press(self, _x, _y, _button, _modifiers):
        """ If the user presses the mouse button, start the game. """
        self.window.show_view(self._next_view)

class MsgView(arcade.View):
    def __init__(self, msg):
        super().__init__()
```

జో

```
self._msg = msg
self._next_view = None

def on_draw(self):
    self.clear()
    arcade.draw_text(self._msg,
                     WIN_WIDTH / 2,
                     WIN_HEIGHT / 2,
                     RED,
                     font_size=20,
                     anchor_x="center")

def on_mouse_press(self, _x, _y, _button, _modifiers):
    if self._next_view:
        self.window.show_view(self._next_view)

def main():
    window = arcade.Window(WIN_WIDTH, WIN_HEIGHT, "View Switch")
    end_view = MsgView("The End")
    start_view = MsgView("Click to Start!")
    main_view = MainView()

    start_view._next_view = main_view
    main_view._next_view = end_view
    window.show_view(start_view)
    arcade.run()

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

టోటోగెబ్రాంచల్డీ View కి ప్రథమ Window ర్యాపిమయ్|| Arcade పరిగణితంలో run తోఅట Window తాండ్రాకాటొ స్క్రీన్‌ర్యామ్‌పి|| అట్లి Window కి View ను self.window లో రైస్ట్‌ఫ్స్ట్:అవ్యాప్తి క్రింతయ్ (View కి ఐఏఎంటాట్లి subclass మాలాల్స్: య్యాట్లిగ్రాట్య్)|| Window లు ప్రొవెంట్లి View కి window.set\_view పగ్వాంక్కి వింటమార్ట్రాట్య్|| MsgView లో MainView లు on\_mouse\_press కి ఓట్లి override ల్యాప్టిషిట్య్

```
def on_mouse_press(self, x, y, button, modifiers):
    if self._next_view:
        self.window.show_view(self._next_view)
```

గాలంక్కిబ్బర్చు self.\_next\_view కి బ్రోచ్చేపేమాపి||

ఫోగ్వాంతాంక్కి View అంగ్వ్యాంక్కి అభ్రాంబ్హా ద్వాంక్కిప్రమాపేట్లి Window అప్పీ మ్లాట్ట్ర్యుంతయ్|| ఓ గ్రోండ్ క్రాంక్కి View తాండ్రాంచ్చుంక్కి అట్లాగ్వ్యాంక్కి అంగ్వ్యాంక్కి అభ్రాంబ్హా మింటమార్ట్రాట్య్|| Window అంగ్వ్యాంక్కి అభ్రాంబ్హా మింటమార్ట్రాట్య్||

main မက်သဒ်မှာ Window တစ်ခု နဲ့ View သိုးခု ဖန်တီးထားတယ်။ View တစ်ခုကနေ တစ်ခု ပြောင်းလဲဖို့ အခုလို ချိတ်ဆက်ပေးထားတာ

```
start_view._next_view = main_view
main_view._next_view = end_view
```

တွေ့ရမှာပါ။ start\_view ပေါ်မှာ ကလစ်နိုပ်ရင် main\_view, main\_view ပေါ်မှာ နိုပ်ရင် end\_view ကို ပြေားမှာပါ (on\_key\_press မက်သဒ်နဲ့ ဆက်စပ်ကြည့်ပြီး နားလည်အောင်လုပ်ပါ။)

## ၁၀.၇ Breakout တည်ဆောက်ခြင်း

ဂိမ်း မတည်ဆောက်ခင် ကြိုတင်နားလည်ထားရမဲ့ Arcade သဘောတရား အတော်များများ ရှင်းပြုခဲ့ပြီ။ တကယ် လက်တွေ့ ဂိမ်းရေးဖို့ပဲ ကျော်ပါတော့တယ်။ ပရိုဂရမ် အပြည့်အစုံကို တမျက်နှာ (၂၁၉) မှာ ကြည့်ပါ။ အခု တစ်ပိုင်းချင်း ခွဲထွက် ရှင်းပြပါမယ်။ ဂိမ်းအတွက် လိုအပ်တဲ့ constant တွေကို ပထမဆုံး သတိမှတ်ထားတယ်။ အများစုက တည်နေရာ၊ အရွယ်အစားနဲ့ သက်ဆိုင်တာတွေ။

```
WIN_WIDTH = 400
WIN_HEIGHT = 600

# paddle size
PDL_WIDTH = 60
PDL_HEIGHT = 10
PDL_Y_OFFSET = 30      # distance between paddle and lower edge of window

BALL_RADIUS = 10
BRICKS_PER_ROW = 10
BRICK_ROWS = 10        # number of brick rows
BRICK_GAP = 4           # gap between bricks

# calculate and define constants for brick width and height
BRICK_WIDTH = ((WIN_WIDTH - (BRICKS_PER_ROW - 1) * BRICK_GAP)
               // BRICKS_PER_ROW)
LEFT_MARGIN = (WIN_WIDTH - (BRICK_WIDTH * BRICKS_PER_ROW +
                            BRICK_GAP * (BRICKS_PER_ROW - 1))) // 2
BRICK_HEIGHT = 8

# Window အောက်ဘက် ဘောင်နဲ့ နံရုံအောက်ပြီ အကွာအငေး။
BRICK_Y_OFFSET = 414
# Window ပုံစံမှတ်
CENTER_X = WIN_WIDTH // 2
CENTER_Y = WIN_HEIGHT // 2
```

## အုတ်ခဲ့ခြင်း

setup\_bricks က အုတ်ခဲ့တွေ နေရာတကျ စီပေးတဲ့ ဖန်ရှင်း။ အုတ်ခဲ့ကို SpriteSolidColor နဲ့ ဆွဲ လိုရတယ်။ ဒီကလပ်စက အရောင်အပြည့် ထောင့်မှန်စတုဂံပုံ Sprite ပါပဲ။ image ဖိုင် ရှိဖို့ မလိုဘူး။

ဒိတ္ထားတဲ့ အုတ်ခဲအားလုံးကို SpriteList နဲ့ သိမ်းထားပြီး return ပြန်ပေးထားတယ်။

```
def setup_bricks():
    colors = [CYAN, CYAN, GREEN, GREEN, GOLD, GOLD,
              ORANGE, ORANGE, RED, RED]
    # အုတ်ခဲအားလုံး ထည့်ထားဖို့ SpriteList
    brick_lst = arcade.SpriteList()
    y = BRICK_Y_OFFSET + BRICK_HEIGHT // 2
    # 10 × 10 bricks wall
    for i in range(BRICKS_PER_ROW):
        x = LEFT_MARGIN + BRICK_WIDTH // 2
        for j in range(BRICK_ROWS):
            brick = arcade.SpriteSolidColor(BRICK_WIDTH,
                                              BRICK_HEIGHT,
                                              colors[i])
            brick.center_x = x
            brick.center_y = y
            # အုတ်ခဲတစ်ခုချင်း SpriteList ထဲထည့်
            brick_lst.append(brick)
            x += (BRICK_WIDTH + BRICK_GAP)
        y += (BRICK_HEIGHT + BRICK_GAP)
    return brick_lst
```

SpriteList ထဲမှာ Sprite တွေ တစ်စုတစ်စည်းတည်း သိမ်းထားတာဟာ ဘောလုံးနဲ့ ဝင်တိုက်မိတဲ့ အုတ်ခဲတွေ (တစ်ခုထက်ပိုင်းတယ်) ကို စစ်ထုတ်ဖို့ လွယ်ကူရနေတယ်ဆိတာ ခဏနေ တွေ့ရမှာပါ။ ထဲပေါ်နေရာ အတွက်အချက် အသေးစိတ်နားလည်ချင်ရင် စာမျက်နှာ (၁၀၈) မှ ကျားကွဲက်ခံ ဥပမာကို ကြည့်ပါ။

### ဘောလုံးနှင့် တာထွက် velocity

ဘောလုံးအတွက် Ball ကေလပ်ကို အခုလို သတ်မှတ်ပါမယ်။ ဘောင်တွေကို တိုက်တဲ့အခါ ပြန်ကန်ထွက်အောင် လုပ်တဲ့နည်းလမ်းက ရှုမှာတွေ့ခဲတဲ့ ဥပမာကလိုပါပဲ။ self.left < 0 ဖြစ်ရင် ဘယ်ဘက်ဘောင်နဲ့ တိုက်တာ၊ self.right < WIN\_WIDTH ဆိုရင် ညာဘက်နဲ့တိုက်တာ စသည်ဖြင့်ပေါ့။ အထက်/အောက် ဘောင်နဲ့တိုက်တာလည်း ဒီသဘောတရားပါပဲ။

```
class Ball(arcade.SpriteCircle):
    def update(self):
        self.center_y += self.change_y
        self.center_x += self.change_x

        if self.left < 0:
            self.change_x *= -1

        if self.right > WIN_WIDTH:
            self.change_x *= -1
```

```
# hitting bottom edge doesn't bounce
if self.bottom < 0:
    pass

if self.top > WIN_HEIGHT:
    self.change_y *= -1
```

ဒီကလပ်စ်က arcade.SpriteCircle ကို inherit လုပ်ထားတယ်။ SpriteCircle က အရောင်ဖြည့် စက်ပိုင်းပုံ အတွက်။ image ဖိုင် မလိုဘူး။

setup\_ball ကတော့ ဘေးလုံးနဲ့ ကန်းတည်နေရာနဲ့ တာထွက် velocity ကို အဓိက တွက်ချက်သတ်မှတ်ပေးတာပါ။ ဖန်တီးထားတဲ့ ဘေးလုံးကိုလည်း return ပြန်ပေးတယ်။

```
def setup_ball():
    ball = Ball(BALL_RADIUS, arcade.color.BLACK)
    ball.center_x = WIN_WIDTH // 2
    ball.center_y = WIN_HEIGHT // 2
    ball.change_y = -6
    random.uniform(1.0, 3.0)
    ball.change_x = random.uniform(1.0, 3.0) \
        if random.uniform(0.0, 1.0) <= 0.5 \
        else random.uniform(-1.0, -3.0)
    return ball
```

အစမှာ ဘေးလုံးအောက်ကို ကျေလာတဲ့အခါ ဦးတည်ရာက အောက်တည့်တည့်ကိုပဲ အမြတစ်သမှတ်တည်းဖြစ်နေရင် သိပ်မကောင်းဘူး။ ဤဦးငွေ့စွာဖြစ်နေမယ်။ ဘယ်ဘက်ကို ဦးတည် ဆင်းလာမှုလဲ ခန့်မှန်းလို့ရရင် ပိုမိုက်တယ်။ စိတ်လှပ်ရားဖို့ ပိုကောင်းတယ်။ ဒီအတွက် change\_x ကို 1.0 ကနေ 3.0 အတွင်းနဲ့ -1.0 ကနေ -3.0 အတွင်း random ထုတ်ထားတယ်။ random.uniform ဖန်ရှင် သုံးပါတယ်။ change\_x အနှစ်တန်ဖိုးဆိုရင် ဘယ်ဘက် အပေါင်းဆိုရင် ညာဘက်ကို ဦးတည်တယ်။ ဘယ်ဘက်လား ညာဘက်လားကိုလည်း ငါးဆယ် ငါးဆယ် ဖြစ်ချင်တယ်။ ဒါကြောင့်

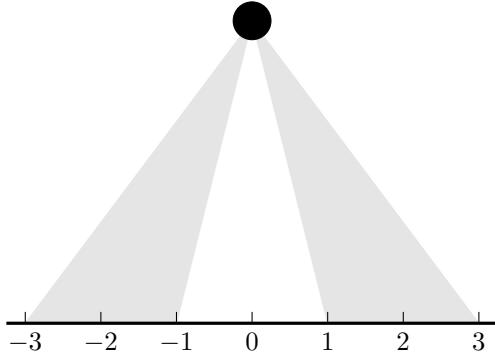
```
random.uniform(0.0, 1.0) <= 0.5
```

ဖြစ်ရင် အပေါင်းတန်ဖိုး ဖြစ်အောင် random.uniform(1.0, 3.0) နဲ့ ထုတ်ပေးတယ်။ မဟုတ်ရင်တော့ အနှစ်တန်ဖိုး ထွက်အောင်လုပ်ထားတယ်။ ဒါကြောင့် စစချင်းမှာ ဘေးလုံးကျေလာတဲ့ လားရာက ပုံ (၁၀.၅) မှာ ပြထားတဲ့ နယ်ပယ်အတွင်းမှာပဲ ရှိပါမယ်။

## Breakout Class

Breakout ကလပ်ကို ဆက်ကြည့်ရအောင်။ ဂိမ်းရဲ့ အဓိက View ဖြစ်ပြီး အရှုံးအနိုင်ဆုံးဖြတ်တာ၊ ဘေးလုံးနဲ့ အုတ်ခဲ့ ဘေးလုံးနဲ့ paddle တိုက်မိတာ စတာတွေကို ဒီကလပ်မှာ ရေးထားတာပါ။ paddle ကို မောက်စံနဲ့ ရွှေ့လို့ရအောင် on\_mouse\_motion ကလည်း ဒီထဲမှာပဲ။ [Breakout ကလပ်အပြည့်အစုံကို စာမျက်နှာ (၂၁၉) မှာ ကြည့်ပါ။]

`__init__` မက်သဒ်က သိပ်ရှုပ်ရှုပ်ထွေးထွေး မရှိဘူး။ မောက်စံပိုင်တာ ပေါ်နေအောင် self.window.set\_mouse\_visible(True) နဲ့ လုပ်ထားတယ်။ ဖျောက်ထားချင်ရင် False ထည့်ပေး။ Instance variable တွေ အားလုံးကို None ထည့်ထားတယ်။



ပုံ ၁၀.၅ ဘေးလုံးကျလာနိုင်တဲ့နေရာ

```
def __init__(self):
    super().__init__()

    arcade.set_background_color(WHITE)
    self.window.set_mouse_visible(True)
    self._ball = None
    self._paddle = None
    self._brick_lst = None
    self._brick_remains = None
    self._next_view = None
```

Initialization ကို setup မက်သဒ်မှာ အဓိက လုပ်ထားတယ်။ \_\_init\_\_ မှာ ဘာလို့ မလုပ်လဲ မေးစရာရှိတယ်။ \_\_init\_\_ မက်သဒ်က အော့ဘ်ဂျက် ဖန်တီးတဲ့အခါမှာပဲ အလိုအလျောက်ခေါ်ပေးတာ။ တိုက်ရှိက်ခေါ်ရတဲ့ မက်သဒ်မဟုတ်ဘူး။ ရုံးသွားလို (သို့) နိုင်သွားလို နောက်တစ်ခါ ပြန်ဆော့ချင်ရင် အစအနေအထား ဖြစ်အောင် setup မက်သဒ်ကို ခေါ်လို့ရမယ်။ (ဒီတော့ Breakout အော့ဘ်ဂျက် အသစ်တစ်ခု ဖန်တီးတာ၊ \_\_init\_\_ ကို တိုက်ရှိက်လည်း မခေါ်ဘဲ ပြန်စချင်ရင် setup ကို ခေါ်လို့ရမယ်။) ရှုမှာတွေ့ခဲ့တဲ့ setup\_ball, setup\_paddle, setup\_bricks တို့ကို တစ်ခာင့် ပြန်ခေါ်ထားတာ တွေ့ရမယ်။ \_brick\_remains က လက်ကျွန်းအုတ်ခဲ အရေအတွက် မှတ်ထားဖို့ အုတ်ခဲ တစ်လုံးပျက်သွားတိုင်း တစ်လျှော့ပေးရမယ်။

```
def setup(self):
    self._paddle = arcade.SpriteSolidColor(PDL_WIDTH,
                                            PDL_HEIGHT,
                                            BLACK)

    self._paddle.bottom = PDL_Y_OFFSET
    self._paddle.center_x = CENTER_X

    self._ball = setup_ball()
    self._brick_lst = setup_bricks()
    self._brick_remains = 20
```

draw မက်သဒ်က ဂိမ်းမှာပါတဲ့ အုတ်နံရုံး ဘေးလုံးနဲ့ paddle ပြားတို့ကို ဆွဲတယ်။ ဒီမက်သဒ်က Breakout ကလပ်စ်မှာ ထပ်ဖြည့်ထားတာ။ View ကလပ်စ်ရဲ့ တစ်စက်နဲ့ အကြိမ်ခြောက်ဆယ် ခေါ်နေမဲ့

`on_draw` ကို override လုပ်တာ မဟုတ်ဘူး။ `draw` ကို ခွဲထုတ်ထားရတာက လိုတဲ့အချိန်မှာ ကိုယ်တိုင် ခေါ်လို့ရအောင် ရော်ရွယ်တာပါ။

```

def on_draw(self):
    self.draw()

def draw(self):
    self.clear()
    self._paddle.draw()
    self._brick_lst.draw()
    self._ball.draw()

    ဂိမ်းရဲ့ အဓိကလေ့ဂျွစ်ကို on_update မက်သင်မှာ တွေ့ရမှာပါ။ View ကလပ်စဲ့ on_update
    ကို override လုပ်ပေးတာပါ။

def on_update(self, delta_time):
    self._ball.update()
    self._paddle.update()
    self._brick_lst.update()
    bricks_hit = arcade.check_for_collision_with_list(self._ball,
                                                    self._brick_lst)

    if len(bricks_hit) > 0:
        self._ball.change_y *= -1
    for brick in bricks_hit:
        brick.remove_from_sprite_lists()
        self._brick_remains -= 1

    if self._brick_remains == 0:
        self._next_view._msg = "You Win!"
        self.window.show_view(self._next_view)

    if self._ball.center_y <= PDL_Y_OFFSET:
        self._next_view._msg = "You Lost!"
        self.window.show_view(self._next_view)

    if (arcade.check_for_collision(self._ball, self._paddle)
        and self._ball.change_y < 0):
        self._ball.change_y *= -1

on_update ကို တစ်စက်နဲ့ အကြိမ်ခြေက်ဆယ် အဆက်မပြတ် အလိုအလျောက် ခေါ်ပေးတယ်လို့ ပြော
ခဲ့တာ ပြန်အမှတ်ရမယ် ထင်ပါတယ်။ ဂိမ်းရဲ့ state ကို ဒီမက်သင် တစ်ကြိမ်ခေါ်တိုင်း update လုပ်ပေး
ရပါမယ်။ ဒါဟာ ဂိမ်းတစ်ခုရဲ့ အချိန်နဲ့အမျှ ပြောင်းလဲနေတဲ့ အရာအားလုံးအတွက် အဓိကသေ့ချက်ပဲ။
ပထမဆုံး Breakout မှာပါတဲ့ ဘောလုံး paddle နဲ့ အုတ်ခဲအားလုံးရဲ့ လက်ရှိအခြေအနေကို update
လုပ်ရမယ်။ ဒီအတွက် ဂိမ်းမှာပါဝင်တဲ့ သက်ဆိုင်ရာ Sprite (သို့) SpriteList အားလုံးရဲ့ update
မက်သင်ကို ခေါ်ပေးရမှာပါ။

| self._ball.update()
```

၂၁၃

```
| self._paddle.update()  
| self._brick_lst.update()
```

check\_for\_collision\_with\_list က ဝင်တိုက်မိတဲ့ Sprite တွေကို စစ်ထုတ်ပေးတဲ့ မက်သင်။ ဘောလုံးနဲ့ တိုက်မိတဲ့ အုတ်ခဲတွေကို လိုချင်တာ။ ဒီတော့ အခုလို ခေါ်ရမယ်

```
| bricks_hit = arcade.check_for_collision_with_list(self._ball,  
|                                                 self._brick_lst)
```

အုတ်ခဲ (တော့) နဲ့ ဝင်တိုက်ရင် ဘောလုံးကို အထက် (သို့) အောက် ပြန်ကန်ထွက်အောင် change\_y ကို လက္ခဏာ ဆန့်ကျင်ဘက် ပြောင်းပေးပါတယ်။

```
| if len(bricks_hit) > 0:  
|     self._ball.change_y *= -1
```

ဝင်တိုက်တဲ့ အုတ်ခဲတွေကို ဖျက်ပစ်ရပါမယ်။ ဖျက်လိုက်ရင် လက်ကျွန်ုပ်အုတ်ခဲလည်း လျှော့သွားရမယ်။ ဒီအတွက် အခုလို

```
| for brick in bricks_hit:  
|     brick.remove_from_sprite_lists()  
|     self._brick_remain -= 1
```

တိုက်မိတဲ့ အုတ်ခဲတစ်ခဲချင်း ဖယ်ထုတ်ပါတယ်။

### Start, Main and End Views

ပထာမ စစ်ချင်းမှာ ပုံ (၁၀.၁) မှာ တွေ့ရတဲ့ အနေအထားအဓိုင်း ရှိနေမယ်။ ဒါပေမဲ့ စတာနဲ့ ဘောလုံးက ချက်ချင်းထွက်ရင် ဆော့ရတာ သိပ်အဆင်မပြောဘူး။ ကလစ်နှုပ်လိုက်မှ ဘောလုံးစထွက်မယ်ဆိုရင် ပို့ကောင်းမယ်။ StartView ကို အခုလိုသတ်မှတ်ထားတယ်

```
class StartView(arcade.View):  
    def __init__(self):  
        super().__init__()  
        self._next_view = None  
  
    def on_draw(self):  
        self.clear()  
        # Breakout ခဲ setup နဲ့ draw ကိုခေါ်တာ  
        self._next_view.setup()  
        self._next_view.draw()  
  
    def on_mouse_press(self, x, y, button, modifiers):  
        # ကလစ်နှုပ်ရင် Breakout View ကို ပြောင်းပေးတာ  
        if self._next_view:  
            self._next_view.setup()  
            self.window.show_view(self._next_view)
```

ဘောစောကရှင်းပြဲတဲ့ Breakout ဟာ ဂိမ်းရဲ့ အဓိက View ။ ဒါပေမဲ့ ဒီ View က စတာနဲ့ ဘောလုံး

က ရွှေ့မှာ on\_draw နဲ့ on\_update က ရပ်ထားလို့မရဘူး။ View ကို Window မှာ ပြပြီဆိုတာနဲ့  
တော်လျှောက် ခေါ်နေမှာ။ StartView က ရုပ်ပုံကို အပြုမှုပဲ ပြပေးရမယ်။ ဖြစ်နိုင်တဲ့ နည်းလမ်းတစ်ခုက Breakout ရဲ့ setup နဲ့ draw ကို StartView ရဲ့ on\_draw ကနေ ခေါ်လို့ရပါတယ်။ StartView  
နဲ့ Breakout ကို main မက်သင်ထဲမှာ အခုလို ချိတ်ပေးထားပါမယ်။

```
start_view = StartView()
game_view = Breakout()

# ...
start_view._next_view = game_view
window.show_view(start_view)
# ...
```

အခြားအနိုင် You Win!/You Lose! ပြန့် EndView ရဲ့ တာဝန်။ ရုံး (သို့) နိုင်ရင် မက်ဆွေချုပ်  
ပေးပြီး တစ်ခါတပ်ဆော့ချုပ်ရင် ကလစ်နိုပ်ရပါမယ်။

```
class EndView(arcade.View):
    def __init__(self):
        super().__init__()
        self._next_view = None
        self._msg = None

    def on_draw(self):
        self.clear()
        arcade.draw_text(self._msg,
                        WIN_WIDTH / 2,
                        WIN_HEIGHT / 2,
                        RED,
                        font_size=20,
                        anchor_x="center")

    def on_mouse_press(self, x, y, button, modifiers):
        if self._next_view:
            self.window.show_view(self._next_view)

main မက်သင်ထဲမှာ အခုလို ချိတ်ပေးထားပါတယ်
```

```
end_view = EndView()
# ...
end_view._next_view = start_view
```

### နိုင်း

ဒီအခန်းမှာ Breakout ဂိမ်းကို အသုံးချုပ်မှုမှာ အဖွဲ့အစည်း ထည့်ပေးထားတဲ့ ရည်ရွယ်ချက်က Arcade လိုက်  
ဘရီနဲ့ ဂိမ်းတွေ ဖန်တီးနိုင်ဖို့ အဓိက မဟုတ်ပါဘူး။ Inheritance ကို လိုက်ဘရီတွေမှာ အသုံးချေလေ့ရှိ  
တဲ့ ပုံစံတရာ့၍ကို နားလည်းသော်ပေါက်အောင်၊ သတိပြုမိအောင်၊ ဆက်စပ်မိအောင် အဓိက ရည်ရွယ်တာ  
ပါ။ စလေ့လာသူတွေအတွက် လွယ်လွယ်နဲ့ နားလည်နိုင်မယ်လို့တော့ မမျှော်လင့်နိုင်ဘူး။ စိတ်ရည်ရည်ထား

အချိန်ပေးပြီး နားလည်သဘောပေါက်အောင် လေ့လာဖို့ လိုပါလိမ့်မယ်။ ပရိုဂရမ်အပြည့်အစုံကို အောက်မှာ ဖော်ပြပေးထားပါတယ်။

တည်နေရာ အတွက်အချက်တွေ နားမလည်လိုလည်း သချ်ပြောက်သူတွေ စိတ်ခါတ်ကျစရာ မလိုပါဘူး။ အတွက်အချက်တွေ အကြမ်းဖျော်းလောက် နားလည်အောင် ကြည့်၊ ကျွန်ုတဲ့ ပရိုဂရမ်းမင်းနဲ့ဆိုင်တဲ့ သဘောတရားတွေ အမိကထား ကြည့်မယ်ဆိုရင်လည်း အတိုင်းအတာတစ်ခုထိ အကျိုးရှိမှုပါပဲ။

```

import random
import arcade
from arcade.color import *

WIN_WIDTH = 400
WIN_HEIGHT = 600

PDL_WIDTH = 60
PDL_HEIGHT = 10
PDL_Y_OFFSET = 30      # distance between paddle and lower edge of window

BALL_RADIUS = 10
BRICKS_PER_ROW = 10
BRICK_ROWS = 10          # number of brick rows
BRICK_GAP = 4            # gap between bricks
BRICK_WIDTH = ((WIN_WIDTH - (BRICKS_PER_ROW - 1) * BRICK_GAP)
                // BRICKS_PER_ROW)
LEFT_MARGIN = (WIN_WIDTH - (BRICK_WIDTH * BRICKS_PER_ROW +
                            BRICK_GAP * (BRICKS_PER_ROW - 1))) // 2
BRICK_HEIGHT = 8
BRICK_Y_OFFSET = 414
CENTER_X = WIN_WIDTH // 2
CENTER_Y = WIN_HEIGHT // 2

class Breakout(arcade.View):
    def __init__(self):
        super().__init__()

        arcade.set_background_color(WHITE)
        self.window.set_mouse_visible(True)
        self._ball = None
        self._paddle = None
        self._brick_lst = None
        self._brick_remains = None
        self._next_view = None

    def setup(self):
        self._paddle = arcade.SpriteSolidColor(PDL_WIDTH,

```

```
PDL_HEIGHT,
BLACK)

self._paddle.bottom = PDL_Y_OFFSET
self._paddle.center_x = CENTER_X

self._ball = setup_ball()
self._brick_lst = setup_bricks()
self._brick_remains = 20

def on_draw(self):
    self.draw()

def draw(self):
    self.clear()
    self._paddle.draw()
    self._brick_lst.draw()
    self._ball.draw()

def on_update(self, delta_time):
    self._ball.update()
    self._paddle.update()
    self._brick_lst.update()
    bricks_hit = arcade.check_for_collision_with_list(self._ball,
                                                       self._brick_lst)
    if len(bricks_hit) > 0:
        self._ball.change_y *= -1
    for brick in bricks_hit:
        brick.remove_from_sprite_lists()
        self._brick_remains -= 1

    if self._brick_remains == 0:
        self._next_view._msg = "You Win!"
        self.window.show_view(self._next_view)

    if self._ball.center_y <= PDL_Y_OFFSET:
        self._next_view._msg = "You Lost!"
        self.window.show_view(self._next_view)

    if (arcade.check_for_collision(self._ball, self._paddle)
        and self._ball.change_y < 0):
        self._ball.change_y *= -1

def on_mouse_motion(self, x, y, dx, dy):
    self._paddle.center_x = x
```

JJº

```
def setup_bricks():
    colors = [CYAN, CYAN, GREEN, GREEN, GOLD, GOLD,
              ORANGE, ORANGE, RED, RED]
    brick_lst = arcade.SpriteList()
    x = LEFT_MARGIN + BRICK_WIDTH // 2
    y = BRICK_Y_OFFSET + BRICK_HEIGHT // 2
    # 10 x 10 bricks wall
    for i in range(BRICKS_PER_ROW):
        for j in range(BRICK_ROWS):
            brick = arcade.SpriteSolidColor(BRICK_WIDTH,
                                              BRICK_HEIGHT,
                                              colors[i])
            brick.center_x = x + (j * (BRICK_WIDTH + BRICK_GAP))
            brick.center_y = y + (i * (BRICK_HEIGHT + BRICK_GAP))
            brick_lst.append(brick)
    return brick_lst

class Ball(arcade.SpriteCircle):
    def update(self):
        self.center_y += self.change_y
        self.center_x += self.change_x

        if self.left < 0:
            self.change_x *= -1

        if self.right > WIN_WIDTH:
            self.change_x *= -1

        # hitting bottom edge doesn't bounce
        if self.bottom < 0:
            pass

        if self.top > WIN_HEIGHT:
            self.change_y *= -1

def setup_ball():
    ball = Ball(BALL_RADIUS, arcade.color.BLACK)
    ball.center_x = WIN_WIDTH // 2
    ball.center_y = WIN_HEIGHT // 2
    ball.change_y = -6
    random.uniform(1.0, 3.0)
    ball.change_x = random.uniform(1.0, 3.0) \
        if random.uniform(0.0, 1.0) <= 0.5 \
        else random.uniform(-1.0, -3.0)
```

```
    return ball

class StartView(arcade.View):
    def __init__(self):
        super().__init__()
        self._next_view = None

    def on_draw(self):
        self.clear()
        self._next_view.setup()
        self._next_view.draw()

    def on_mouse_press(self, x, y, button, modifiers):
        if self._next_view:
            self._next_view.setup()
            self.window.show_view(self._next_view)

class EndView(arcade.View):
    def __init__(self):
        super().__init__()
        self._next_view = None
        self._msg = None

    def on_draw(self):
        self.clear()
        arcade.draw_text(self._msg,
                        WIN_WIDTH / 2,
                        WIN_HEIGHT / 2,
                        RED,
                        font_size=20,
                        anchor_x="center")

    def on_mouse_press(self, x, y, button, modifiers):
        if self._next_view:
            self.window.show_view(self._next_view)

def main():
    window = arcade.Window(WIN_WIDTH, WIN_HEIGHT, "Breakout")
    start_view = StartView()
    game_view = Breakout()
    end_view = EndView()

    start_view._next_view = game_view
```

JJR

```
game_view._next_view = end_view
end_view._next_view = start_view

window.show_view(start_view)
arcade.run()

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



# အခန်း ၁၁

## *Exceptions and Exception Handling*

တကယ့်လက်တွေ အသုံးချ ပရိုက်မ်တွေမှာ ရာနှစ်းပြည့် အမှားကင်းစင်ဖို့ဆိုတာ မဖြစ်နိုင်ပါဘူး။ ဒီလိုလုပ် ပေးနိုင်တဲ့ နည်းပညာလည်း ခုချိန်ထိ မရှိသေးဘူး။ ဒါကြောင့် ပရိုက်မ်တွေဟာ bug အနည်းနဲ့အများတော့ ပါကြတာပါပဲ။ ပရိုက်မ်မှ အမှားကြောင့် ဖြစ်တဲ့ ဟော တွေ လုံးဝမရှိအောင် တစ်နည်းတစ်လမ်းနဲ့ လုပ် နိုင်တယ် ဆုအုံတော့၊ တစ်ဖက်မှာ ပရိုက်မ်တစ်ခုကို အသုံးပြုနေစဉ် ကြိုတွေ့ရတဲ့ ချင်းချက် အခြေအနေ တွေက ရှိနေပါသေးတယ်။ အီးမေးလုပ်ပို့တဲ့ အချိန် နက်ဝံက ဒေါင်းနေတာ၊ ဖွင့်တဲ့ ဖိုင်က ပျက်နေတာ၊ သုည်း တဲ့တာ (division by zero)၊ မမ်းမို့ရှိမလုံလောက်တာ စတဲ့ ကိစ္စတွေ ပရိုက်မ် အလုပ်လုပ်နေ စဉ် ကြိုတွေ့ရတဲ့ ပါတယ်။ ဒီလို ပြဿနာတွေက အမြဲတမ်း ဖြစ်နေတာတော့ မဟုတ်ဘူး၊ ရုံဖန်ရံခါပဲ ဖြစ် တာဆိုပေမဲ့ ရှေ့ငှုံလွှဲလို့ (သို့) လုံးဝမဖြစ်အောင် ကာကွယ်လို့လည်း မရပြန်ဘူး။ ဒီလို အဖြစ်အပျက်တစ် ခု ဖြစ်လာခဲ့ရင် ပရိုက်မ်က လိုအပ်သလို စီမံထိန်းကွပ်လို့ရအောင် ရဲးရှင်းတဲ့ နည်းစနစ်တစ်မျိုး ရှိသင့်ပါ တယ်။ ပရိုက်မ်ရဲ့ ပုံမှန်စီးဆင်းမှု (ပြဿနာ မဖြစ်ခဲ့ရင် ပုံမှန်အတိုင်း လုပ်ဆောင်သွားမဲ့ ကုဒ်တွေကို ဆိုလို တာ) လမ်းကြောင်းကိုလည်း ဒီနည်းစနစ်ကြောင့် အများကြီးပါပြီး မရှုပ်ထွေးစေသင့်ဘူး။ တစ်နည်းအားဖြင့် ပြဿနာ မဖြစ်ရင် လုပ်ဆောင်မဲ့ အပိုင်းနဲ့ ဖြစ်ခဲ့ရင် လုပ်ဆောင်ရမဲ့ အပိုင်း ရောထွေးမနေသင့်ဘူး။ ခွဲဌား ထားရပါမယ်။

ဒီလို လိုအပ်ချက်တွေကို ဖြည့်ဆည်းပေးနိုင်တဲ့ နည်းလမ်းတွေတဲ့ အသုံးအများဆုံး တစ်ခုကတော့ exception-handling mechanism ပါပဲ။ ခေတ်ပေါ် programming language အားလုံးလိုလိုမှာ ထောက်ပံပေးထားပါတယ်။ တရှို့ language တွေမှာ အခြားနည်းလမ်းတွေ အသုံးပြုတာ တွေ့ရပေမဲ့ လက်တွေမှာ အခုပြောတဲ့ exception-handling လောက် မတွင်ကျယ်သေးဘူး။

*Exception-handling* သဘောတရားကို ဒီအခန်းမှာ အသေးစိတ် လေ့လာကြမှာပါ။ အောက်ပါ အတိုင်း အပိုင်းတွေခဲ့ လေ့လာကြမှာပါ။

- Raising exception
- Handling exception
- Control flow
- Built-in exception class hierarchy
- User-defined exceptions
- Handling multiple exceptions

## ၁၁.၁ Raising Exception

ဖန်ရှင်တစ်ခုဟာ ပြဿနာတစ်ခုခဲ့ကြောင့် သူတောင် ပြီးမြောက်အောင်မြင်အောင် ဆက်လက်လုပ်ဆောင် ဖို့ မဖြစ်နိုင်တဲ့အခါ exception တစ်ခုကို raise လုပ်နိုင်ပါတယ်။ (မြန်မာလိုဘေး exception တက် အောင် လုပ်တာလို့ ပြောလေ့ရှိတယ်)။ အောက်ပါ fun\_c ဟာ အကြိမ်တစ်ရာမှာ (၅၀) လောက် IOError exception တက်အောင် တမင်ရည်ရွယ် လုပ်ထားတယ်။ (ဥပမာပြီ့ အတွက်ပါ။ လက်တွေ့မှာ ဒီလိုလုပ် ဖို့ အကြောင်းမရှိပါဘူး)။ Random number ထုတ်ပြီး simulate လုပ်ထားတယ်။ အင်တာနှက်ကနေ ဒေတာတချို့၊ ဖတ်ပေးတဲ့ ဖန်ရှင်လို့ ယူဆချင် ယူဆပါ။ လိုင်း မကောင်းတဲ့ ဒေသမှာဆိုတော့ ဒီဖန်ရှင်က မကြာခဏ ပြဿနာပေးတယ်ပေါ့။

```
import random

def fun_c():
    print('Starting fun_c...')
    # simulate IOError, will fail 50% of the time
    if random.uniform(0.0, 1.0) <= 0.5:
        raise IOError("Failed to read!")
    print('fun_c ends!')
```

fun\_c ကို main ကနေ ခေါ်ပြီး အကြိမ်အနည်းငယ် run ကြည့်ပါ။

```
def main():
    fun_c()
    print('main ends')

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

အဆင်ပြတဲ့ အခါမှာ အခုလို

```
Starting main...
Starting fun_c...
fun_c ends!
main ends!
```

ထွက်တယ်။ Exception တက်ရင်တော့ ဒီလိုမျိုး error မက်ဆွဲချုပ်တွေ

```
Traceback (most recent call last):
  File ".../ch11/how_exceptions_works1.py", line 19, in <module>
    main()
  File ".../ch11/how_exceptions_works1.py", line 14, in main
    fun_c()
  File ".../ch11/how_exceptions_works1.py", line 8, in fun_c
    raise IOError("Failed to read!")
IOError: Failed to read!
Starting main...
Starting fun_c...
```

ကျေလာမှုပါ။ `main()` ကနေ `fun_c()` ခေါ်ပြီး အဲဒီမှာ `I0Error` ဖြစ်သွားတယ်လို့ ဖော်ပြထားတာ တွေ့ရတယ်။ (`most recent call last`) လိုလည်း တွေ့ရတယ်။ အောက်ဆုံးခေါ်ခဲ့တဲ့ ဖန်ရှင်လို့ ဆိုလိုတာ (`နောက်ဆုံး ခေါ်ခဲ့တာ fun_c`)။ ဖန်ရှင်တစ်ခုမှာ exception တက်တဲ့အခါ (သို့) ဖန်ရှင်တစ်ခုက exception ကို `raise` လိုက်တဲ့အခါ ငြင်းဖန်ရှင်ကို ခေါ်တဲ့ ဖန်ရှင်တွေအားလုံး ‘တောက်လျောက် fail ဖြစ်မယ်’။ အခုပ်မလုပ်မှာ `main` ကနေ `fun_c` ကို ခေါ်တယ်။ `fun_c` မှာ exception ဖြစ်တော့ `main` လည်း ပြီးအောင်ဆက် အလုပ်မလုပ်ပေးနိုင်ဘူး။ `fail` ဖြစ်သွားတယ်။

စောောက မက်ဆွဲချုပ်တွေကို သေချာဂရစိုက်ကြည့်ပါ။ `fun_c` မှာဆုံးရင် exception ဖြစ်စေတဲ့ နေရာ အောက်ပိုင်းက စတိတ်မန်တွေ၊ `main` မှာဆုံးရင် `fun_c` ကို ခေါ်ထားတဲ့နေရာ၊ အောက်က စတိတ်မန်တွေ အလုပ်မလုပ်သွားဘူး (`main` အတွက် `fun_c` ခေါ်တဲ့လိုင်းက exception ဖြစ်စေတဲ့နေရာ)။ အခုကိုစွဲမှာ ဖန်ရှင်နှစ်ခုလုံးရဲ့ exception ဖြစ်တဲ့နေရာ အောက်ပိုင်းမှာ `print` စတိတ်မန့် တစ်ကြောင်း စီပဲ ရှိပါတယ်။ အခြားစတိတ်မန်တွေ ရှိခဲ့ရင်လည်း အလုပ်လုပ်မှာ မဟုတ်ဘူး။

အကယ်၍ `main` က `fun_a` ကိုခေါ်၊ `fun_a` က တစ်ဆင့် `fun_b` ကိုခေါ်၊ `fun_b` ကနေမှ `fun_c` ကို နောက်ဆုံး ခေါ်ထားရင် `fun_c` မှာ exception ဖြစ်တဲ့အခါ `fun_b` က စပြီး `fail` ဖြစ်မယ်။ ပြီးရင်သူကိုခေါ်တဲ့ `fun_a` ဆက် `fail` မယ်။ နောက်ဆုံးမှာ `fun_a` ကိုခေါ်ထားတဲ့ `main` ဖန်ရှင် `fail` ဖြစ်ပြီး ပရှိကရမ်တစ်ခုလုံး ရပ်ဆိုင်းသွားမှာ ဖြစ်တယ်။ ရှေ့စာပိုဒ်မှာ ပြောခဲ့တဲ့ ‘တောက်လျောက် fail ဖြစ်မယ်’ ဆိုတာ အဲဒီလို့ဖြစ်စဉ်ကို ဆိုလိုတာ။

```
# fun_c exception ဖြစ်ရင် သူကို ခေါ်ထားတဲ့ ဖန်ရှင်အားလုံး fail ဖြစ်ပါမယ်
def main():
    print('Starting main...')
    fun_a()
    print('main ends!')

def fun_a():
    print('Starting fun_a...')
    fun_b()
    print('fun_a ends!')

def fun_b():
    print('Starting fun_b...')
    fun_c()
    print('fun_b ends!')

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

Exception raise လုပ်တာဟာ ဖန်ရှင်တစ်ခုက ငြင်းလုပ်ဆောင်ရမဲ့ တာဝန်ကို ပြသေနာ တစ်ခုကြောင့် ပြီးမြောက် အောင်မြင်အောင် မလုပ်ဆောင်နိုင်တော့ဘူးဆိုတာ ဖန်ရှင်ခေါ်ခဲ့သူကို အသိပေးတဲ့ နည်းလမ်းတစ်မျိုးလို့ ယူဆနိုင်ပါတယ်။ ဖန်ရှင်တစ်ခုကနေ အစပြု ဖြစ်ပေါ်တဲ့ exception ဟာ အဲဒီဖန်ရှင်ကို ခေါ်ထားတဲ့ ကွင်းဆက် (call chain) တစ်လျောက် ပါဝင်တဲ့ဖန်ရှင် တစ်ခုပြီးတစ်ခု exception ဖြစ်စေပြီး နောက်ဆုံးမှာ ပရှိကရမ်တစ်ခုလုံးကို ရပ်တန်းသွားစေမှာပါ။ Exception ဖြစ်ခဲ့ရင် ပရှိကရမ်တစ်ခုလုံးကို ပြန့်မသွားဘဲ မသက်ရောက်စေဘဲ ထိန်းကွပ်ပေးလို့ရတဲ့ နည်းလမ်းရှိရပါမယ်။ အဲဒီကတော့ exception ကို `handle` လုပ်ပေးခြင်းပါဘူး။

## 00. J Handling Exception

თეთრი უბრალა fun\_c კი დონეზე გვერდის გადასახვაზე გამოიყენება exception კი fun\_b ის გადასახვაზე და ამავე გადასახვაზე გვერდის გადასახვაზე გამოიყენება handle და ლუპნინგის გადასახვაზე გვერდის გადასახვაზე გამოიყენება.

```
def fun_b():
    print('Starting fun_b... ')
    try:
        fun_c()
        print("fun_c was successful!")
    except IOError as e:
        print(e)
        print("Poor connection!")
    print('fun_b ends!')
```

Python မှာ try...except က exception handling အတွက်ပါ။ Exception ဖြစ်နိုင်တဲ့ ဖန်ရင်ကို ခေါ်တဲ့အခါ try ဘလောက်ထဲမှာ ခေါ်ပါမယ် (ဖြစ်ခဲ့ရင် handle လုပ်မယ်ဆိုတဲ့ ပည့်ရွယ်ချက်ရှင်ပေါ့)။ except ဘလောက်က exception ဖြစ်ခဲ့ရင် handle လုပ်မဲ့ ကိုစွဲအတွက်။

```
except IOError as e:
```

IOError က handle လုပ်မဲ့ exception အမျိုးအစားကို သတ်မှတ်တာ။ ဆိုလိုတာက IOError သီးသန့်ကိုပဲ handle လုပ်မယ်။ IOError မဟုတ်တဲ့ အခြား exception တွေကို handle မလုပ်ဘူး။ (ဒါနဲ့ ပါတ်သက်ပြီး နောက်ပိုင်းမှာ ထပ်ရှင်းပြုဗာပါ)။ e က ဖြစ်ပေါ်တဲ့ IOError exception အတွက် ပေရံရောငဲ့လဲပါ။ Exception ဖြစ်ရင် fun\_c က raise လုပ်လိုက်တဲ့ IOError အော့က်ဂျက်ကို ဒီ ပေရံရောငဲ့လဲမှာ ထည့်ပေးမှာ ဖြစ်တယ်။ (Python မှာ IOError, ValueError, NameError စိတ်ကလပ်စွေတွေ ပါရှိပြီး ဖြစ်ပေါ်စဲ့ exception အမျိုးအစားအလိုက် သက်ဆိုင်ရာ exception အော့က်ရှုက်ကို raise လုပ်ရတာပါ)။

თეთოვის უბრალა fun\_b გვი აထინავთ ასეთი exception handling თანამდებობა: მაგრავი გვი აკავშირდეთ ასეთი exception ფუნქციის გვი აკავშირდეთ

```
Starting main...
Starting fun_a...
Starting fun_b...
Starting fun_c...
Failed to read!
Poor connection!
fun_b ends!
fun_a ends!
main ends!
```

တွေ့ရမှာပါ။ `fun_b` မှာ exception handling လုပ်ထားတဲ့အတွက် `fun_c` က exception ဖြစ်ခဲ့ရင် အဲဒီ exception ဟာ `fun_a` နဲ့ `main` ဆိုကို ထပ်ဆင့် မကူးစက်သွားတော့ဘူး။ Exception handling ဆိတ် ပရိုဂရမ် အခြားအစိတ်အပိုင်းတွေကို exception မကူးစက်သွားအောင် ကွာလန်တင်းလုပ် ထိန်းချုပ်တာလို့ ယူဆနိုင်ပါတယ်။

## Failure နဲ့ Exception ဘဏ္ဍားလဲ

အချုပ်းပြုးခဲ့သလောက်မှာ fail ဖြစ်တာ နဲ့ exception ဖြစ်တာ၊ ဒီသဘောတရားနှစ်ခု မရောထွေးသင့်ပါဘူး။ ဖန်ရှင်တစ်ခု fail ဖြစ်တယ်ဆိုတာ ပြဿနာတစ်ခုခေါ်ကြာင့် သူတာဝန်ကို အောင်မြင်အောင် မလုပ်နိုင်၊ အဲဒီ ပြဿနာကိုလည်း ကိုင်တွယ်ထိန်းကွပ်မထားတဲ့ အခြေအနေလို့ အကြမ်းဖျဉ်းယူဆပါ။ ဖန်ရှင်တစ်ခုက ပုံမှန်လမ်းကြောင်းအတိုင်း ပြီးမြောက်အောင် လုပ်ဆောင်သွားရင်၊ သို့မဟုတ် ပြဿနာ တစ်ခုခု ဖြစ်ခဲ့ရင်လည်း ဆက်မပြန့်သွားအောင် ကိုင်တွယ် ထိန်းကွပ်ပေးလိုက်ရင် successful ဖြစ်တယ်လို့ ယူဆရပါမယ်။ Exception က failure ဖြစ်စေ ‘နိုင်’ တဲ့ အကြောင်းအရင်း။ ဒါပေမဲ့ exception ဖြစ်ရင် fail ဖြစ်မယ် ပုံသေမှတ်လို့မရဘူး။ Exception ကို handle လုပ်လိုက်ရင် fail မဖြစ်တော့ဘူး။

`fun_c` exception ဖြစ်တော့ ခေါ်တဲ့ဖန်ရှင်အားလုံး တစ်ခုပြီးတစ်ခု ဆက်တိုက် fail ဖြစ်တယ်လို့ ရော်ပိုင်းမှာ ပြောခဲ့တယ်။ ဒါကို ပိုပြီးတိကျအောင် ပြောရမယ်ဆိုရင် `fun_c` exception ဖြစ်တဲ့အခါ သူကိုခေါ်တဲ့ `fun_b` ကိုလည်း exception ဖြစ်စေတယ်။ `fun_b` က အဲဒီ exception ကို handle လုပ်ထားရင် fail မဖြစ်ဘူး။ မလုပ်ထားရင်တော့ သူကိုယ်တိုင်လည်း fail ဖြစ်ပြီး သူကိုခေါ်တဲ့ `fun_a` ကို exception ဆက်ဖြစ်ပေါ်တယ်။ `fun_a` မှာလည်း ဒီသဘောအတိုင်း ဆက်ဖြစ်မှာပါ။ Exception handle လုပ်လိုက်ရင် fail မဖြစ်တော့ဘူး။ မလုပ်ထားရင်တော့ သူကိုခေါ်တဲ့ main ဖန်ရှင်ကို exception ဆက်ဖြစ်ပေါ်လိမ့်မယ်။

နောက်ထပ် သိထားဖို့ အရေးကြီးတာ တစ်ခုက exception ဖြစ်တဲ့အခါ try ဘလောက်ထဲမှာ ပါတဲ့ အောက်က စတိတ်မန်တွေကို ကျော်ပြီး except ဘလောက်ထဲ ချက်ချင်း ရောက်သွားမှာပါ။ try ဘလောက်ဟာ ပုံမှန် လုပ်ဆောင်မဲ့ လမ်းကြောင်း (normal execution flow) အတွက်ပါ။ Exception ဖြစ်ခဲ့ရင်တော့ ဒီလမ်းကြောင်းအတိုင်း ဆက်အလုပ်လုပ်လို့ မရတော့ဘူး။ ပုံမှန်မဟုတ်တဲ့ အခြေအနေမှာ လုပ်ဆောင်ရမဲ့ except ဘလောက်ကို လွှဲပြောင်း လုပ်ဆောင်ပေးရပါမယ်။ အခု ပြထားတာက ပုံမှန် အတိုင်း သွားမဲ့လမ်းကြောင်းပါ။

```
def fun_b():
    print('Starting fun_b... ')
    try:
        fun_c()
        print("fun_c was successful!")
    except IOError as e:
        print(e)
        print("Poor connection!")
    print('fun_b ends!')
```

အောက်မှာပြထားတာက `IOError` ဖြစ်ခဲ့ရင် လုပ်ဆောင်မဲ့ပဲ့ (`fun_c` ခေါ်ထားတဲ့လိုင်းကနေ except ကို ခုန်ပြီးရောက်သွားတာ သတိပြုပါ)။

```
def fun_b():
    print('Starting fun_b... ')
    try:
        fun_c()
        print("fun_c was successful!")
    except IOError as e:
        print(e)
        print("Poor connection!")
    print('fun_b ends!')
```

ရှော ဥပမာမှာ fun\_b handle မလုပ်ဘဲ fun\_a က handle လုပ်လိုလဲရတယ်။ ဒါမှမဟုတ် fun\_b နဲ့ fun\_a မှာ handle မလုပ်ဘဲ main က လုပ်နိုင်ပါတယ်။ အောက်ပါအတိုင်း fun\_a မှာ handle လုပ်မယ်ဆိုပါစိုး

```
def fun_a():
    print('Starting fun_a...')
    try:
        fun_b()
        print("fun_b was successful!")
    except IOError as e:
        print(e)
        print("Poor connection!")
    print('fun_a ends!')


def fun_b():
    print('Starting fun_b...')
    fun_c()
    print('fun_b ends!')
```

fun\_c exception တက်ရင် handle မလုပ်ထားတဲ့ fun\_b လည်း exception ဆက်ဖြစ်ပါမယ်။ အဲဒီ exception ကို fun\_a က handle လုပ်လိုက်တဲ့ အတွက် main ဆီကို ဆက်လက်မကူးစက် သွားတော့ပါဘူး။ Output အခုလို ထွက်တာ တွေ့ရမှာပါ။

```
Starting main...
Starting fun_a...
Starting fun_b...
Starting fun_c...
Failed to read!
Poor connection!
fun_a ends!
main ends!
```

Exception ဖြစ်တဲ့အခါ "fun\_b ends!" နဲ့ "fun\_b was successful!" အတွက် print စတိတ် မနဲ့တွေကို ကျော်သွားတာ သတိပြုကြည့်ပါ။

## ၁၁.၃ ဘယ်နေရာမှ handle လုပ်သင့်လဲ

ဖန်ရှင်တွေ တစ်ခုပြီးတစ်ခုဆင့် ခေါ်ထားတဲ့အခါ exception ဖြစ်ခဲ့ရင် ဘယ်ဖန်ရှင်က handle လုပ်သင့်လဲ စဉ်းစားဆုံးဖြတ်ဖို့ လိုလာတယ်။ ဒီကိစ္စက ဘယ်မှာ handle လုပ်ရမယ် ပုံသေပြေလိုတော့ မရဘူး။ အခြေအနေနဲ့ လိုအပ်ချက်ပေါ် မူတည် ဆုံးဖြတ်ရတာမျိုး။

```
def read_sensor():
    if random.uniform(0.0, 1.0) <= 0.2:
        raise IOError("Failed to read!")
    return random.randrange(1, 11)
```

ဒီဖန်ရှင်က sensor device တစ်ခုဆီကနေ ဒေတာဖတ်တာကို simulate လုပ်ထားတဲ့ ဖန်ရှင်ပါ။ Sensor ကြောင့်လို့သော်လည်းကောင်း၊ network ကြောင့်သော်လည်းကောင်း (၂၀) ရန်နှင့် fail ဖြစ်တယ်ဆိုပါတော့။

Sensor တန်ဖိုး သုံးခုတစ်တွဲ ဖတ်ပြီး စောင့်ကြည့်လေ့လာရမယ်လို့ စိတ်ကူးကြည့်ပါ။ တန်ဖိုးသုံး ခု အတွဲလိုက်ရအောင် ဖတ်ရမှုပါ။ Exception ဖြစ်လို့ သုံးခုမပြည့်သေးရင် ပြည့်တဲ့ထိ ထပ်ပြီးစားရပါမယ်။ ဖတ်တဲ့အခါ တစ်ခါနဲ့တစ်ခါ စက်နှင့်တစ်ဝက်ခြား ဖတ်ပါတယ်။ တစ်ကယ့် လက်တွေ့မှုလည်း sensor ကနေ ဒေတာဖတ်တဲ့အခါ တရရပ် ဖတ်လေ့မရှိဘူး။ အချိန်အနည်းငယ် ခြားပြီးဖတ်တယ်။

```
def read_3vals():
    vals = []
    while True:
        try:
            vals.append(read_sensor())
            if len(vals) == 3:
                return vals
        except IOError as err:
            pass
    time.sleep(0.5)
```

အခုကိစ္စအတွက် read\_3vals ဖန်ရှင်မှာ handle လုပ်ပေးရပါမယ်။ မလုပ်ဘဲထားရင် fail ဖြစ်ပြီး တန်ဖိုးသုံးခု ပြည့်အောင်ဖတ်လို့ မရှိနိုင်ဘူး။

အခုတစ်ခါ ကြားထဲမှာ ပြဿနာတစ်စုံတစ်ရာ မရှိဘဲ ဆက်တိုက် ဖတ်လို့ရတဲ့ တန်ဖိုးသုံးခု လိုချင်တယ် ယူဆပါ။ ဒီကိစ္စအတွက် exception handle မလုပ်ဘဲ ဖန်ရှင်တာစုံ အခုလို ရေးနိုင်တယ်။

```
def read_3_times():
    vals = []
    for i in range(3):
        vals.append(read_sensor())
        time.sleep(0.5)
    return vals
```

ဒီဖန်ရှင်က ပုံမှန်ဆုံးရှင်တော့ sensor ကို သုံးကြိမ်ဖတ်မှုပါ။ ဒါပေမဲ့ read\_sensor မှာ IOError exception ဖြစ်ခဲ့ရင် handle မလုပ်ထားတဲ့အတွက် အခါ read\_3\_times လည်း ဆက် fail ဖြစ်မယ်။ ကံကောင်းလို့ သုံးခါလို့ အဆင်ပြေားရင်တော့ ဖတ်ထားတဲ့ တန်ဖိုးသုံးခုပါတဲ့ vals ကို ပြန်ပေးမှုပါ။ စောစောကဖန်ရှင်နဲ့ အဓိက ကွာခြားချက်ကို သတိပြုပါ။ read\_3vals က တန်ဖိုး သုံးခုပြည့်တဲ့ထိ ဖတ်ပေးရမှုပါ။ ဒါကြောင့် exception handle လုပ်ဖို့ လိုက် လိုတယ်။ ခုဖန်ရှင်က တန်ဖိုးသုံးခုကို ကြားထဲမှာ fail မဖြစ်ဘဲ ဆက်တိုက် ဖတ်လို့ရမှုပဲ ပြန်ပေးရမယ်။ ဒါကြောင့် handle မလုပ်ဘဲ ထားလို့ရတယ်။ read\_3\_times ကို ခေါ်တဲ့သူက handle လုပ်/မလုပ် ဆက်လက်ဆုံးဖြတ်နိုင်ပါတယ်။

ခုဏာက ဖန်ရှင် အသုံးတည်လာမဲ့ အခြေအနေတစ်ခု စဉ်းစားကြည့်ရအောင်။ သုံးခုတစ်တွဲ (၁၀) ခါ ဖတ်ရင် ဘယ်နှစ်ခါ fail ဖြစ်လဲ ဆန်းစစ်ကြည့်ချင်တယ် စိတ်ကူးကြည့်ပါ။ read\_3\_times ကို အခြေပြီး failure\_rate ဖန်ရှင်ကို အခုလို သတ်မှတ်နိုင်ပါတယ်။

```
def failure_rate():
    fail = 0
    for i in range(10):
```

```

try:
    read_3_times()
except IOError as e:
    fail += 1
print(fail)
return Fraction(fail, 10)

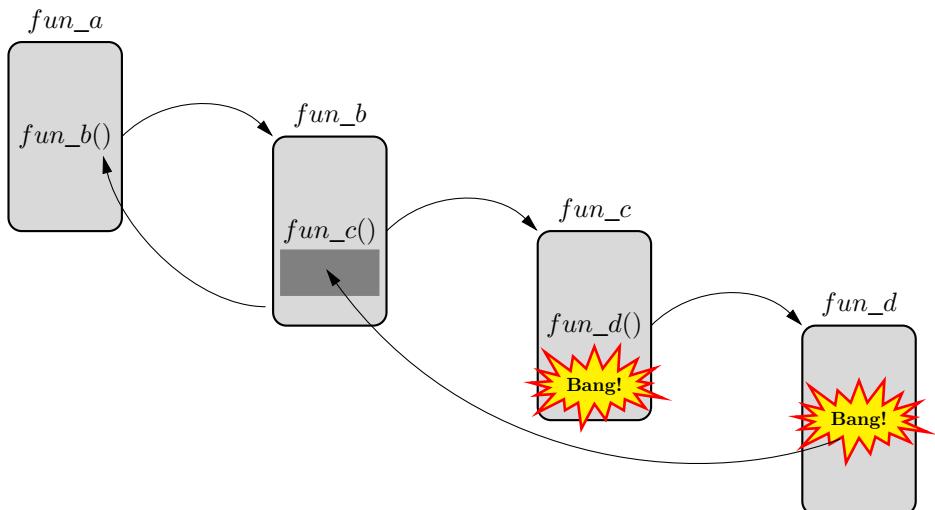
```

`read_3_times` က ဖြစ်ပေါ်တဲ့ exception ကို ဘယ်နှစ်ကြိမ် fail ဖြစ်လဲ ရေတွက်ဖို့ အသုံးချထားတော်းတွေရပါတယ်။

အခု လေ့လာတွေချိချက်တွေကို အနှစ်ချုပ်ပြန်ကြည့်ရင် ဖန်ရှင်တစ်ခု exception ဖြစ်တဲ့အခါ ငင်းဖန်ရှင်ကို ခေါ်ထားတဲ့ ကွင်းဆက်တစ်လျှောက်လုံးက ဖန်ရှင်တွေ fail ဖြစ်နိုင်ပါတယ်။ Fail မဖြစ်အောင် ဖန်ရှင်တစ်ခုက exception ကို handle လုပ်ရပါမယ်။ ဘယ်ဖန်ရှင်က handle လုပ်ရမလဲကတော့ ပုံသေမရှိဘူး။ လိုအပ်ချက်ပေါ် မူတည်ပြီး ဆုံးဖြတ်ရလေ့ရှိတယ်။

## ၁၁.၄ Exception Handling နဲ့ Control Flow

Exception handling ကို ပရိုဂရမ်တစ်ခု ပုံမှန်စီးဆင်းရာ လမ်းကြောင်းနဲ့ ပုံမှန်အခြေအနေ မဟုတ်တဲ့အခါ စီးဆင်းရာလမ်းကြောင်း ခွဲဗြားသတ်မှတ်ပေးတဲ့ နည်းစနစ်အဖြစ် ရှုပ်ဖိုင်ပါတယ်။ Exception ဖြစ်တဲ့အခါ စီးဆင်းပုံကို အကြမ်းဖျဉ်းအားဖြင့် ပုံ (၁၁.၁) မှာတွေ့ရသလို မြင်ကြည့်နိုင်ပါတယ်။



### ပုံ ၁၁.၀ Exception Handling

`fun_a`, `fun_b`, `fun_c`, `fun_d` တစ်ခုဗြီးတစ်ခု ဆင့်ပြီး ခေါ်ထားတယ်။ မီးခီးဖျော့ ထောင့်စွန်းပိုင်း ထောင့်မှန်စတုဂံတွေက ဖန်ရှင်တစ်ခုဗြီးကို ကိုယ်တွဲပြုတယ်။ အထဲမှာ ဖန်ရှင်ခေါ်ထားတာ တွေ့ရမယ်။ Handle လုပ်တာက `fun_b` မှာ (မီးခီးရင့် ထောင့်မှန်စတုဂံအသေးလေးက handle လုပ်တဲ့ အပိုင်းလို ယူဆပါ)။

`fun_d` မှာ exception ဖြစ်တယ်။ `fun_c` လည်း exception ဆက်ဖြစ်ပြီး fail ဖြစ်မယ်။ သူ့ကို ခေါ်တဲ့ `fun_b` ကိုလည်း exception ဆက်ဖြစ်ဖော်တယ်။ `fun_b` handle လုပ်လိုက်တဲ့ အတွက် fail မဖြစ်ဘူး။ handle လုပ်ပြီးသွားတော့ ငင်းကို ခေါ်ခဲ့တဲ့ `fun_a` တဲ့ကို ပြန်ရောက်သွားပြီး `fun_a`

ဆက်အလုပ်လုပ်မှာပါ။

ပုံအရ exception ဖြစ်တဲ့အခါ မူလစဖြစ်တဲ့နေရာကနေ handle လုပ်ထားတဲ့ အနီးဆုံး နေရာကို ခုန်ပြီးရောက်သွားတာကို တွေ့ရှုမှာပါ။ ဒီလို ခုန်သွားရှင်တာဟာ exception handling မှာ အဓိကကျတဲ့ လုပ်ဆောင်ချက်ဖြစ်တယ်။

### **else နဲ့ finally**

try...except အောက်မှာ else ဘလောက် နဲ့ finally ဘလောက် ရှုနိုင်ပါတယ်။ Exception မဖြစ်တဲ့အခါမှပဲ လုပ်ဆောင်ချင်တဲ့ စတိတ်မန်တွေ့ကို else ဘလောက်ထဲမှာ ထည့်နိုင်ပါတယ်။ တစ်နည်းအားဖြင့် try ဘလောက် ပြဿနာမရှိဘဲ အောင်မြင်ပြီးစီးမှသာလျှင် else ဘလောက်ကို လုပ်ဆောင်မှာပါ။ Exception ဖြစ်ခဲ့ရင်တော့ လုပ်ဆောင်ပေးမှာ မဟုတ်ပါဘူး။

```
def process_sensor_data():
    while True:
        try:
            val = read_sensor()
        except IOError as e:
            print(e)
        else:
            use_data(val)
            notify_if_necessary()
            print("Sensor data read successfully...")

        time.sleep(1)
        print("One iteration completed...")
```

```
Sensor data read successfully...
One iteration completed...
Sensor data read successfully...
One iteration completed...
Failed to read!
One iteration completed...
Failed to read!
```

ဒီ output ကို လေ့လာကြည့်ရင် else ဘလောက်ကို exception မတက်မှပဲ လုပ်ဆောင်ပေးတယ်ဆိုတာ မြင်နိုင်မှာပါ။ try...except...else အပြင် အောက်ဆုံးက နှစ်ကြောင်းနဲ့ သဘောတရား မတူတာကိုလည်း သတိပြု ကြည့်ပါ။

```
time.sleep(1)
print("One iteration completed...")
```

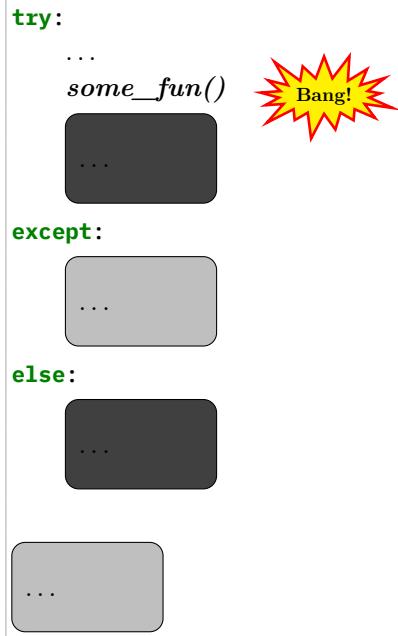
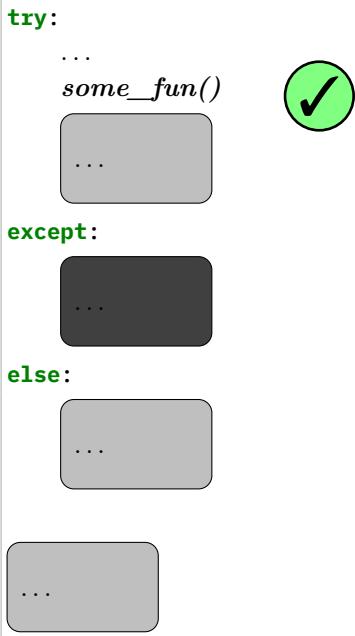
Exception ဖြစ်ဖြစ် မဖြစ်ဖြစ် ဒီ စတိတ်မန် နှစ်ခုကိုက လုပ်ဆောင်ပေးတယ်။ else အပိုင်းကိုတော့ exception မတက်မှပဲ လုပ်ဆောင်တယ်။

စော့စောက ဖုန်ရှင်ကို else ပေါ်ဘဲ အောက်ပါကဲ့သို့ ရေးမယ်ဆုံးရင်လည်း ရလဒ်အားဖြင့် တူတူပါပဲ။ else ကို အသုံးပြုရတဲ့ အဓိက အကြောင်းအရင်းက exception ဖြစ်စေနိုင်တဲ့ အပိုင်းနဲ့ မဖြစ်စေနိုင်

တဲ့ အပိုင်း သီးသန်ခဲ့ထားဖို့အတွက်ပါ။

```
def process_sensor_data():
    while True:
        try:
            val = read_sensor()
            use_data(val)
            notify_if_necessary()
            print("Sensor data read successfully...")
        except IOError as e:
            print(e)
        time.sleep(1)
        print("processed single sensor data successfully...")
```

အခုပုံစံမှာက `try` ထဲက ဖန်ရှင်တွေထဲက ဘယ်ဟာက exception ဖြစ်စေတဲ့ အရင်းအမြစ်လဲ အလွယ်  
တကူ မသိနိုင်တော့ဘူး။ စောင့်က `else` နဲ့ ပုံစံမှာက `read_sensor` က exception ဖြစ်နိုင်တဲ့ ဖန်  
ရှင်ဖြစ်ရမယ်၊ အဲဒီ exception ကို handle လုပ်ထားတယ်ဆိတာ သိသာ ဖြင့်သာပါတယ်။



Exception မဖြစ်တဲ့ အခါနဲ့ ဖြစ်တဲ့အခါ `try...except...else` အလုပ်လုပ်ပုံ နှင့်ယဉ်ပြထားတာ  
ပါ။ မီးခိုးရောင် အဖျော် ဘလောက်တွေက လုပ်ဆောင်မဲ့ ဘလောက်တွေပါ။ မီးခိုးရင့်ရောင် ဘလောက်  
တွေကိုတော့ လုပ်ဆောင်မှာ မဟုတ်ပါဘူး။

### `finally`

`except` အပြီးမှာ `finally` ဘလောက် ရှိနိုင်ပါတယ်။ Exception ဖြစ်သည်ဖြစ်စေ၊ မဖြစ်သည်ဖြစ်စေ  
`finally` အပိုင်းကို လုပ်ဆောင်ပေးမှာ ဖြစ်တယ်။ `return` လုပ်ရင်တောင်မှ `finally` လုပ်ဆောင်ပြီး  
မှပဲ လုပ်မှုပါ။

```
def test_read():
    try:
        val = read_sensor()
        print("Value: " + str(val))
        return
    except IOError as e:
        print("Error while reading!")
        return
    finally:
        print("Don't skip this!!!")
```

ဒီဖန်ရှင်ကို ထပ်ခါထပ်ခါ run ပြီး စမ်းကြည့်ပါ။ Exception မဖြစ်တဲ့ အခါ

```
Value: 5
Don't skip this!!!
```

ဖြစ်တဲ့အခါ

```
Error while reading!
Don't skip this!!!
```

ကို တွေ့ရမှုပါ။

ဒုန်ရှင် return မဖြစ်မီ finally ဘလောက်ကို လုပ်ဆောင်တာကို တွေ့ရတယ်။ return လုပ်ရင် ခေါ်ခဲ့တဲ့နေရာ ချက်ချင်းပြန်ရောက်တယ် ဆိုပေမဲ့ finally ပါရင်တော့ ချင်းချက်အနေနဲ့ မှတ်ရပါမယ်။ try...except ဘလောက်တွေဟာ တကယ့်လက်တွေမှာ အခုံပေးတွေလို ရီးရှင်းမှာ မဟုတ်ဘူး။ ဒီ ထက် အများကြီး ပုံပြီးရှုပ်ထွေး နိုင်ပါတယ်။ ကွန်ဒါရှင်နယ်လွှာ loop တွေ၊ break, early return စတဲ့ဟာတွေ ရောယ်နေတဲ့အခါ နောက်ဆုံးပိတ် လုပ်ဆောင်ပေးရမဲ့ final steps တချို့ ကျွန်းတဲ့တော့ ဖြစ် ဖို့ အလားအလာများတယ်။ ဥပမာ test\_read မှာ အခြေအနေပေါ်မှတည်ပြီး early return လုပ်မယ် ဆိုပါစူး။ finally မသုံးဘူးဆိုရင် return မထိုင်ခင် print တွေ အခုံလို ထည့်ပေးရပါမယ်။

```
def test_read():
    try:
        val = read_sensor()
        print("Value: " + str(val))
        if val < 3:
            # Do not forget here
            print("Don't skip this!!!")
            return
        elif val < 5:
            use(val)
        elif val < 8:
            # ...
            etc.,
            # Do not forget here
            print("Don't skip this!!!")
    except IOError as e:
        print("Error while reading!")
```

```
# Do not forget here
print("Don't skip this!!!")
return
```

တစ်နေရာရာမှာ ကျဉ်ခဲ့ပြီး များနိုင်ခြေ များတယ်။ ဒီအားနည်းချက်အပြင် ပိုကြီးတဲ့ ပြဿနာက ဘယ်ဟာက မဖြစ်မနေ လုပ်ဆောင်ရမဲ့ကိုစွဲ လဆိတ် ကြည့်ရနဲ့ ကွဲကွဲပြားပြား မမြင်နိုင်တော့ဘူး။ နေရာအတော် များများမှာ ဖြန့်ပြီးရှိနေတယ်။ finally သုံးခြင်းအားဖြင့် ဒီပြဿနာကို ဖြေရှင်းနိုင်ပါတယ်။

ဒေတာဘော့ ချိတ်ဆက်ခြင်း ဖိုင်ဖတ်ခြင်း/ရေးခြင်း နက်ဝပ်ချိတ်ဆက်ခြင်း စတဲ့ကိုစွဲတွေ လုပ်ဆောင် တဲ့အခါ အသုံးပြုထားတဲ့ ဖိုင် ကွန်နက်ရှင် စတဲ့ resource တွေကို release လုပ်ပေးဖို့ အရေးကြီးပါတယ်။ ဒီလို့ မလုပ်ရင် ကွန်ပျူးတာ စနစ်ရဲ့ CPU cycles, memory အစရိတ် အသုံးပြုထားတော့အတွက် ပြဿနာရှိပါတယ်။ အလုပ်ပြီးတဲ့အခါ လက်စသတ်တာ (clean up code)၊ resource release လုပ်တာ စတဲ့ကိုစွဲတွေအတွက် finally ကို သုံးလေ့ရှိတယ်။

**try:**

```
...
some_fun()
...
...
```



**except:**

```
...
...
```

**finally:**

```
...
...
```

```
...
...
```

**try:**

```
...
some_fun()
...
...
```



**except:**

```
...
...
```

**finally:**

```
...
...
```

```
...
...
```

Exception မဖြစ်တဲ့ အခါနဲ့ ဖြစ်တဲ့အခါ try...except...finally အလုပ်လုပ်ပုံ နှင့်ယူဉ်ပြထား တာပါ။ မီးခါးရောင် အဖျော် ဘလောက်တွေက လုပ်ဆောင်မဲ့ ဘလောက်တွေပါ။ မီးခါးရောင် ဘလောက် တွေကိုတွေ့ လုပ်ဆောင်မှာ မဟုတ်ပါဘူး။ စောောကဖော်ပြုခဲ့သလို early return ဖြစ်မယ်ဆိုလည်း finally အပိုင်း လုပ်ဆောင်ပြီးမှာပဲ return ဖြစ်မယ်။ finally မဟုတ်တဲ့ အခြား ဘလောကတွေ ကတော့ ငါးဘလောက မတိုင်မဲ့ early return ဖြစ်သွားရင်တွေ့ လုပ်ဆောင်မှာ မဟုတ်ပါဘူး။

## ၁၁.၅ Built-in and User-defined Exceptions

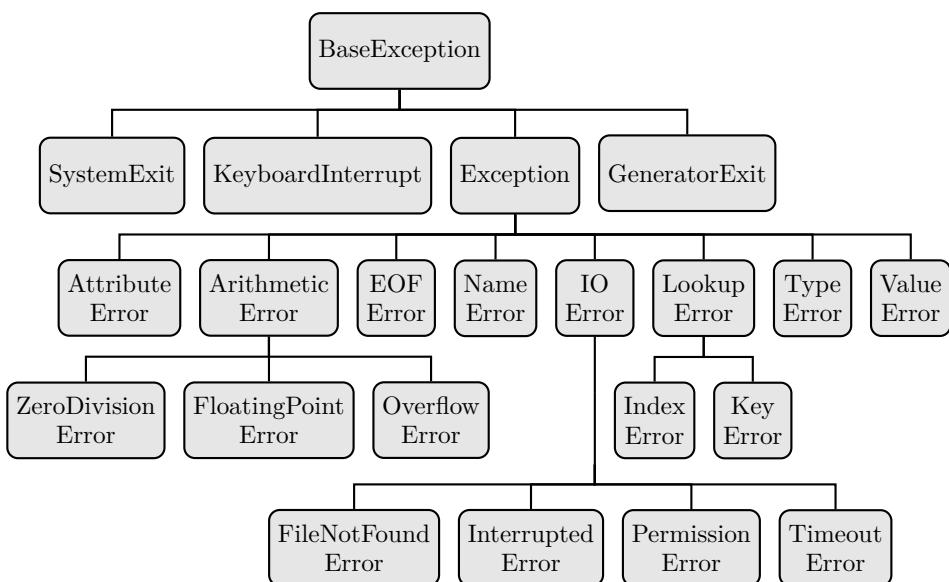
Exception ဆိုတာ ပရိုဂရမ်တစ်ခု ဖြစ်ရှိုးဖြစ်စဉ် စီးဆင်းရာ လမ်းကြောင်းကို အနောက်အယှက် အဟန် အတား ဖြစ်စေတဲ့ ‘ပုံမှန်မဟုတ်တဲ့ အဖြစ်အပျက်’ လို့ အကြမ်းဖျော်း ပြောနိုင်ပါတယ်။ ဒီလို့ အဖြစ်ပျက်မျိုး တွေက တစ်မျိုးတည်း မဟုတ်တာ သေချာပါတယ်။ ဖိုင်တစ်ခု ဖွင့်တဲ့အခါ အဲဒီဖိုင်က မရှိတာဖြစ်နိုင်သလို

ဖိုင်ကရီပေမဲ့ ခွင့်ပြုချက် (permission) မပေးထားလို့ ဖွင့်မရတယ်။ ကွန်ပျိုတာ မမှုပိုရီ မလုံလောက်တာ၊ နက်ဝပ်ချိတ်မရတာ၊ ဆာဟာပြသနာဖြစ်ပြီး ဒေါင်းနေတာ၊ array index ဘောင် ကျော်သွားတာ၊ သူညန့် စားမိတာ (division by zero)၊ ပုံမှန်မဟုတ်တဲ့ operating system signal စတဲ့ ပြသနာ တစ်ခုမဟုတ် တစ်ခု ပရိုဂရမ်အလုပ်လုပ်နေစဉ် ရုဖန်ရံခဲ့ကြော်တွေရတတ်ပါတယ်။

ဒီလို အကြောင်း အမျိုးမျိုးကြောင့် ဖြစ်ပေါ်နိုင်တဲ့ exception အမျိုးမျိုးအတွက် Python မှ သက်ဆိုင်ရာ built-in exception ကလပ်စ် အသီးသီး ပါရှိပါတယ် ပုံ (၁၁.၂)။ Exception အားလုံးဟာ BaseException ရဲ့ subclass တွေပါ။ Built-in exception တွေ အပြင် လိုအပ်ရင် ကိုယ်တိုင် သတ်မှတ်ချင်လည်း ရတယ်။ User-defined exception တွေက BaseException ကနေ တိုက်ရှိက် inherit မလုပ်ရဘူး။ Exception ကလပ်စ်ကို inherit လုပ်ရပါမယ်။ ဥပမာ

```
class InvalidAgeException(Exception):
    """Raised when age is not valid"""
    pass

class BalanceNotEnoughException(Exception):
    """Raised when account balance is not enough"""
    pass
```



ပုံ ၁၁.၂ Exception Class Hierarchy (အပြည့်အစုံ မဟုတ်ပါ၊ ချိန်ခဲ့တာတွေရှိသေးတယ်)

## ၁၁.၆ Handling Multiple Exception

ရှေ့ပိုင်း exception handling ဥပမာတွေမှာ exception တစ်မျိုးတည်းကိုပဲ handle လုပ်တာတွေရ မှုပါ။ except အပိုင်းမှာ သတ်မှတ်ထားတဲ့ exception ကလပ်စ် အမျိုးအစားကိုပဲ အဲဒီဘလောက်က handle လုပ်ပေးမှာ ဖြစ်တယ်။

```
try:
    ...

```

```
except IOError as e:
```

```
...
```

ဒီတိုင်းဆိုရင် IOError ကိုပဲ handle လုပ်မှာ ဖြစ်ပြီ။ ValueError (သို့) အခြား IOError instance မဟုတ်တဲ့ exception တွေကို handle မလုပ်ပါဘူး။ ValueError ကိုလည်း handle လုပ်မယ်ဆိုရင် အခုလို except ဘလောက်တစ်ခု ထပ်ဖြတ်နိုင်ပါတယ်။

```
try:
```

```
...
```

```
except IOError as e:
```

```
    # handle IOError here
```

```
except ValueError as e:
```

```
    # handle ValueError here
```

ဒီနေရာမှာ IOError နဲ့ ValueError တို့ဟာ is-a relationship မရှိတဲ့အတွက် except ဘလောက် နှစ်ခု အထက်အောက် ဖလှယ်လို့ရတယ်။ (IOError နဲ့ ValueError ကြေးမှာ superclass/subclass အပြန်အလှန် ဆက်စပ်မှု မရှိတာကို သတိပြုပါ)။

```
try:
```

```
...
```

```
except ValueError as e:
```

```
    # handle ValueError here
```

```
except IOError as e:
```

```
    # handle IOError here
```

တေဘာကနဲ့ အစီအစဉ်က ပြောင်းပြန်ဆိုပေမဲ့ ဖြစ်ပေါ်တဲ့ exception နဲ့ ကိုက်ညီတဲ့ except အပိုင်းက အလုပ်လုပ်မှုပါ။ ဘယ်ဟာ အရင်လာလာ အရေးမကြိုးဘူး။

အောက်ပါ fun\_c က FileNotFoundError (သို့) PermissionError တက်နှုင်ပါတယ်။ ဒီ exception နှစ်ခုလုံးက IOError ရဲ့ subclass ဖြစ်တာကိုလည်း သတိပြုပါ 〔ပုံ (၁၁.၂)〕။

```
def fun_c():
```

```
    print('Starting fun_c...')
```

```
    if random.uniform(0.0, 1.0) <= 0.25:
```

```
        raise FileNotFoundError("File not found!")
```

```
    if random.uniform(0.0, 1.0) > 0.75:
```

```
        raise PermissionError("No permission!")
```

```
    print('fun_c ends!')
```

ပုံစံတစ်မျိုးစိန့် သီးခြား handle လုပ်မယ်ဆိုရင် အခုလို

```
def fun_b():
```

```
    try:
```

```
        fun_c()
```

```
    except FileNotFoundError as e:
```

```
        print("Handle FileNotFoundError")
```

```
    except PermissionError as e:
```

```
        print("Handle PermissionError")
```

အကယ်၍ exception နှစ်ခုလုံးကို ပုံစံတစ်မျိုးတည်း handle လုပ်မယ်ဆိုရင်တော့ အခုလို

```
def fun_b1():
    try:
        fun_c()
    except (FileNotFoundException, PermissionError) as e:
        print("Handle both FileNotFoundError and PermissionError")
```

ရေးရပါမယ်။ ဂိုက်ကွင်းထဲမှာ handle လုပ်မဲ့ exception တွေကို ထည့်ပေးပါတယ်။ နောက်ထပ်နည်းလမ်းတစ်ခုက ဒီ exception နှစ်ခုရဲ့ superclass ဖြစ်တဲ့ IOError ကို handle လုပ်လိုလည်း ရပါတယ်။

```
def fun_b1():
    try:
        fun_c()
    except IOError as e:
        print("Handle both FileNotFoundError and PermissionError")
```

ဒီလိုဆိုရင်တော့ IOError အပါအဝင် သူ့ subclass exception အားလုံး handle လုပ်မှာဖြစ်တယ်။ ပုံ (၁၁.၂) class hierarchy အရ InterruptedError နဲ့ TimeoutError တို့ကိုပါ handle လုပ်မှုပါ။ ဖော်ပြခဲ့တဲ့ handling နည်းတွေကို ပေါင်းစပ်ပြီး လိုအပ်သလို အသုံးပြုနိုင်ပါတယ်။ ဆက်စပ်မှုရှိတဲ့ exception တွေကို တစ်ပေါင်းတည်း handle လုပ်တာ၊ အမျိုးအစား တစ်ခုချင်းအလိုက် သီး၌ဗျား handle လုပ်တာ၊ ဒါမူမဟုတ် တရာ့ဘို့တော့ ရွေးထုတ်ပြီး ကျွန်တာတွေကို ယေဘုယျပုံစံတစ်မျိုးနဲ့ handle လုပ်တာ စသည့်ဖြင့် လိုချင်တဲ့အတိုင်း ရအောင် အသေးစိတ် ချိန်ညိုပေးလို့ ရပါတယ်။ စမ်းကြည့်ရုံးသက်သက် တမင်ဖန်တီးထားတဲ့ အောက်ပါ ဥပမာဏကို လေ့လာကြည့်ပါ။

```
try:
    fun_d()
except InterruptedError as e:
    print(e)
except TimeoutError as e:
    print(e)
except (FileNotFoundException, PermissionError) as e:
    print(e)
except IOError as e:
    print(e)
except ArithmeticError as e:
    print(e)
```

InterruptedError နဲ့ TimeoutError ကို သီး၌ဗျား handle လုပ်ထားတယ်။ FileNotFoundException နဲ့ PermissionError က ပေါင်းထားတယ်။ ဒီ လေးခု မဟုတ်တဲ့ ကျွန်တဲ့ အေား IOError အားလုံးအတွက် အောက်ဆုံး မတိုင်ခေါ် except က တာဝန်ယူမယ်။ နောက်ဆုံးမှာ ZeroDivisionError, OverflowError စတဲ့ ArithmeticError အားလုံးကို ခြော်ပြီး handle လုပ်တယ်။ fun\_d ကို အောက်မှာ ကြည့်ပါ။ စမ်းကြည့်ချင်တဲ့ exception တက်အောင် သက်ဆိုင်ရာ ကေန်းရိုက်ထည့်ရှုပဲ။

```
def fun_d():
    val = int(input("Enter an int: "))
```

```

if val == 1:
    raise TimeoutError("Timeout")
if val == 2:
    raise InterruptedError("Interrupted")
if val == 3:
    raise FileNotFoundException("File not found")
if val == 4:
    raise PermissionError("Not permitted")
if val == 5:
    raise FileExistsError("File already exists")
if val == 6:
    raise FloatingPointError("Floating point error")
if val == 7:
    x = 10/0           # will cause ZeroDivisionError
if val == 8:
    x = 2.0 ** 5000  # will cause OverflowError

```

### **except** ဘလောက် အခီအခုံ

Subclass တွေတဲ့ ဘလောက် အခီအခုံ handle လုပ်ပြီး ကျန်တာတွေကို superclass instance အနေ နဲ့ ခြေပြီး handle လုပ်တဲ့အခါ except ဘလောက်တွေ အထက်အောက် စီစဉ်ထားတာကို သတိထားရ ပါမယ်။ Subclass ကို အရင် handle လုပ်ပြီး superclass ကို subclass အောက်မှာ လုပ်ရပါမယ်။ အခုံလို့ handle နည်းလမ်းမမှန်ပါဘူး။

```

try:
    fun_c()
    fun_c1()
except IOError as e:
    print("Handle IOError")
except (FileNotFoundException) as e:
    print("Handle FileNotFoundException")

```

FileNotFoundException ၃ IOERROR ရဲ့ subclass ။ ဒါကြောင့် FileNotFoundException တက်ခဲ့ရင် အပေါ် IOError အတောက် except ဘလောက်က အရင် handle လုပ်ပါလိမ့်မယ်။ သူကို သို့ဟေး handle လုပ်ဖို့ ရည်ရွယ်တဲ့ အောက်က except ကို ဘယ်တော့မှ လုပ်အောင်မှာ မဟုတ်တော့ဘူး။ ဒီလို့ မှပဲ လိုချင်တဲ့အတိုင်း အမှန်ဖြစ်မှပါ။

```

try:
    fun_d()
except (FileNotFoundException) as e:
    print("Handle FileNotFoundException")
print("Handle IOError and all its subclasses "
      "except FileNotFoundException")

```

```

try:
    fun_d()

```

```

except Exception as e:
    print("Handle all Exception")
except (FileNotFoundException) as e:
    print("Handle FileNotFoundError")
except IOError as e:
    print("Handle all IOError")

```

ဒါလည်း သိပ်တော့ မဟုတ်သေးဘူး။ Exception ၏ IOError ခဲ့ superclass ဆိုတော့ အောက်ဆုံး မှာ ထားသင့်တယ်။

### နိဂုံး

ဒီအခန်းမှာ exception handling နဲ့ ပါတ်သက်ပြီး အခြေခံအဆင့် သိသင့်သိတိက်တဲ့ သဘောတရား တွေကို ထည့်သွင်းဖော်ပြပေးထားပါတယ်။ အတွေ့အကြံ သိပ်မရှိသေးတဲ့ သူတွေအတွက် အပြည့်အဝ နားလည်ဖို့ အခက်အခဲ ရှိနိုင်ပါတယ်။ သဘောတရား နားလည်ရှုနဲ့လည်း မရသေးဘူး။ လက်တွေ့ အခြေ အနေတွေမှာ အသုံးချက်တိဖို့လည်း အရေးကြီးတယ်။ လုပ်သက်အတွေ့အကြံ ရင့်လာတာနဲ့အမျှ ပိုပြီးနား လည်လာမှုပါ။ အခြေအနေ၊ အချိန်အခါး လိုအပ်ချက်ပေါ့ မူတည်ပြီးတော့ လည်း သင့်တော်တဲ့ နည်းလမ်း ရွေးချယ် အသုံးချက်တိလာမှုပါ။ ပြီးစလွယ် ဖြစ်ကဗာတ်ဆန်း မလုပ်ဘဲ တတ်နိုင်သမျှ အကောင်းဆုံးဖြစ် အောင် စဉ်ဆက်မပြတ် ဆင်ခြင်စဉ်းစား သုံးသပ်ဖို့ ပိုကောင်းသည်ထက် ကောင်းမဲ့ နည်းလမ်းကို အမြှေဖွေနှုန်းဖို့တော့ လိုပါလိမ့်မယ်။



# ଓପଟିକ୍ ଓ ଜୀ

ଓଡ଼ିଆରେ ମାତ୍ରରେ ପରିଚୟ ଦେଇଛନ୍ତି

〔Not completed yet!〕



# ଓପାର୍ଟିକ୍ସ୍ ଓଡ଼ିଆ

## *File Input and Output*

〔Not completed yet!〕



# ଓପଟିକ୍ ଓଡ଼ିଆ

*Graphical User Interface (GUI) Programming*

〔Not completed yet!〕



# အခန်း ၁၅

*Basic Concurrency*

〔Not completed yet!〕



# ଓପନ୍ଦିଃ କଟ

*Miscellaneous*

〔Not completed yet!〕

JJJ

# နောက်ဆက်တဲ့ က

## လိုအပ်သည့် ဆောဖံ့ဌများ ထည့်သွင်းခြင်း

အခြား အင်ဂျင်နီယာ/သိပ္ပံာ ပညာရပ်တွေလိုပဲ ပရိုကရမ်မင်းလေ့လာတဲ့အခါ လက်တွေ့လုပ်ကြည့်ဖို့ အင်မတန်အရေးကြိုးပါတယ်။ လက်တွေ့ရေးမကြည့်ဘဲ စမ်းသပ်မကြည့်ဘဲ သဘောတရားပိုင်းဆိုင်ရာတွေကို အမှန်တကယ်နားလည်ဗူး မဟုတ်ပါဘူး။ အခြားပညာရပ်တွေထက် ကွန်ပျိုးတာ ပရိုကရမ်မင်းရဲ့ အားသာချက်တစ်ခုကတော့ လက်တွေ့စမ်းသပ်ခန်းကြိုးတွေ ရှိစရာမလိုတာပါပဲ။ စမ်းသပ်ပစ္စည်းတွေ လည်း များများစားစား မလိုအပ်ဘူး။ ကွန်ပျိုးတာ တစ်လုံးနဲ့ လိုအပ်တဲ့ ဆောဖံ့ဌပဲတဲ့ ရှိရင်ရပြီ။ ဆောဖံ့ဌတွေကလည်း ပိုက်ဆံကုန်စရာမလိုဘူး။ မပေးဘဲ သုံးလို့ရတာ။

ပရိုကရမ်လက်တွေ့ရေး လေ့ကျင့်ဖို့အတွက် လိုအပ်တဲ့ ဆောဖံ့ဌတွေ ထည့်ထားရပါမယ်။ Python programming language အတွက် Python ဆောဖံ့ဌ ထည့်ရပါမယ်။ Python ဆောဖံ့ဌမရှိဘဲ Python ကုဒ်တွေ၊ Python ပရိုကရမ်တွေ run လို မရပါဘူး။ Python အပြင် ပရိုကရမ်ကုဒ် ရေးဖို့ အတွက် အထောက်အကူဗြာ ဆောဖံ့ဌပဲတစ်ခုလည်း လိုအပ်တယ်။

ပရိုကရမ် ကုဒ်ရေးဖို့အတွက် PyCharm သိမဟုတ် Visual Studio Code (VS Code) ကို အသုံးပြုနိုင်ပါတယ်။ အက်ဆေးတစ်ပုဒ်ရေးတဲ့အခါ ပိုက်ခရိုဆောဖံ့ဌ Word နဲ့ရေးလိုရသလို ရိုးရိုးရှင်းရှင်း Notepad လောက်နဲ့ ရေးလိုလည်း ရတာပါပဲ။ အက်ဆေးရဲ့ အဓိပ္ပာဇဵ်ကသာ အစိုကပါ။ ပရိုကရမ်ကုဒ် ရေးတဲ့အခါမှာလည်း ခီးသောပါပဲ။ ဆောဖံ့ဌပဲရေးရေးကြိုးတွေ တည်ဆောက်တဲ့အခါ လိုအပ်မဲ့ ဖီချာတွေ အားလုံး စုံစုံလင်လင်ပါပြီးသား PyCharm လို Integrated Development Environment(IDE) ဆောဖံ့ဌပဲမျိုးနဲ့ ကုဒ်တွေရေးလိုရသလို ပါပေါ့ပါးပါးနဲ့ လိုအပ်မှုပဲ လိုတဲ့ဖီချာအတွက် extensions (plugin လိုလည်းခေါ်တယ်) ထည့်သွင်းရတဲ့ Visual Studio Code (VS Code) လို ကုဒ်အသိဒ်ဘာ (Code Editor) ဆောဖံ့ဌပဲမျိုး သုံးပြီး ရေးရင်လည်း ရတာပါပဲ။ နောက်ဆုံး ကုဒ်နည်နည်း၊ ဖိုင်နည်းနည်း ရိုးရှင်းတဲ့ ပရိုကရမ်လေးတွေဆိုရင် Notepad နဲ့ ရေးလိုတောင် ရပါတယ်။ ကုဒ်တွေများမယ် ဖိုင်တွေ များမယ်၊ အသင်းအစွဲလိုက် ပူးပေါင်းရေးရတဲ့ ပရိုကရမ်မျိုးတွေ ဆိုရင်တော့လည်း Notepad လောက်နဲ့ အဆင်မပြနိုင်တော့ဘူးပေါ့။

PyCharm ရေး VS Code ထည့်သွင်းနည်းပါ ဖော်ပြပေးပါမယ်။ မိမိ နှစ်သက်ရာ အဆင်ပြောရာ သို့မဟုတ် နီးစပ်ရာ ပရိုကရမ်ဘာ အသိမိတ်ဆွေ အကြုပြုတဲ့ တစ်ခုကို ရွေးချယ်သုံးပါ။ စလေ့လာသူအနေနဲ့ PyCharm ကို အသုံးပြုတာ ပိုအဆင်ပြောမယ်လို ထင်တယ်။ PyCharm သုံးကြည့်လို မိမိကွန်ပျိုးတာ မှာ နေးလွန်းတယ်ဆိုရင် VS Code ကိုစမ်းကြည့်ပါ။ နှစ်ခုလုံး စက်အရမ်းကောင်း/မြင့် ဖို့ မလိုပါဘူး။ တော်ရုံး အတန်အသင့်ကောင်းတဲ့ စက်လောက်နဲ့ အဆင်ပြောပါတယ်။

မိမိကိုယ်တိုင်က ကွန်ပျိုးတာ အသုံးပြုပဲ အခြေခံ အားနည်းပြီး ဖော်ပြပေးထားတဲ့ အတိုင်း တစ်ဆင့် ချင်း အင်စတောလ် လုပ်တာလည်း အဆင်မပြောဖြစ်နေရင် ဒီဘာအုပ်ရဲ့ အောက်ပါ ဖွံ့ဖြိုးတွေနဲ့ ယူကျ။

ချုပ်နယ်တွေမှာ ကြည့်ရှုမေးမြန်း အကူအညီ တောင်းဆိုင်ပါတယ်။ ဒါမှမဟုတ် အတွေးအကြိုးစိတ် နဲ့ အကူအညီယူပြီး အင်စတောလ်လုပ်ပါ။

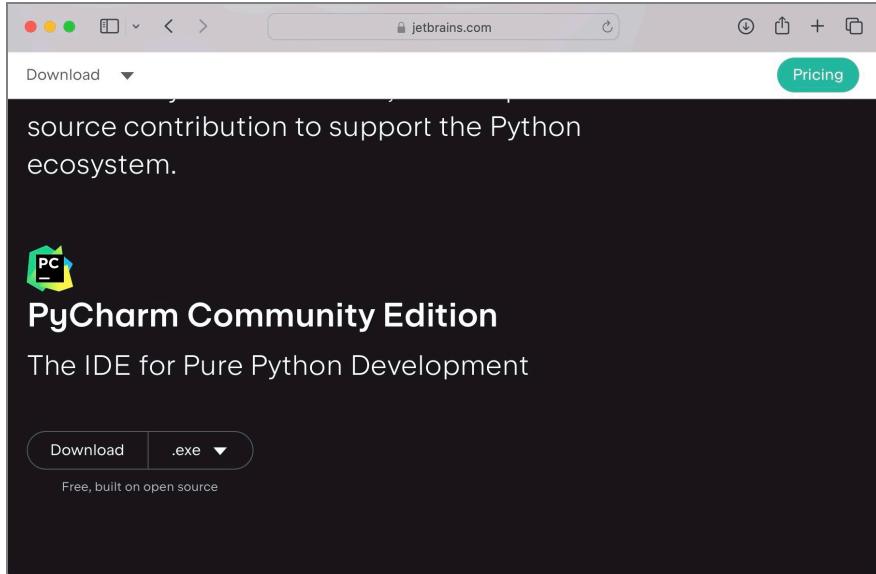
<https://www.facebook.com/bpwp>

<https://www.youtube.com/bpwp>

PyCharm အတွက် အရင်ဖော်ပြပေးပါမယ်။ VS Code အတွက် စာမျက်နှာ ၂၆၉ မှာ ကြည့်ပါ။

## Python နှင့် PyCharm IDE ထည့်သွင်းခြင်း

<https://www.jetbrains.com/pycharm/download/> လင့်ကိုဖွံ့ဖြိုးပါ။ ဝဘ်စာမျက်နှာ အောက်ဘက် နည်းနည်း ဆဲချလိုက်ရင် PyCharm Community Edition ဒေါင်းလုအုပ်ခလုတ်ကို တွေ့ရပါမယ်။ ပုံ (၁/၁) ကိုကြည့်ပါ။

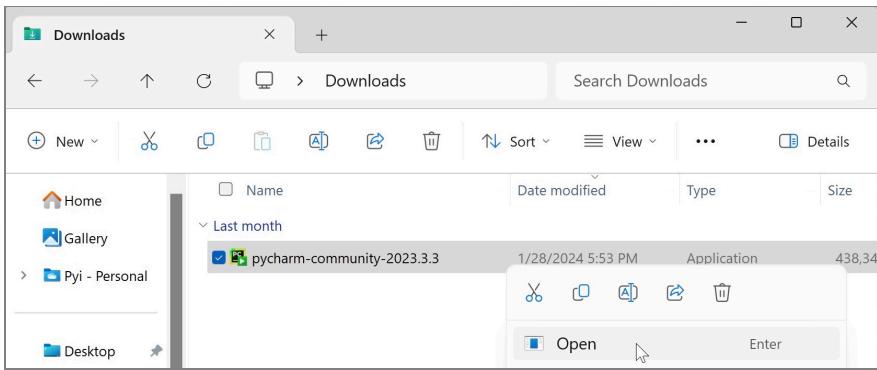


ပုံ ၁/၁

PyCharm Community Edition ကို ဒေါင်းလုအုပ်ပါ။ (ဝဘ်စာမျက်နှာ အပေါ်ပိုင်းက ၀၍ သုံးရတဲ့ PyCharm Professional ကို ဒေါင်းလုအုပ်မှုပါ။) အင်စတောလုဖိုင်ကို ပြောလစ်နှုပ်ပြီး Open လုပ်ပါ (ပုံ ၁/၂ ကိုကြည့်ပါ။) Yes/No မေးတဲ့အခါ Yes နှိပ်ပါ။ Next > ကို နှိပ်၍ ရွှေ့ကိုဆက်သွားပြီး နောက်ဆုံးမှာ Install နှိပ်ပြီး ကွန်ဖန်းလုပ်ပါ။ အင်စတောလ်ပြီးသွားရင် Finish နှိပ်ပါ။

ဝင်းခိုး Taskbar Search ကနေ PyCharm ကိုရှာပြီးဖွင့်ပါ (ပုံ ၁/၃ ကို ကြည့်ပါ။) သဘောတူ ကြောင်း ကွန်းဖန်းလုပ်ခိုင်းရင် ချက်ချေဘောက်စ် ချက်ချေလုပ်ပြီး Continue နှိပ်ပါ။ ဒေတာပို့ချင်လား ထပ် မေးပါလိမ့်မယ်။ Don't Send နှိပ်ပါ။ Welcome စာရင်ကိုပေါ်ဘာမယ်။ ပုံ (၁/၄) မှာ ကြည့်ပါ။

## ၂၁၁

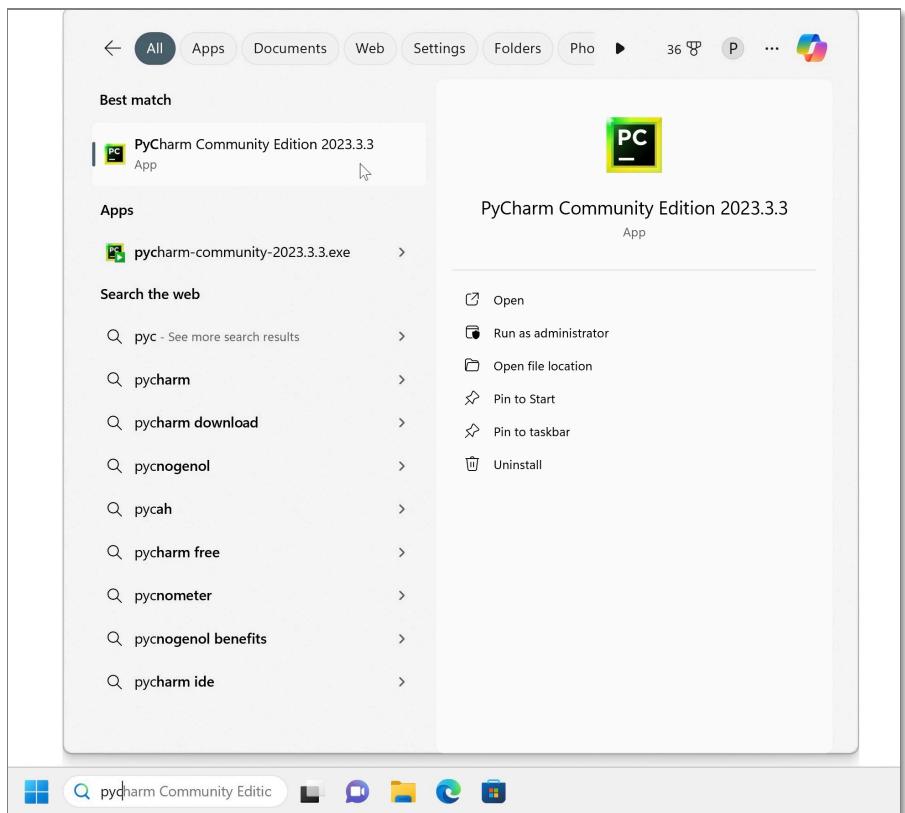


ပုံ ၃/၂

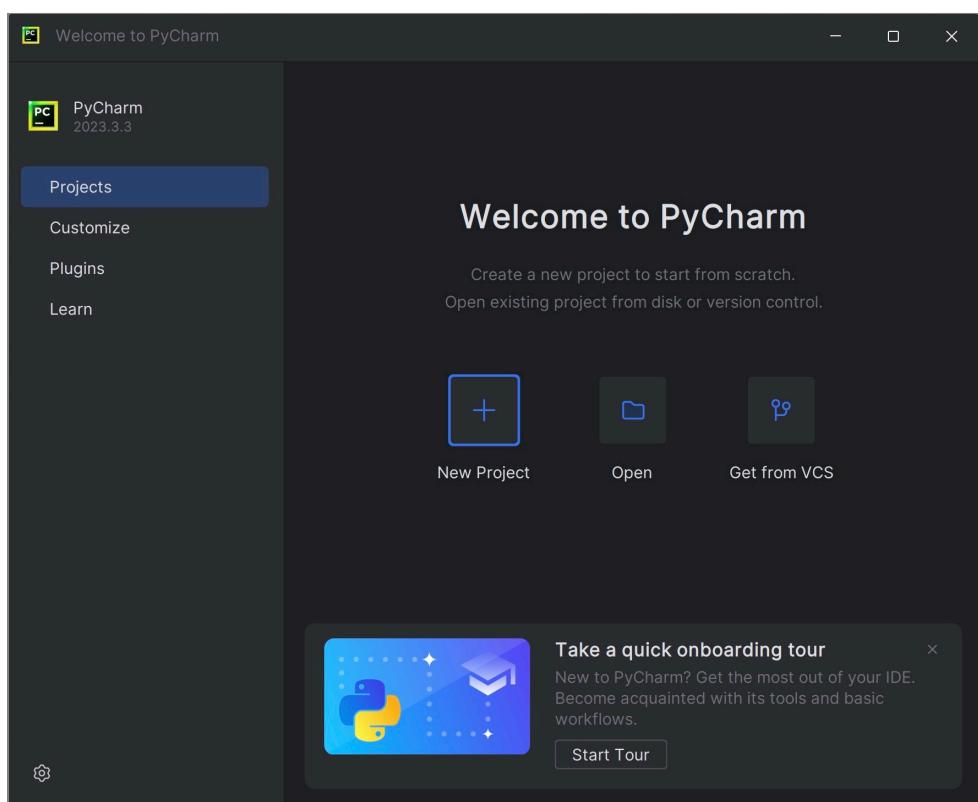
ဒီတရေးနေချိန် လက်ရှိ PyCharm ဘာရှင်းက ၂၀၂၃ ပါ။ သိပ်မကြာခင် ၂၀၂၄ ထွက်ပါတော့မယ်။ အကယ်၍ လက်ရှိဟားရှင်းထက် နိမ့်တဲ့ဟားရှင်းတွေကို ဒေါင်းလုပ် လုပ်ချင်ရင် အောက်ပါ လင့်ကို သွားပါ။

<https://www.jetbrains.com/pycharm/download/other.html>

ဘာရှင်း ၂၀၂၄/၂၅ ထွက်ပြီးတဲ့ အချိန်ဟာ ၂၀၂၃ ဘာရှင်းကို လိုချင်ရင် ဝတ်စာမျက်နှာမှ ရှာပြီး ဒေါင်းလုပ် လုပ်ပါ။ ၂၀၂၃ မှာလည်း ဘာရှင်းအခွဲတွေ ရှိပါသေးတယ်။ လက်ရှိအမြင့်ဆုံး ဘာရှင်းအခွဲ (ဥပမာ ၂၀၂၃.၃.၃) ကို သုံးလို့ရပါတယ်။ ၂၀၂၄/၂၅ သုံးမယ်ဆုံးရင်လည်း ပြဿနာတော့ မရှိပါဘူး။ Update ဘာရှင်းဖြစ်တဲ့အတွက် ပုံတွေမှာ ပြထားတာနဲ့တော့ ကွာခြားချက်တရာ့၊ ရှိကောင်းရှိနိုင်ပါတယ်။



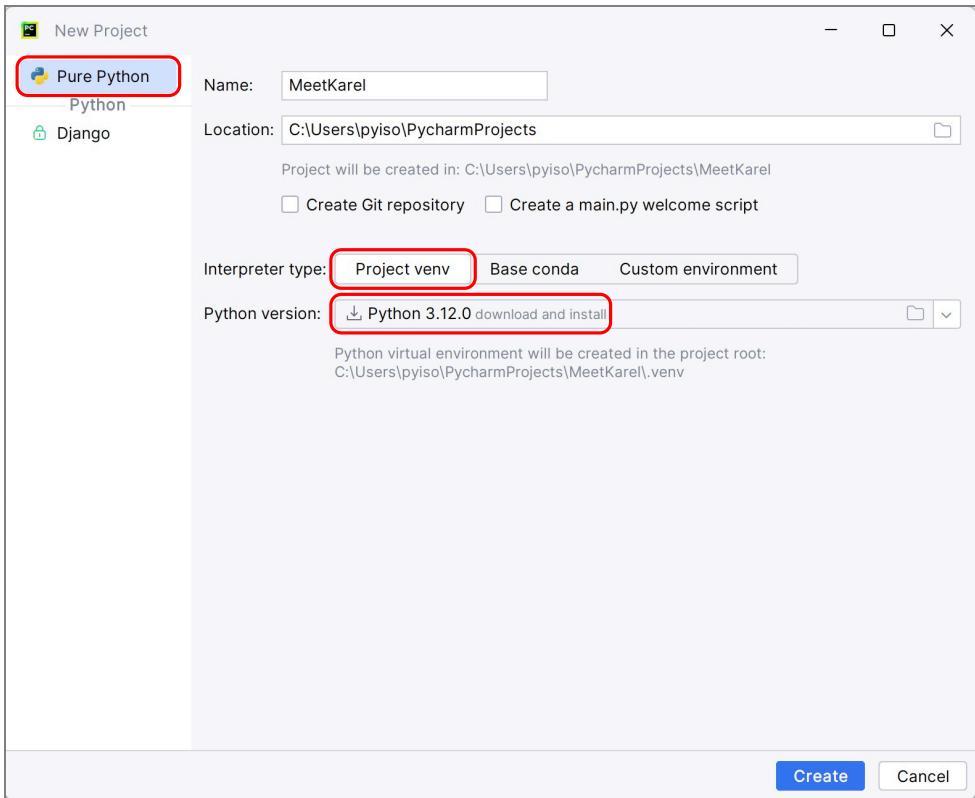
ଓ ম/ৰ



ଓ ম/ৰ

## PyCharm IDE ဆိတာဘာလ

PyCharm ဟဲ Python နဲ့ ဆော်ဖို့ ရေးဖို့အတွက် အထောက်အကူဗြိ Integrated Development Environment(IDE) ဆော်ပဲခြင်ပါတယ်။ စာစီတာရှိက်လုပ်တဲ့အခါ Microsoft Word ကို အသံးပြု ကြသလိုပဲ Python ကုဒ်ရေးဖို့ PyCharm ကိုသုံးတဲ့ သဘောပါ။ PyCharm IDE က Python ပရောဂျက်တွေ အတွက် အခိုက်ရည်ရွယ်တယ်။ အဆောက်အအီးတစ်ခု၊ တံတားတစ်ခု ဆောက်လုပ်တာ ကို ပရောဂျက်လို့ ပြောလေ့ရှိသလို ပရိုဂရမ်/ဆော်ပဲတစ်ခု တည်ဆောက်တာကိုလည်း ပရောဂျက်လို့ပဲ သုံးနှုန်းပါတယ်။ တစ်ဦးတစ်ယောက်တည်း ရေးတဲ့ ပရိုဂရမ်အသေးလေးတွေ အတွက် PyCharm ကို အသံးပြုနိုင်သလို ပရိုဂရမ်မာတွေ အဖွဲ့လိုက်နဲ့ တည်ဆောက်ရတဲ့ ပရောဂျက်ကြီးတွေ အတွက်လည်း သုံးပါတယ်။ ဒီစာအုပ်မှာတော့ PyCharm ရဲ့ အဆင့်မြင်စီချေတွေကို အသံးပြုမှာ မဟုတ်ပါဘူး။ ပရိုဂရမ်းမင်း စလေ့လာသူတွေကို လွယ်ကူးအဆင်ပြောစေတဲ့ အခြေခံ ဖီချေတွေလောက်ပဲ အသံးပြုမှုပါ။



ပုံ ၂/၁ ပရောဂျက် အသစ်ယူခြင်း

### PyCharm ပရောဂျက်ဆောက်ခြင်း

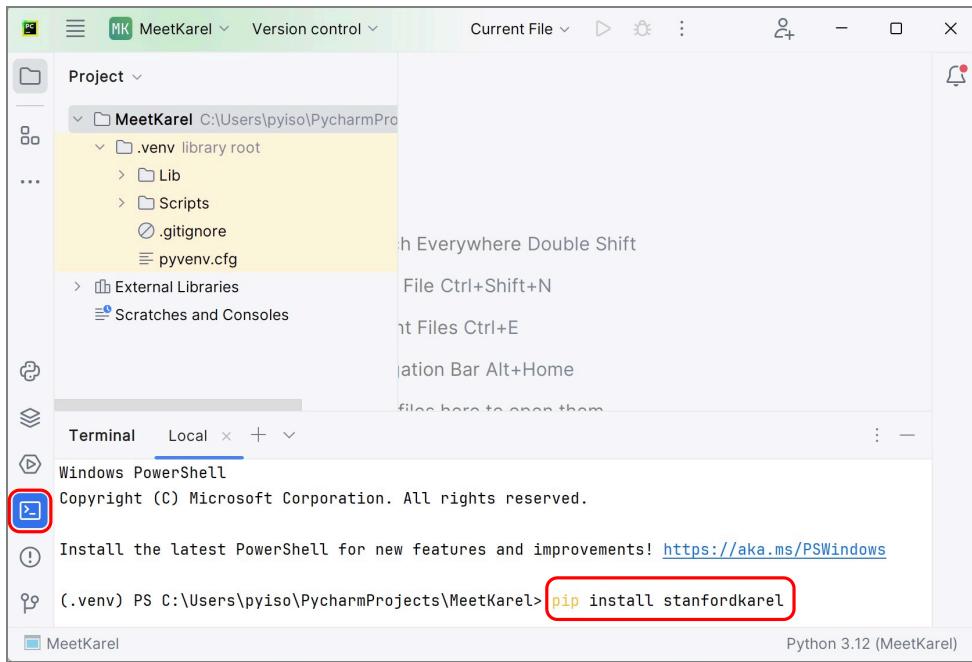
အင်စတောလုပ်ပြီးရင် PyCharm IDE ကိုဖွေ့ပြီး Welcome စခရင်မှ သိမဟုတ် File မိန္ဒီးမှ New Project နှင့်ပြီး ပရောဂျက် အသစ်ယူပါ (ပြီးခဲတဲ့ ပုံ ၂/၁ ကို ပြန်ကြည့်ပါ)။ နံမည်ကို MeetKarel သေးပါ။ ပုံ (၂/၁) မှာ တွေ့ရတဲ့အတိုင်း Pure Python, Proj venv နဲ့ Python ဘားရှင်း 3.12.xx ကို ရွေးပါ။ xx က အခြားကဏ္ဍား ဖြစ်နေနိုင်တယ်။ အစိက ဘားရှင်း 3.12 သာ ဖြစ်ပါခေါ် Create ခလုတ်နှင့်ပါ။ ပရောဂျက်အတွက် Python ကို အင်စတောလ် လုပ်ပါလိမ့်မယ်။

မိမိလေ့လေနေတဲ့ အခန်းတစ်ခန်းချင်းစီအတွက် ပရောဂျက်တစ်ခဲ ဆောက်နိုင်ပါတယ်။ အခန်း (၂) အတွက် Chapter02၊ အခန်း (၃) အတွက် Chapter03 စသည်ဖြင့်။ သက်ဆိုင်ရာအခန်းအလိုက် ကုဒ်ဖိုင်တွေကို ပရောဂျက်တစ်ခုစီမှာ ထားတဲ့အတွက် ဖိုင်တွေများပြီး ရှုပ်ထောင်းပွဲနေတဲ့ ပြဿနာ မရှိတော့ဘူး။ ကုဒ်ဖိုင်တွေကို ပရောဂျက်တစ်ခုလဲမှာပဲ ဖိုဒ်အလိုက်ခဲ့ထားလို့ရပေမဲ့ စလေ့လာသူအနေနဲ့ ပရောဂျက်တစ်ခုချင်း ခဲ့ထားတာလောက် မလွယ်ကူဘူး။

ပရောဂျက် အသစ်ယူတဲ့အခါ တည်နေရာ Location: ကို သူ့ရှိအတိုင်း ထားနိုင်သလို မိမိထားချင်တဲ့နေရာကို ညာဘာက်စွန်း ဖိုဒ်အိုင်ကွန်လေးနှင့်ပြီး ပြောင်းလို့ရတယ်။

### stanfordkarel လိုက်ဘရီ အင်စတောလ်လုပ်ခြင်း

ပုံ (၂/၆) မှာ အနီရောင် ဝိုင်းပြထားတဲ့ အိုင်ကွန်ကို နှင့်ပြီး Terminal ကိုဖွေ့ပြီး Terminal မှာ အောက်ပါ ကွန်မန်းဖြင့်



## ပုံ ၂/၆ Karel လိုက်ဘရီ အင်စတောလ်လုပ်ခြင်း

```
pip install stanfordkarel
```

ကားရဲလ်လိုက်ဘရီကို အင်စတောလ်လုပ်ပါ။ ပုံ (၂/၆) မှာ အနီးဆုံး ပိုင်းပြထားပါတယ်။ ခက္ကာတဲ့ အခါ အခုလို မက်ဆွဲချုပ်တွေ ကျလာပါလိမ့်မယ်။

```
Collecting stanfordkarel
  Downloading stanfordkarel-0.2.7-py3-none-any.whl (51 kB)
  ━━━━━━━━━━━━━━━━ 51.9/51.9 kB 443.1 kB/s
    eta 0:00:00
Installing collected packages: stanfordkarel
Successfully installed stanfordkarel-0.2.7
```

```
[notice] A new release of pip is available: 23.2.1 -> 24.0
[notice] To update, run: python.exe -m pip install --upgrade pip
(.venv) PS C:\Users\pyiso\PycharmProjects\MeetKarel>
```

ဟိုက်လိုက်ပြထားတဲ့ မက်ဆွဲချုပ် တွေ့ရရင် အင်စတောလ်လုပ်တာ အောင်မြင်လိုပါ။

မှတ်ချက်။ ပရောဂျက် အသစ်ဆောက်တိုင်း လိုအပ်တဲ့ လိုက်ဘရီကို တစ်ခါတပ်ပြီး အင်စတောလ် လုပ်ရပါမယ်။ ကားရဲလ်အခန်း ပရောဂျက်တစ်ခုစီအတွက် stanfordkarel လိုက်ဘရီကို အထက် ဖော်ပြပါအတိုင်း အင်စတောလ် လုပ်ဖို့လိုတယ်။

## နမူနာ ကားရဲလ် ကဗ္ဗာနှင့် ပရိဂရမ်ဘ် ဖိုင်များထည့်ခြင်း

meet\_karel.zip ဖိုင်ကို ဒီလင့် <http://tinyurl.com/3mm9c7j> ကနေ အောင်လုပ်လုပ်ပါ။ ငှါး zip ဖိုင်ကို extract လုပ်ပါ။ meet\_karel နံမည်နဲ့ ဖိုဒါတစ်ခု ရလာပါမယ်။ ငှါးဖိုဒါထဲမှ အောက်ပါ worlds ဖိုဒါနှင့် .py ဖိုင်အားလုံးကို ကော်ပီလုပ်ပါ။

- worlds
  - ▶ meet\_karel.w
  - ▶ move\_beeper\_to\_other\_side.w
- meet\_karel.py
- move\_beeper\_to\_other\_side.py
- world\_editor.py

MeetKarel ပရောဂျက်ထဲတွင် ကူးထည့်ပါ။ ပင်မ ပရောဂျက် MeetKarel (ပုံ က/၃ မှာ မြှားပြထား) ပေါ်မှာ ညာကလစ်နှင့်ပြီး Paste လုပ်ရမှာပါ။ ကော်ပီကူးထည့်လိုက်တဲ့ ဖိုင်တွေက ပုံမှာ တွေ့ရတဲ့ အတိုင်း MeetKarel ဖိုဒါအောက်မှာ ရှိသွေ့ပါတယ်။

အခန်းအလိုက် နှုံးနှုံး ကုဒ်ဖိုင်တွေ ထည့်ပေးထားတဲ့ .zip ဖိုင်တွေကိုလည်း အထက်ပါအတိုင်း အလားတူ လုပ်ရပါမယ်။ ပရောဂျက်အသစ်ဆောက် .zip ဖိုင်ကို ဖြည့်၊ ရလာတဲ့ ဖိုဒါထဲက ဖိုင်တွေကို ပင်မ ပရောဂျက် ဖိုဒါထဲ ကော်ပီကူးထည့် ရုံပါပဲ။

meet\_karel.py ဖိုင်ကို ကလစ်နှစ်ချက်နှင့်ပွင့်ပါ။ ပုံ (က/၃) မှာ အနိဂုံးထားတဲ့ ကုဒ်အယ်ဒီတာ (code editor) ပွင့်လာပါမယ်။ အဲဒီ မူရင်းဖိုင်မှာ အောက်ပါအတိုင်း ဆက်လက် ပြင်ဆင်ဖြည့်စွာက်ပါ။

```
from stanfordkarel import *

def main():
    """Karel code goes here!"""
    move()
    move()
    move()
    pick_beeper()
    turn_left()
    move()
    move()

    turn_left()
    turn_left()
    turn_left()

    move()
    put_beeper()

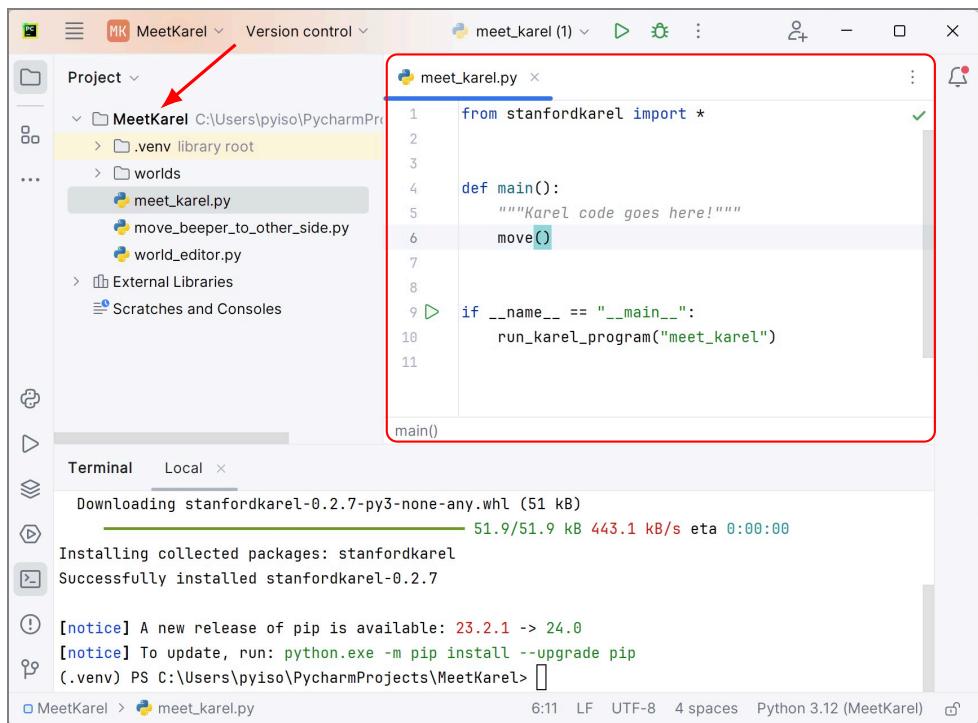
if __name__ == "__main__":
    run_karel()
```

၂၆

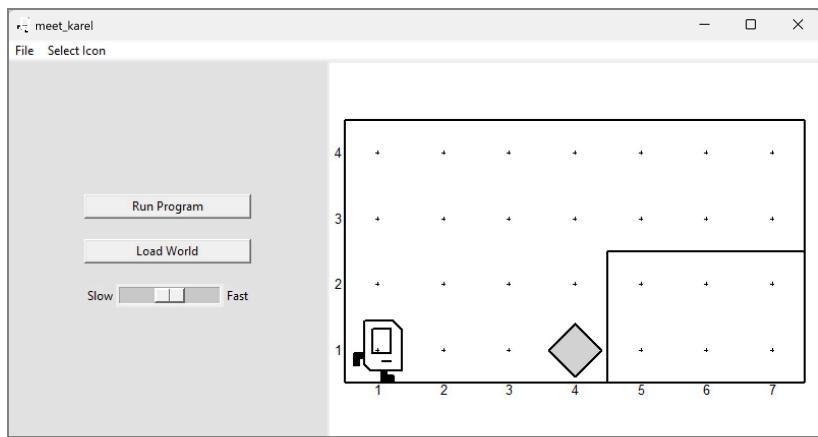
```
run_karel_program("meet_karel")
```

run ထားတဲ့ ပရိုကရမ်တစ်ခုကို တစ်ခါထပ် run ရင် Cancel (သို့) Stop and Rerun လုပ်မှုလား မေးတယ်။ Stop and Rerun လုပ်ရပါတယ်။

meet\_karel.py ဖိုင်ကို ညာကလစ်နှုမြို့း Run 'meet\_karel' လုပ်ပါ (အယ်ဒီတာ ဧရိယာမှာ ညာကလစ်နှုမြို့း Run 'meet\_karel' လုပ်လိုလည်း ရပါတယ်)။ ပုံ (က/စ) က ကားရဲ့လ် ပရိုကရမ် တက်လာရင် Run Program ခလုတ်ကိုနှိပ်ပါ။ ဘိပါကို ကားရဲ့လ်က ခွဲပေးပါလိမ့်မယ်။



ပုံ ၃/၄



ج ۶ ج

## ဆင်းတက်စံအမှားများ

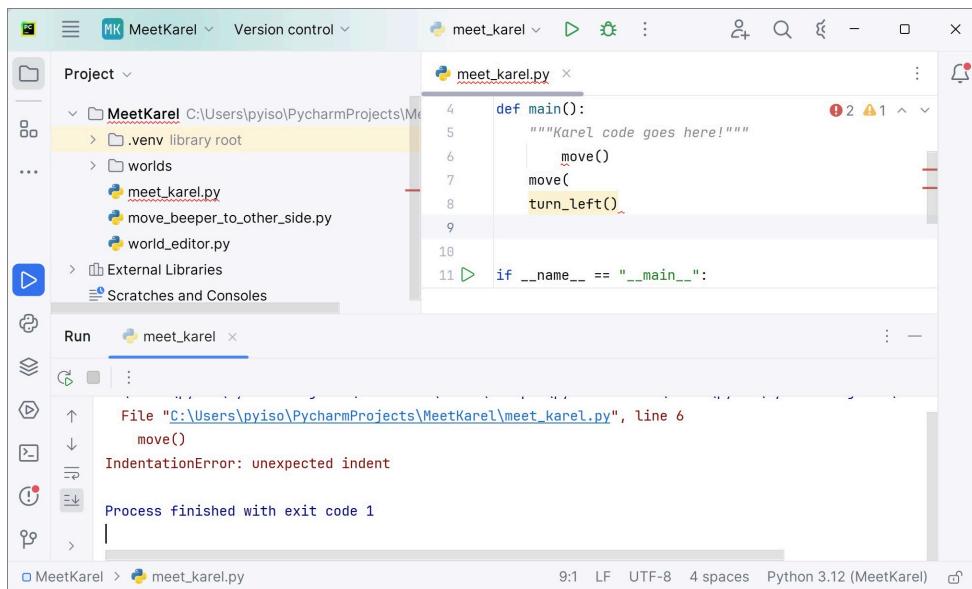
အကယ်၍ ပရိုကရမဲ့ run လို့မရရင် ကုဒ်ရေးတာမှားနေလို့ ဖြစ်နိုင်တယ်။ မိမိရေးထားတာကို စာမျက်နှာ (၂၆၀) က ပရိုကမ်းကုဒ်နဲ့ နိုင်းယူဉ် စစ်ဆေးကြည့်ပါ။ PyCharm အယ်ဒီတာမှာ အနီလှိုင်တွေ့လေး တွေ (ပုံ ၂/၉) ပြတဲ့နေရင် အဲဒီနေရာတွေမှာ ဆင်းတက်စံမှားနေလို့ (သို့) လိုက်ဘရီမထည့်ရသေးလို့ပါ။

လိုက်ကွင်းကျို့နေတာက အဖြစ်များတဲ့ အမှားပါ။ ကျို့ခဲ့လို့ မရပါဘူး။ အင်ဒန်တေးရှင်း (indentation) လုပ်ရမဲ့နေရာမှာ မလုပ်ထားရင်လည်း ပြဿနာဖြစ်တယ်။ move, turn\_left တွေကို ဘေးမှု၏ ညာဘက်ဆွဲပြီး အင်ဒန်လုပ်ပေးရမယ်။ အဲဒီတွေ ဂရုမစိုက်မိရင် ဆင်းတက်စံအမှားဖြစ်ပြီး ပရိုကရမဲ့ run လို့ မရနိုင်ဘူး။

Terminal မှာ ထုတ်ပေးတဲ့ မက်ဆွဲချုပ်တွေကို ကြည့်ပြီးတော့လည်း ဘာပြဿနာဖြစ်နေလဲ မှန်းဆ လို့ရနိုင်တယ်။ ဘာကြောင့်ဖြစ်နိုင်လဲ ဆက်စပ်စဉ်းတားလို့ ရတယ်။ ဥပမာ ဖြစ်တဲ့ပြဿနာအလိုက် အခုလို တွေ့ရပါမယ်။

```
File "c:\Users\pyiso\VS Code\meet_karel\meet_karel.py", line 6
    move(
        ^
SyntaxError: '(' was never closed
```

```
File "c:\Users\pyiso\VS Code\meet_karel\meet_karel.py", line 7
    move()
        ^
IndentationError: unexpected indent
```

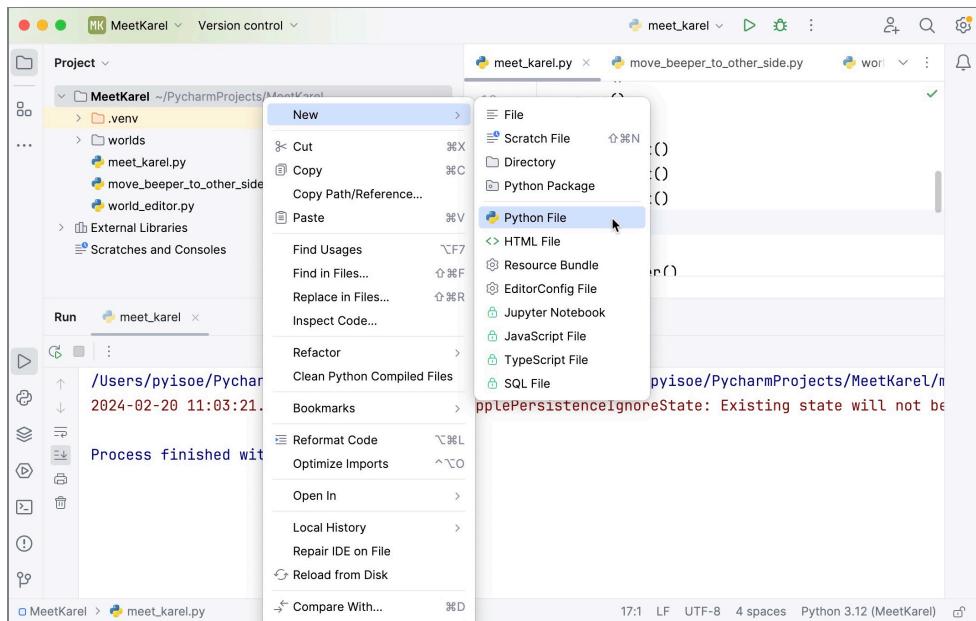


```
Traceback (most recent call last):
  File "c:\Users\pyiso\VS Code\meet_karel\meet_karel.py", line 1,
    in <module>
      from stanfordkarel import *
ModuleNotFoundError: No module named 'stanfordkarel'
```

## Python ဖိုင် အသစ်ယူခြင်း

MeetKarel ပင်မ ပရောဂျက်ဖိုဒီပေါ်မှာ ညာကလစန်ပြီး Python ဖိုင် အသစ်ယူနိုင်ပါတယ်။ Python ဖိုင်တွေက .py အိုပ်စ်တန်းရှင်းနှုပါ။ ကားရဲလ်ပရိုဂရမ်တစ်ခုကို Python ဖိုင်တစ်ခု ထားပါမယ်။ ပင်မ ပရောဂျက်ဖိုဒီ အောက်မှာပဲ တိုက်ရိုက်ရှိရပါမယ်။

နောက်ပိုင်း အဆင့်မြှင့်လာရင် ပရိုဂရမ်တစ်ခုအတွက် ပရောဂျက်တစ်ခု ထားနိုင်တယ်။ ကုဒ်ဖိုင်တွေ အပြင် ပရိုဂရမ်အတွက် လိုအပ်တဲ့ ရုပ်ပုံတွေ၊ အခြားဖိုင်တွေ (config ဖိုင်၊ setting ဖိုင် စသည်ဖြင့်) လည်း ပါနိုင်တယ်။ ပင်မပရောဂျက် အောက်မှာပဲ ဖိုင်တွေက တိုက်ရိုက်ရှိဖို့လည်း မလိုတေဘာ့ဘူး။ ဆက် စပ်ရာ ဖိုင်တွေကို အမျိုးအစားအလိုက် ဖန်ရှင်အလိုက် ဖို့ဒါတွေခဲ့ပြီး စနစ်ကျ စီစဉ်ဖွဲ့စည်း ထားရမှာပါ။ ပရောဂျက်တစ်ခုမှာ ဖိုင်တွေကို စနစ်တကျ စုဖွဲ့ထားဖို့ အရေးကြီးပါတယ်။



ပုံ ၃/၁၀

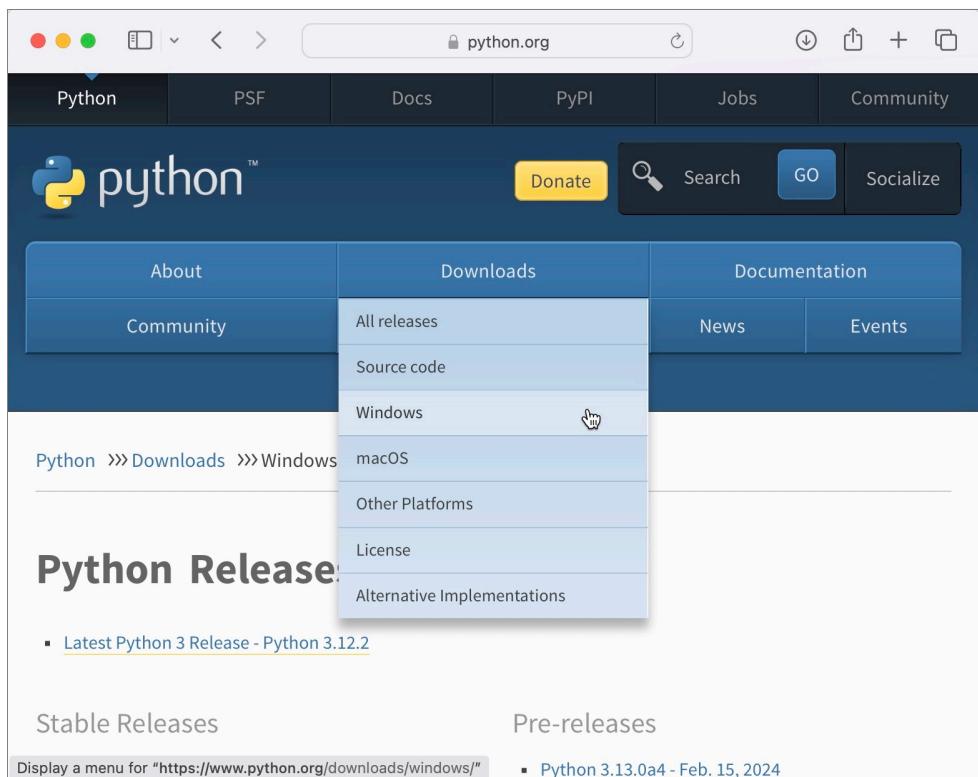
## Visual Studio Code နှင့် Python အင်စတောလ်လုပ်ခြင်း

Visual Studio Code(VS Code) ဟာ ပရိုဂရမ်မာအများစုံ ကြိုက်နှစ်သက်တဲ့ မောဒန် ကုဒ်အယ်ဒီတာ တစ်ခုပါ။ Python, JavaScript, C++ စတဲ့ programming language အမျိုးမျိုးအတွက် အသုံးပြု နိုင်ပါတယ်။

PyCharm မှာတော့ ပရောဂျက်အသစ်ဆောက်ရင် Python ပါ တစ်ခါတည်း ဒေါင်းလုဒ်လုပ်ပြီး အင်စတောလ် လုပ်လိုရတယ်။ VS Code နဲ့ Python ရေးမယ်ဆိုရင် Python Programming Language ကို သိုံးခြား ဒေါင်းလုဒ်လုပ်ပြီး အင်စတောလ် လုပ်ရပါမယ်။

### Python အင်စတောလ်လုပ်ခြင်း

ဒီလုပ် <https://www.python.org/> ကိုဖွေ့စ်ပါ။ Download မိန္ဒီးမှ Windows နိုပ်ပါ (ပုံ က/၁၁ ကို ကြည့်ပါ)။ Apple ကွန်ပျူးတာအတွက် ဆိုရင် macOS ရွေးရပါမယ်။ လူများစုသုံးတဲ့ ပိုက်ခရီးဆော် ဝင်း ခြီးအတွက် အင်စတောလ်လုပ်နည်းကို အခိုက်ပြေားမှာပါ။ ပုံ က/၁၂ မှာလို တွေ့ရပါလိမ့်မယ်။ အင်စတောလ် လာ ဘာရှင်း 3.12 ထဲက လက်ရှိအမြင့်ဆုံး ကိုရွေးပါ။ ဒီစာရေးချိန်မှာ 3.12.2 ဟာ အမြင့်ဆုံးဘာရှင်းပါ။ Windows Installer (64-bit) ကိုနိုပ်ပြီး ဒေါင်းလုဒ်လုပ်ပါ။



ပုံ က/၀၁

ဒေါင်းလုဒ်ပြီးရင် အင်စတောလ်လာဖိုင်ကို ညာကလစ်နိုပ်ပြီး Run as administrator လုပ်ပါ။ ပုံ (က/၁၃) ကိုကြည့်ပါ။ ပုံ (က/၁၄) မှာလို ဒိုင်ယာလော် ဆောက်စ် ပွင့်လာပါမယ်။ အနိဂုင်းထားတဲ့ ချက်ချေ ဆောက်စ်နှစ်ခုကို ချက်ချေလုပ်ပြီး Install Now နိုပ်ပါ။ အင်စတောလ် ပြီးလို Setup was successful

၂၆

Note that Python 3.12.2 cannot be used on Windows 7 or earlier.

- Download Windows embeddable package (32-bit)
- Download Windows embeddable package (64-bit)
- Download Windows embeddable package (ARM64)
- Download Windows installer (32-bit)
- **Download Windows installer (64-bit)**
- Download Windows installer (ARM64)

▪ Python 3.12.1 - Dec. 8, 2023

Note that Python 3.12.1 cannot be used on Windows 7 or earlier.

- Download Windows embeddable package (32-bit)
- Download Windows embeddable package (64-bit)
- Download Windows embeddable package (ARM64)
- Download Windows installer (32-bit)
- Download Windows installer (64-bit)

▪ Python 3.13.0a2 - Nov. 21, 2023

- Download Windows embeddable package (32-bit)
- Download Windows embeddable package (64-bit)
- Download Windows embeddable package (ARM64)
- Download Windows installer (32-bit)
- Download Windows installer (64-bit)
- Download Windows installer (ARM64)

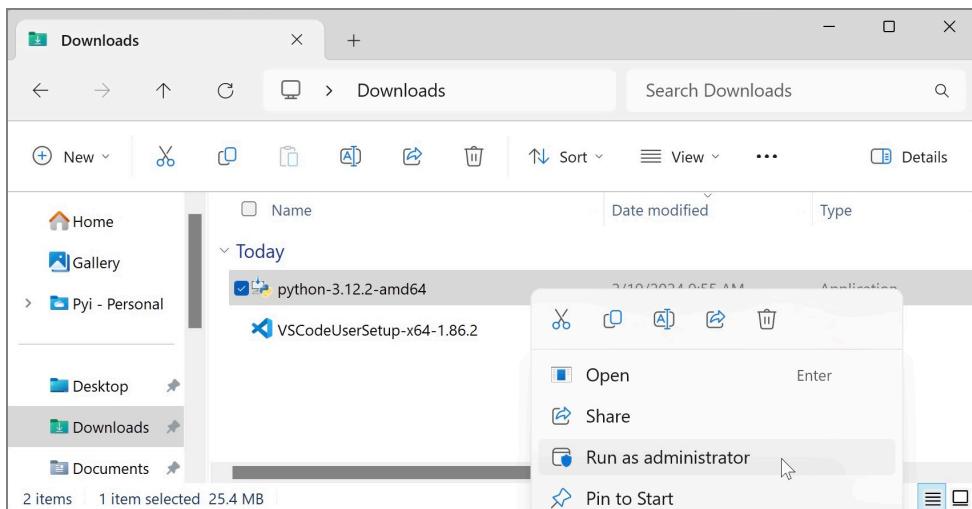
▪ Python 3.13.0a1 - Oct. 13, 2023

- Download Windows embeddable package (32-bit)
- Download Windows embeddable package (64-bit)
- Download Windows embeddable package (ARM64)
- Download Windows installer (32-bit)
- Download Windows installer (64-bit)
- Download Windows installer (ARM64)

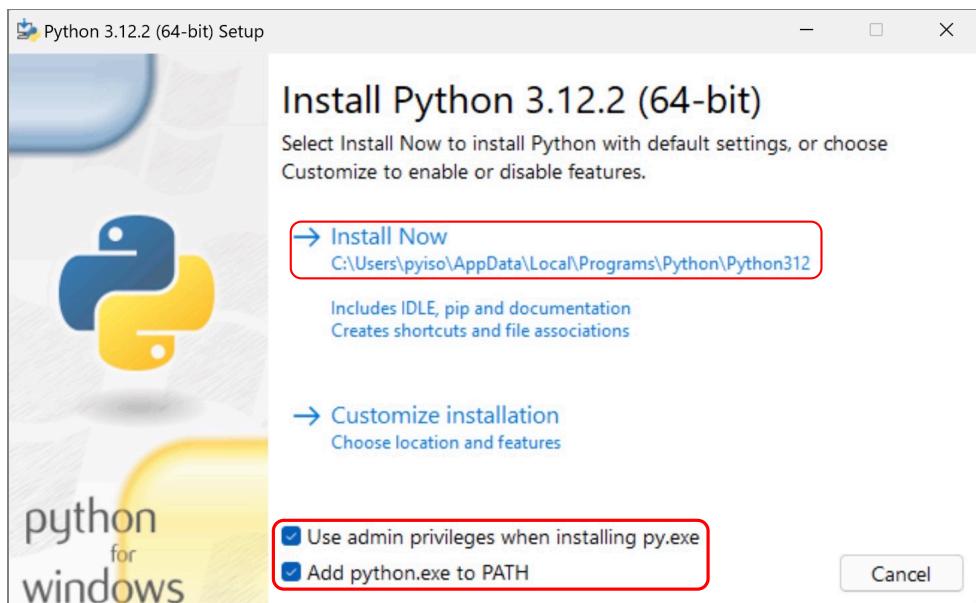
ပုံ ၃/၁၂

ပေါ်လာရင် Close ခလုတ်နိုပ်ပြီး ပိတ်ပါ။

ဝင်းဒီး command prompt (cmd) မှာ `python --version` run ရင် ပုံ (၃/၁၅) မှာလို အင်စကောလုပ်ထားတဲ့ ဗားရှင်းကို ပြပေးသင့်ပါတယ်။



ပုံ ၃/၁၃



ڦ گ/ڳ

The screenshot shows a Microsoft Windows Command Prompt window titled "Command Prompt". The text output is as follows:

```
Microsoft Windows [Version 10.0.22621.2861]
(c) Microsoft Corporation. All rights reserved.

C:\Users\pyiso>python --version
Python 3.12.2

C:\Users\pyiso>
```

ڦ گ/ڳ

## VS Code အင်စတေလုပ်ခြင်း

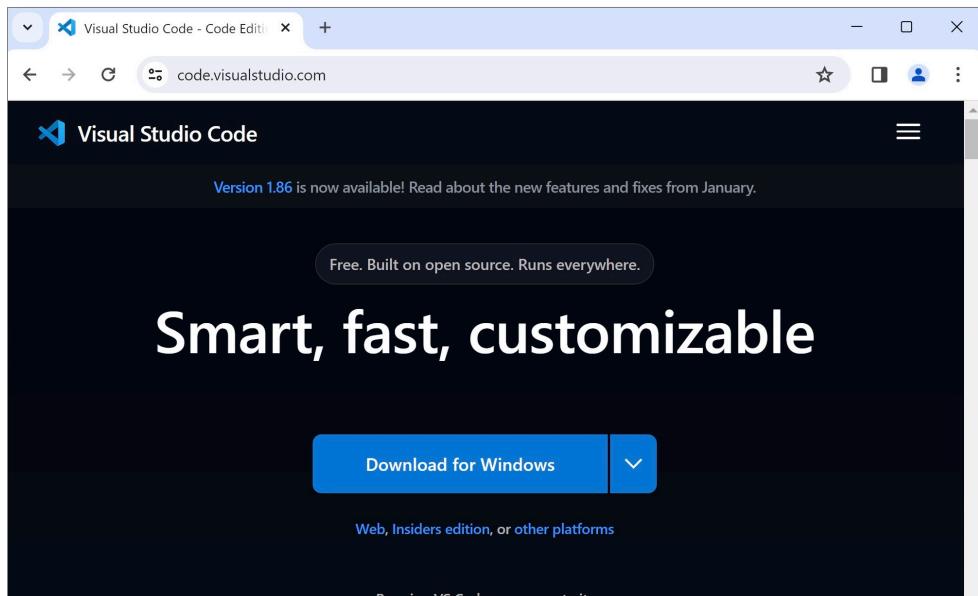
အခုနိရင် Python programming language အောင်မြင်စွာ ထည့်ပြီးသွားပါပြီ။ VS Code အင်စတေလုပ်ခြင်း ဆက်လုပ်ပါမယ်။ အင်စတေလုပ်ခြင်း ဒေါင်းလုပ်လုပ်ရန် ဝေါဘာမျက်နှာကို အောက်ပါလုပ်ခြင်း ဖြစ်ပါသည်။

<https://code.visualstudio.com>

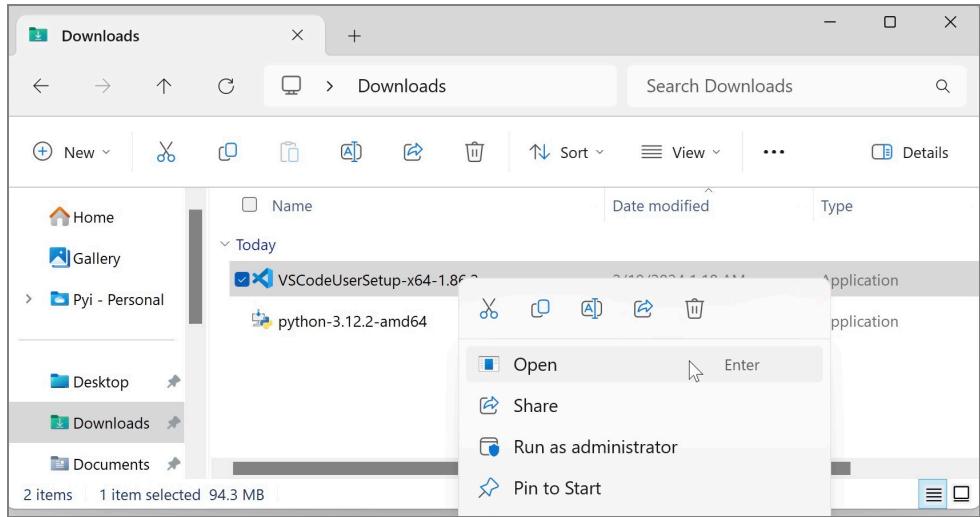
မှတစ်ဆင့် သွားပါ။ ပုံ (က/၁၆) ဝေါဘာမျက်နှာကို တွေ့ရပါမယ်။ [Download for Windows](#) နှုပ်၍ ဒေါင်းလုပ်လုပ်ပါ။

ပြီးတဲ့အခါ အင်စတေလုပ်ခြင်းကို ညာကလစ်နှုပ်ပြီး ဖွင့်ပါ (ပုံ က/၁၇ မှာ ပြထားပါတယ်)။ အင်စတေလုပ်ခြင်းကို ချက်ချွဲလုပ်လုပ်၍ ပွင့်လာရင် [I accept the agreement](#) ကို ချက်ချွဲလုပ်၍ Next > တစ်ခု ပြီးတစ်ခု ဆက်နိုပ်သွားပြီး နောက်ဆုံးမှာ Install နှုပ်ပါ။ အင်စတေလုပ်ခြင်းကို ခက္ကစားပြီး ပြီးသွားရင် Finish နှုပ်ပါ။

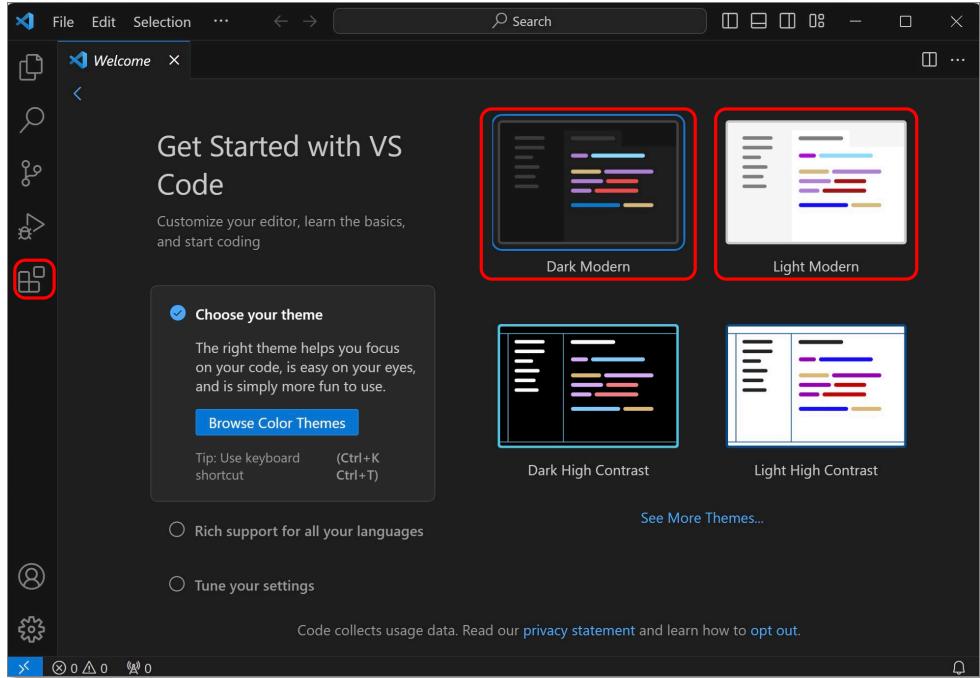
Welcome စခရင်ကို ပုံ (က/၁၈) လို တွေ့ရပါမယ်။ [Dark Modern](#) (သို့) [Light Modern](#) နှစ်သက်ရာ သီးမ်းရွေးပါ။ အဲဆုံးရင် VS Code လည်း အင်စတေလုပ်ခြင်းပြီး ကားရဲ့လုပ်ရမ်းများပါ။



ပုံ က/၁၆



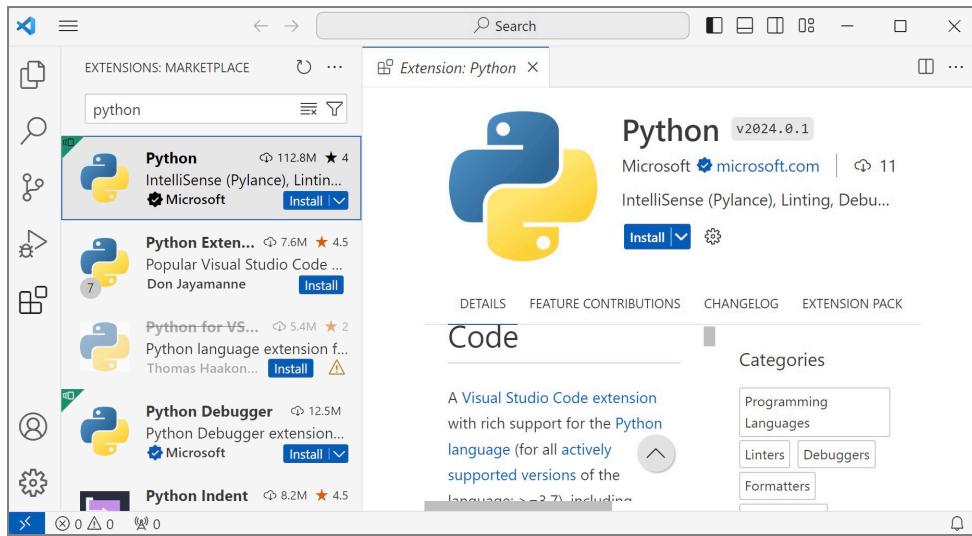
ଓঁ গো



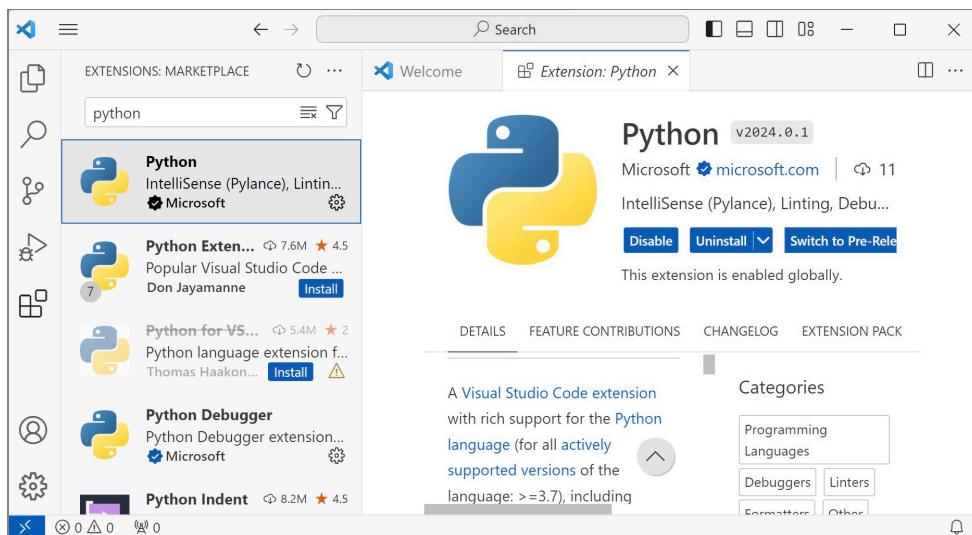
ଓঁ গো

## VS Code Python Extension တည်ခြင်း

Python ပရိဂုဂ္ဂမရေးဖို့အတွက် VS Code က ဒီအတိုင်းဆိုရင် သိပ်အဆင်မပြသေးပါဘူး။ Python အတွက် extension အင်စတောလ် လုပ်ပေးရပါအိုးမယ်။ ပုံ (က/၁၈) ဘယ်ဘက်သော်နားမှာ အနီးပိုင်းထားတဲ့ အိုင်ကွန်လေးကို နှိပ်ပါ။ Python extension ကိုရှာပါ။ ပုံ (က/၁၉) မှာ ဘယ်လိုရှာရမလဲ ပြထားပါတယ်။ ပုံမှာတွေ့ရတဲ့ Microsoft က ထုတ်တဲ့ extension ကို အင်စတောလ်လုပ်ပါ။ အောင်မြင်ရင် ပုံ (က/၂၀) မှာလို ဖြစ်သွားပါမယ်။ Python နဲ့ VS Code ကိစ္စတော့ ပြီးသွားပြီ။ ကားရဲ့လဲ example run ဖို့ ဆက်လုပ်ရမယ်။



ပုံ က/၁၉

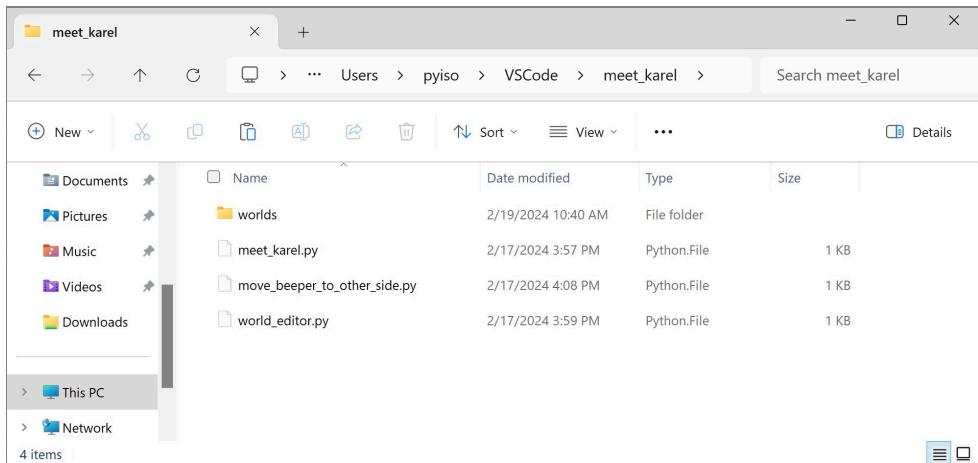


ပုံ က/၂၀

## နမူနာ ကားရဲလ် ကဗ္ဗာနှင့် ပရိဂရမ်ဘ် ဖိုင်များထည့်ခြင်း

meet\_karel.zip ဖိုင်ကို ဒီလင့် <http://tinyurl.com/3mm9c7j> ကနေ အောင်လုပ်လုပ်ပါ။ ငါး zip ဖိုင်ကို extract လုပ်ပါ။ meet\_karel နံမည်နဲ့ ဖိုဒါတစ်ခု ရလာပါမယ်။ ဖိုဒါထဲ ဝင်ကြည့်ရင် အောက်ပါ အတိုင်း ရှိသင့်ပါတယ်။

- worlds
  - ▶ meet\_karel.w
  - ▶ move\_beeper\_to\_other\_side.w
- meet\_karel.py
- move\_beeper\_to\_other\_side.py
- world\_editor.py



## ပုံ က/ဂ

VS Code အတွက် ဖိုဒါတစ်ခုကို မိမိအတွက် အဆင်ပြေမဲ့နေရာမှာ သီးသန့်တည်ဆောက် ထားသင့်တယ်။ ဥပမာ

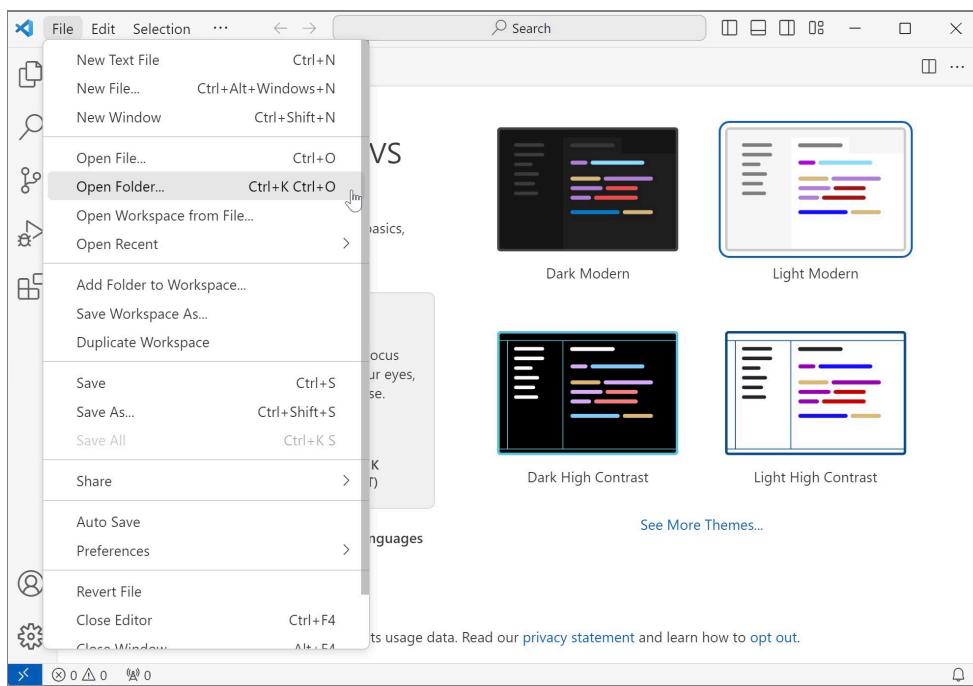
C:\Users\yourname\VS Code

မိမိ လက်ရှိ Home ဖိုဒါကို C: drive ရဲ့ Users ဖိုဒါထဲမှာ တွေ့နိုင်ပါတယ်။ Win + R ရှေ့ကတ် နှုပ်ပြီး %userprofile% ရှိက်ထည့်၍ Ok လုပ်ပြီး Home ဖိုဒါကို သွားနိုင်ပါတယ်။ အကယ်၍ မသွားတတ်ရင်လည်း ပြဿနာမရှိပါဘူး။ ကိုယ့်အတွက် လွယ်ကူမဲ့ Desktop, Downloads, Documents တစ်ခုခဲ့ထဲမှာ VS Code အတွက် ဖိုဒါတစ်ခု ထားလည်းရတယ်။

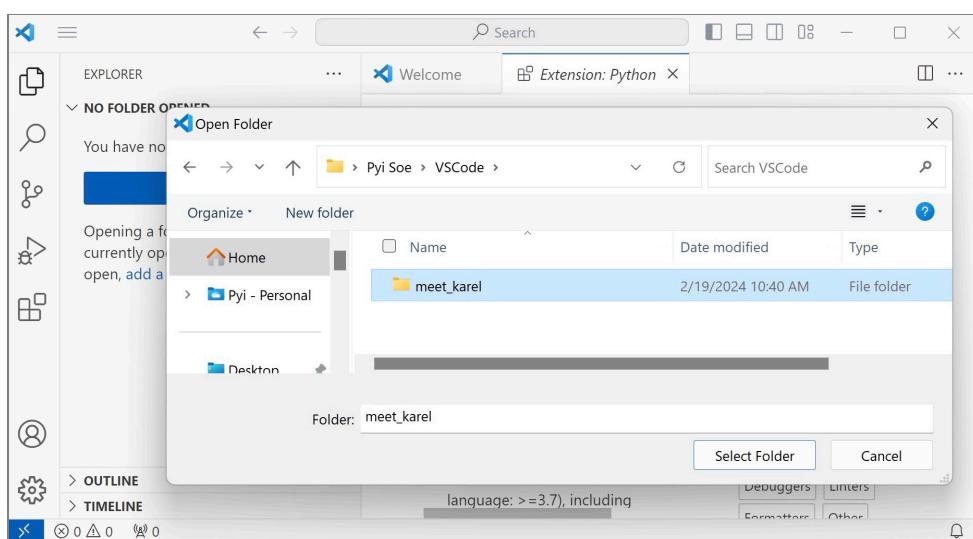
`meet_karel` ဖိုဒါ (.zip ဖိုင်မဟုတ်ပါ) ကို အထက်ပါအတိုင်း အသစ် ဆောက်ထားတဲ့ VS Code သီးသန့်ဖိုဒါထဲကို ကော်ပိကူးထည့်ပါ။ ငါး `meet_karel` ဖိုဒါကို VS Code `File` မိန္ဒာ `Open Folder` နှုပ်၍ ဖွဲ့ပါ။ ပုံ (က/ဂ) တွင်ကြည့်ပါ။

အခန်းအလိုက် နှုန်းကုဒ်ဖိုင်တွေ ထည့်ပေးထားတဲ့ .zip ဖိုင်တွေကိုလည်း အထက်ပါအတိုင်း လုပ်ရပါမယ်။ .zip ဖိုင်ကို ဖြည့်၊ ရလာတဲ့ ဖိုဒါကို သီးသန့်ဖိုဒါတစ်ခုထဲကို ကော်ပိကူးထည့်၍ VS Code နဲ့ အဲဒီဖိုဒါကို ဖွဲ့ရဲ့ပါပဲ။

J2R



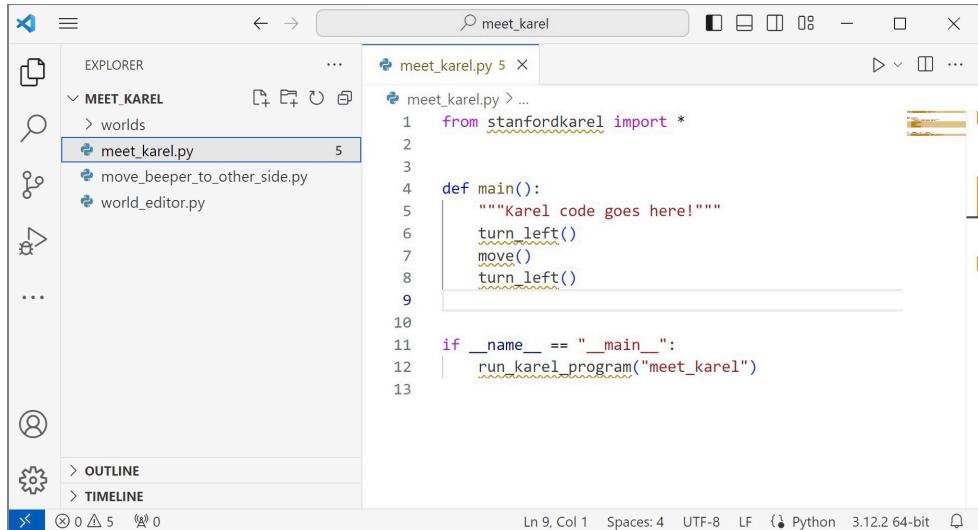
ဗုဒ္ဓန/JR



ဗုဒ္ဓန/JR

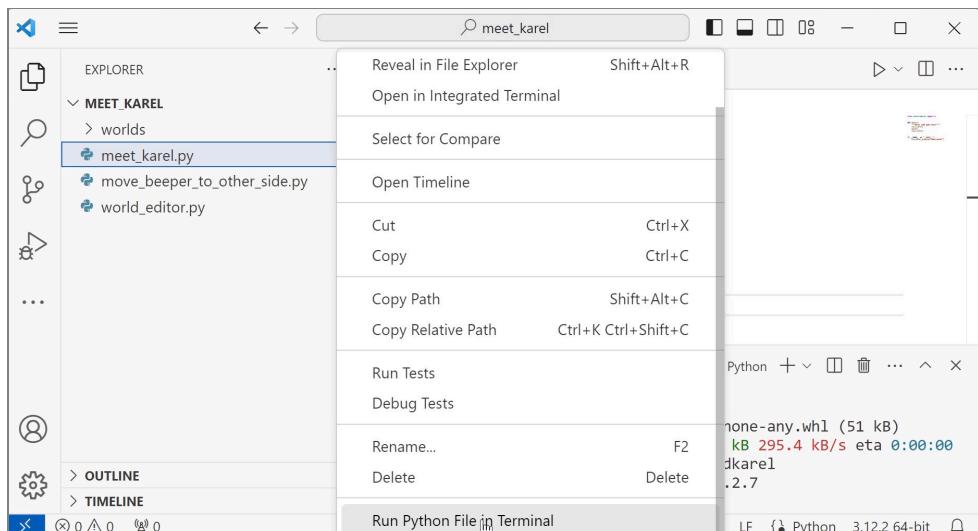
## stanfordkarel လိုက်ဘရီ အင်စတေလုပ်ခြင်း

`meet_karel.py` ဖိုင်ကို ကလစနှစ်ချက်နှင့် ဖွင့်ပါ။ ကုဒ်အယ်ဒီတာ ပွင့်လာမယ် (ပုံ ၁၂၄)။ အဲဒီ ကုဒ် အယ်ဒီတာပေါ် (သို့) `meet_karel.py` ဖိုင်ကို ညာကလစနှင့်ပြီး `Run Python File in Terminal` လုပ်ပါ (ပုံ ၁၂၅)။ Terminal ပွင့်လာပြီး အယ်ရာမက်ဆွေချုပ်တွေ ပြလိမ့်မယ်။ ပုံ (၁၂၆) မှာ ကြည့်ပါ။ ကားရုံပရှိရမ်အတွက် လိုအပ်တဲ့ stanfordkarel လိုက်ဘရီ အင်စတေလုပ်မလုပ်ရသေးပါဘူး။ ဒါ ကြောင့် အယ်ရာဖြစ်နေတာ။



```
meet_karel.py 5 ×
meet_karel.py > ...
1 from stanfordkarel import *
2
3
4 def main():
5     """Karel code goes here!"""
6     turn_left()
7     move()
8     turn_left()
9
10
11 if __name__ == "__main__":
12     run_karel_program("meet_karel")
13
```

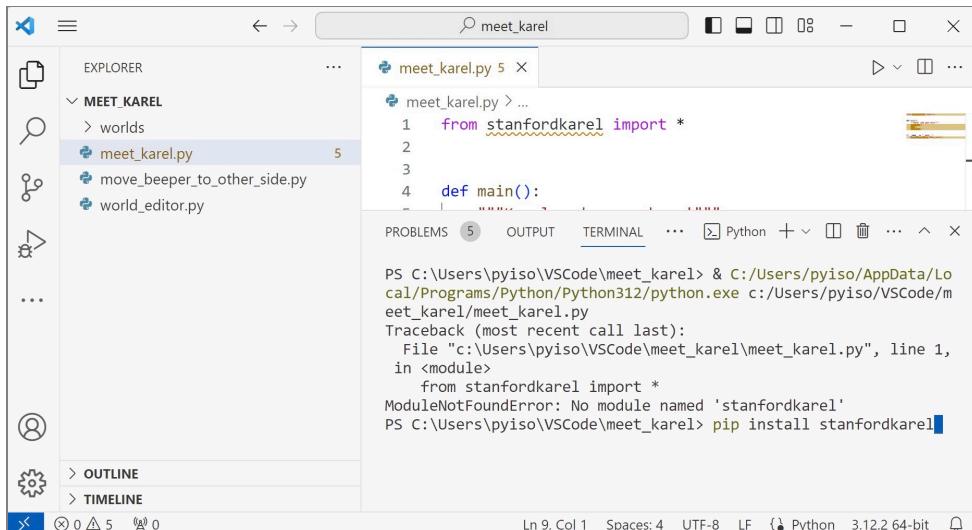
ပုံ ၁၂၄



ပုံ ၁၂၅

ခုနကပွင့်လာတဲ့ Terminal မှာပဲ အောက်ပါကွန်မန်းကို run ပြီး stanfordkarel လိုက်ဘရီကို အင်စတေလုပ်ပါ။

J29



୧୬

```
pip install stanfordkarel
```

ပုံ (က/ဂြ) မှာ အနိဂုံင်းထားတေကို ကြည့်ပါ။ အဲဒီအတိုင်းရိုက်ထည့်ပြီး Enter ကီးနှုပ်ပါ။ ခေါက္ခတဲ့ အခါ အောင် မက်ဆောင်တော် ကျေလာပါလိမ့်မယ်။

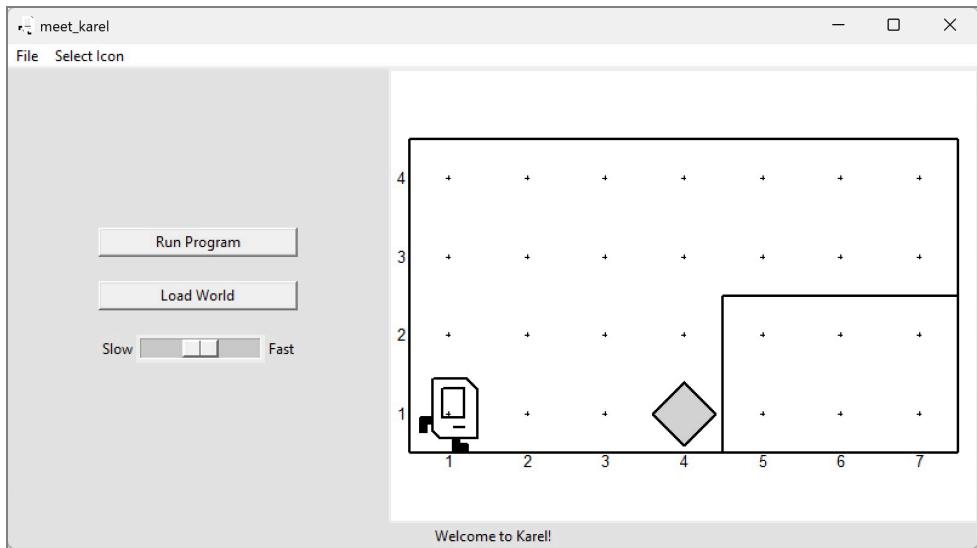
```
ModuleNotFoundError: No module named 'stanfordkarel'
PS C:\Users\pyiso\VSCode\meet_karel> pip install stanfordkarel
Collecting stanfordkarel
  Downloading stanfordkarel-0.2.7-py3-none-any.whl (51 kB)
               ━━━━━━━━━━━━━━━━ 51.9/51.9 kB 295.4 kB/s eta 0:00:00
Installing collected packages: stanfordkarel
Successfully installed stanfordkarel-0.2.7
PS C:\Users\pyiso\VSCode\meet_karel>
```

ບັງຄົນລິ້ນປຼາຍະຕູ ມາກນແຜ່ງວຸງ ຕອຸງຣາດ ຂອບເຂດຕະຫຼາດ ເພື່ອມູນຄົງລິ້ນປີ|| ບຸ້ (ກ/ຈດ) ອາຍືວິທະຍາມູນລິ້ນ  
ວທບະບາດ: ລິ້ນ:ຖົນຝູມູນຕົວ ມຽນຍາດົນຕະຫຼາກວຸງ|| stanfordkarel ລິ້ນກວາສີ ຂອບເຂດຕະຫຼາດ ລຸບປິບ:ປຶ້ມືລິ້ນ  
ວທບະບາດ:ຕາຕູ ເບົາກົວວຸງ:ວົງຕາຍ||

`meet_karel.py` ဖိုင်ကို ညာကလစ်နှုပ်ပြီး Run Python File in Terminal ပြန်လုပ်ကြည့်ပါ။ ပုံ (က/ဂ) မှာပြထားတဲ့ ကားရဲလ်ပရိုကရမ် ပွုင့်လာသင့်ပါတယ်။ Run Program နှုပ်ကြည့်ပါ။ စက်ရှုပ်လေး ကားရဲလ် နေရာရောသားတဲ့ တော်မယ်။ `meet_karel.py` ကို အောက်ပါအတိုင်း ဖော်စက်ရေးပါ။

```
from stanfordkarel import *
def main():
    """Karel code goes here!"""

```



၂၇

```
move()
move()
move()
pick_beeper()
turn_left()
move()
move()

turn_left()
turn_left()
turn_left()

move()
put_beeper()

__name__ == "__main__":
    run karel program("meet karel")
```

`meet_karel.py` ဖိုင်ကို ညာကလစ်နှပ်ပြီး Run Python File in Terminal ပြန်လည်ကြည့်ပါ။ ကားရဲ့လုပ်ပရီဂျာများ ပုံစံလာရင် `Run Program` နှိပ်ကြည့်ပါ။ ဘိပါလေးကို နေရာရွှေ့ပေးပါလိမ့်မယ်။ အကယ်၍ကားရဲ့လုပ်ပရီဂျာများ မတက်လာရင် ကုဒ်ရေးတာမှားနေလို ဖြစ်နိုင်တယ်။ အပေါ်ကုဒ်နဲ့ နှိုင်းယဉ်ပြီး ကြည့်ပါ။ VS Code အယ်ဒီတာမှာ အနိုင်တွန်းလေးတွေ ပြတဲ့နေရင် အဲဒီနေရာတွေမှာ ဆင်းတက်စ်မှားနေတာဖြစ်နိုင်တယ်။

VS Code အယ်ဒီတာမှာ ပရိုကရမ်ကုဒ်ပြင်ပြီး ပြန် run တဲ့အခါ ပထမ run ထားတဲ့ ပရိုကရမ်ကို အရင်ပိတ်ဖို့လိုပါတယ်။ ဆိုလိုတာက meet\_karel.py ကို run ထားတယ်ဆိုပါစူး။ ပုံ (က/ဂျ) က ဝင်းဒီးပင်လာမယ်။ meet\_karel.py ကုဒ်ကို ပြင်ပြီး ပြန် run ချင်ရင် အဲဒီ ဝင်းဒီးကို အရင်ပိတ်ရမယ်။ မဟုတ်ရင် ပြင်ထားတဲ့ ပရိုကရမ်က ချက်ချင်း ပွင့်မလာဘူး။ ပထမ ဟာကို ပိတ်တော့မှုပဲ နောက် run တဲ့ဟာ ပွင့်လာမှာ။

ဝိုက်ကွင်းကျို့နေတာက အဖြစ်များတဲ့ အမှားပါ။ ကျို့ခဲ့လို မရပါဘူး။ အင်ဒန်တေးရှင်း (indentation) လုပ်ရမဲ့နေရာမှာ မလျပ်ထားရင်လည်း ပြဿနာဖြစ်တယ်။ move, turn\_left တွေကို ဘေးမျဉ်းညာဘက်ဆွဲပြီး အင်ဒန်လုပ်ပေးရမယ်။ အဲဒါတွေ ကရာဇ်စိုက်မိရင် ဆင်းတက်စ်အမှားဖြစ်ပြီး ပရိုကရမ် run လို မရနိုင်ဘူး။

Terminal မှာ ထုတ်ပေးတဲ့ မက်ဆွဲချုပ်တွေကို ကြည့်ပြီးတော့လည်း ဘာပြဿနာဖြစ်နေလဲ မှန်းဆလိုရနိုင်တယ်။ ဘာကြောင့်ဖြစ်နိုင်လဲ ဆက်စပ်စဉ်းတားလို ရတယ်။ ဥပမာ ဖြစ်တဲ့ပြဿနာအလိုက် အခုလို တွေ့ရပါမယ်။

```
File "c:\Users\pyiso\VS Code\meet_karel\meet_karel.py", line 6
    move(
        ^
SyntaxError: '(' was never closed
```

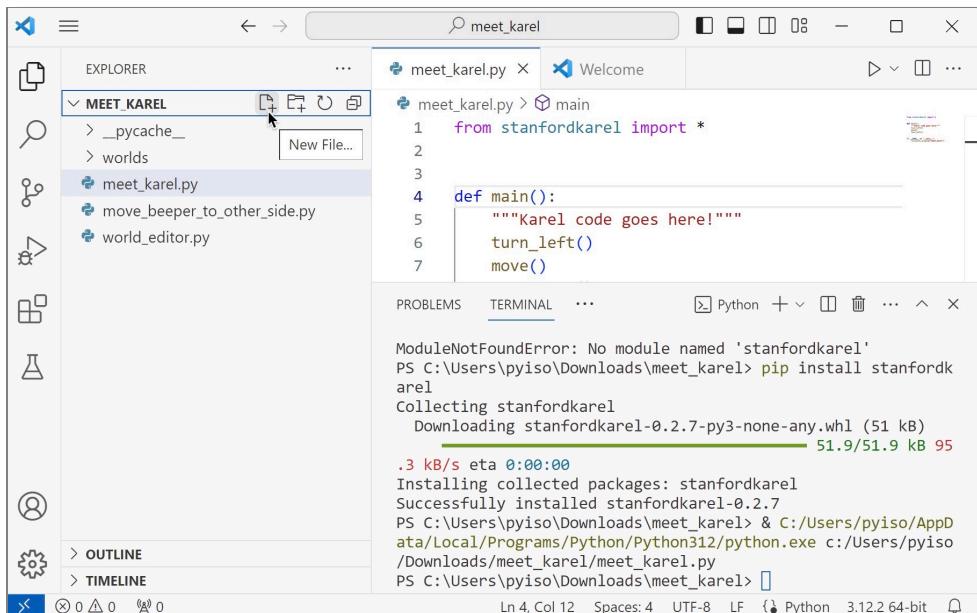
```
File "c:\Users\pyiso\VS Code\meet_karel\meet_karel.py", line 7
    move()
    ^
IndentationError: unexpected indent
```

```
Traceback (most recent call last):
File "c:\Users\pyiso\VS Code\meet_karel\meet_karel.py", line 1,
    in <module>
from stanfordkarel import *
ModuleNotFoundError: No module named 'stanfordkarel'
```

## VS Code တွင် Python ဖိုင် အသစ်ယူခြင်း

MEET\_KAREL ပင်မ ပရောဂျက်ဖိုဒီပေါ်မှာ ကလစ်နိုင်ပါ။ ပုံမှာ ပြထားတဲ့ New File အင်ကွန်ကိုနှိပ်ပါ။ ဖိုင်နံပည်ဖြည့်တဲ့ ဘောက်စံလေး ပေါ်လာမယ်။ Python ဖိုင်တွေက .py အိပ်စံတန်းရှင်းနဲ့ ဖြစ်ရပါမယ်။ ဒါကြောင့် နံပည် ဖြည့်တဲ့အခါ .py နဲ့ အဆုံးသတ်ပေးရပါမယ် (ဥပမာ hello.py)။ ကားချုပ်ပရိုဂရမ်တစ်ခုကို Python ဖိုင်တစ်ခု ထားပါမယ်။ ပင်မ ပရောဂျက်ဖိုဒီ အောက်မှာပဲ တိုက်ရှိကြရပါမယ်။

နောက်ပိုင်း၊ အဆင့်မြင့်လာရင် ပရိုဂရမ်တစ်ခုအတွက် ပရောဂျက်တစ်ခု ထားနိုင်တယ်။ ကုဒ်ဖိုင်တွေ အပြင် ပရိုဂရမ်အတွက် လိုအပ်တဲ့ ရုပ်ပုံတွေ၊ အခြားဖိုင်တွေ (config ဖိုင်၊ setting ဖိုင် စသည်ဖြင့်) လည်း ပါနိုင်တယ်။ ပင်မပရောဂျက် အောက်မှာပဲ ဖိုင်တွေက တိုက်ရှိကြရဖို့လည်း မလိုတော့ဘူး။ ဆက်စပ်ရာ ဖိုင်တွေကို အမျိုးအစားအလိုက်၊ ဖန်ရှင်အလိုက် ဖို့ဒါတွေခဲ့ခြား စနစ်ကျ စီစဉ်ဖွဲ့စည်း ထားရမှာပါ။ ပရောဂျက်တစ်ခုမှာ ဖိုင်တွေကို စနစ်တကျ စွဲထားဖို့ အရေးကြီးပါတယ်။



ပုံ ၃/၂၈

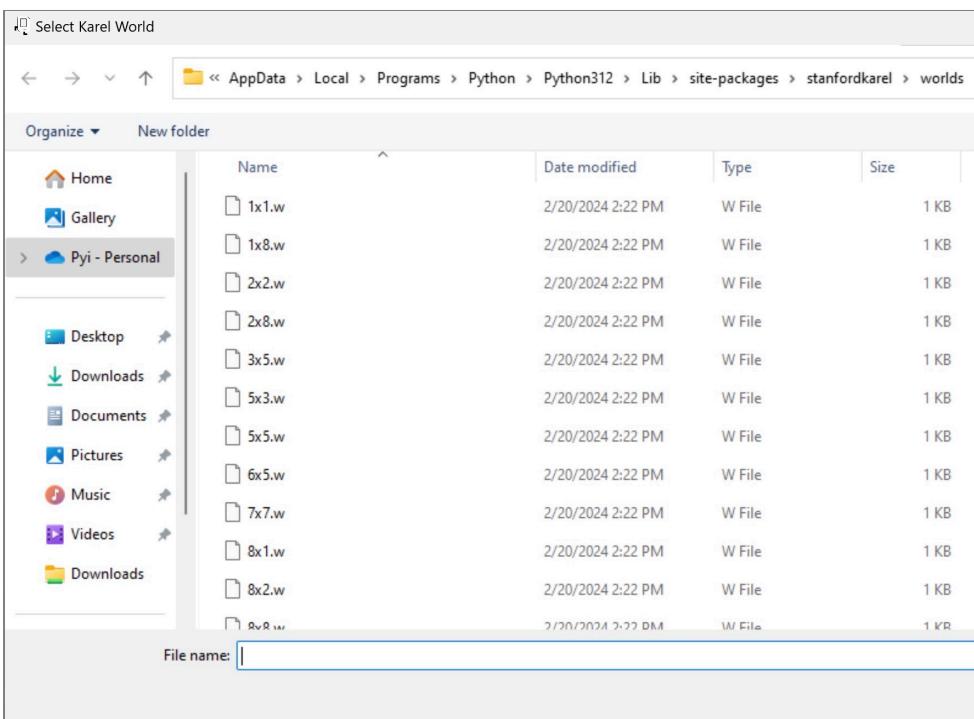


# နောက်ဆက်တွဲ ခ

## ကားရဲလ်ပရီဂရမ် ဖီချာများ

### ကားရဲလ် ကဗ္ဗာဖိုင်များ

ကားရဲလ်ပရီဂရမ် ဝင်းခွဲ့မှာ **Load World** ခလုတ်နှုပ်ပြီး ကဗ္ဗာဖိုင်အသစ် တင်လိုရတယ်။ ခလုတ်နှုပ်လိုက် ရင် အခုလို ဖိုင် ခိုင်ယာလော့ဂုံးပွင့်လာမယ်။



ပုံ ၁/၁

ဒါက stanfordkarel လိုက်ဘရဲ သူနိုင်အရို့အတိုင်း ပါတဲ့ worlds ဖို့ပါ။ ဖိုင်တွေက .w နဲ့ ဆုံးပါတယ်။ ကဗ္ဗာဖိုင်တွေကို ပင်မ ပရောဂျက်အောက် worlds ဖို့ပါထဲမှာ ထားလိုလည်းရတယ်။ အခြားနေရာတွေမှာ ထားလိုတော့ မရဘူး။

စာအုပ်ပါ ဥပမာတွေ၊ လေ့ကျင့်ခန်းတွေအတွက် ကမ္ဘာဖိုင်တွေကို ပရောဂျက် တစ်ခုချင်းအလိုက် သီးခြား worlds ဖိုဒါထဲမှာ ထည့်ပေးထားမှာပါ။ ပရိုဂါရမ်တစ်ခုဟာ ကမ္ဘာတစ်ခုတည်းမှာပဲ အလုပ်လုပ်တာမဟုတ်ဘဲ အလားတူ ကမ္ဘာအမျိုးမျိုးအတွက် အလုပ်လုပ်အောင် ရေးပေးရတာ။ အခြားတာကို နားမလည်ရင် အခန်း (၂) ဖတ်ပြီးရင် နားလည်သွားမှာပါ။

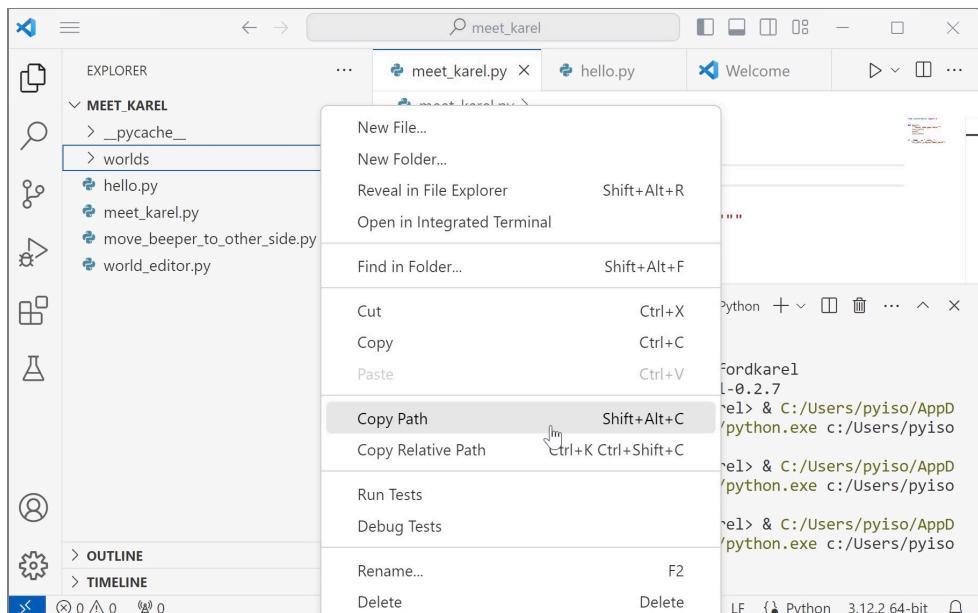
**Load World** လုပ်တဲ့အခါ ပွင့်လာတဲ့ ဖိုင် ခိုင်ယာလော်ကာ ကိုယ်လိုချင်တဲ့ worlds ဖိုဒါ မဖြစ်နေဘူး။ သူနိုင်ပါတဲ့ worlds ဖိုဒါ ဖြစ်နေတယ်။ ကိုယ်ခေါ်တင်ချင်တဲ့ ဖိုင်တွေရှိတဲ့ လက်ရှိပရောဂျက်၏ worlds ဖိုဒါကို သွားရမယ်။ ဥပမာ PyCharm/VS Code အတွက် MeetKarel/meet\_karel ပရောဂျက် worlds ဖိုဒါ လမ်းကြောင်း အပြည့်အစုံက

C:\Users\yourname\VSCode\meet\_karel\worlds

C:\Users\yourname\PycharmProjects\MeetKarel\worlds

ဖြစ်ပါမယ်။ ဖိုင်ခိုင်ယာလော်ကနေ ဖော်ပြပါ လက်ရှိပရောဂျက် worlds ဖိုဒါကို တစ်ဆင့်ချင်း သွားပြီး တင်ချင်တဲ့ ကမ္ဘာဖိုင် (.w ဖိုင်) ကို ရွေးရမှာပါ။

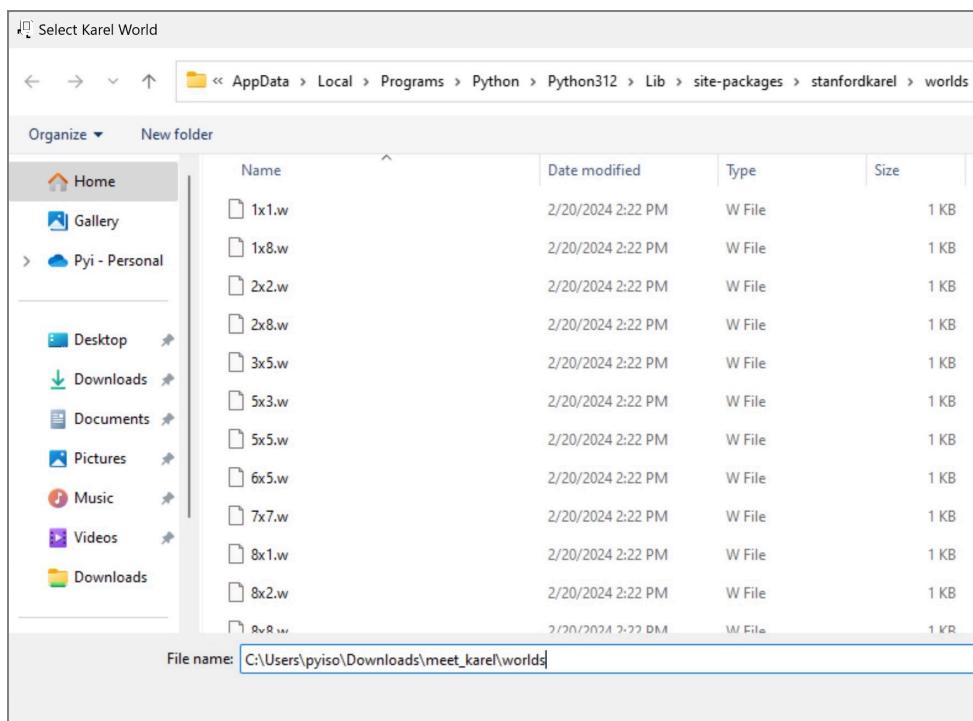
အပေါကနည်းနဲ့ အဆင်မပြောရင် အခုလိုစမ်းကြည့်ပါ။ လက်ရှိပရောဂျက် worlds ဖိုဒါကို ညာကလစ်နှုပ်ပြီး Copy Path လုပ်ပါ (ပုံ ၁/၂)။ ဖိုင် ခိုင်ယာလော် **File name** မှာ ကူးထည့်ပါ (ပုံ ၂/၃)။ **Enter** ကိုနိုပ်ပါ။ ပရောဂျက် worlds ဖိုဒါကိုရောက်သွားပါမယ်။ လိုတဲ့ကမ္ဘာဖိုင် ရွေးတင်ရှုပါပဲ။ ပုံ (၁/၄) မှာ meet\_karel worlds ကို နှုန္တပြထားပါတယ်။



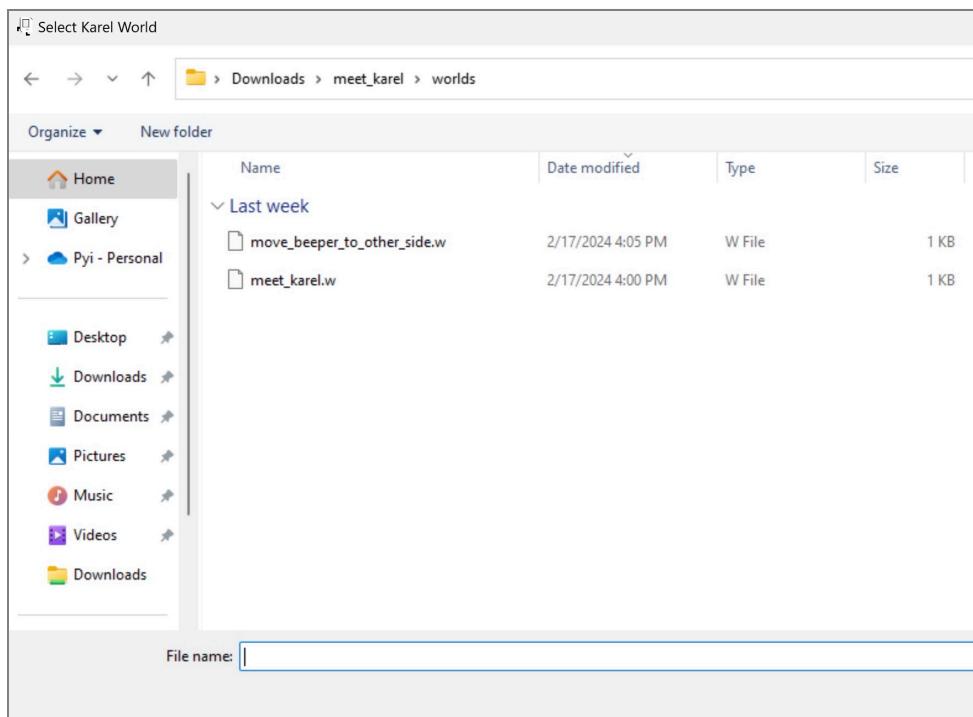
ပုံ ၁/၂

အကယ်၍ အထက်ဖော်ပြပါနည်းတွေက ရှုပ်နေတယ်ထင်ရင် မှတ်ရ/သွားရ လွယ်တဲ့ Desktop, Downloads, Documents လို နေရာတစ်ခွေမှာ သီးသန့်ဖိုဒါတစ်ခု ဆောက်ပြီး ပရောဂျက်အားလုံး ထည့်ထားတာ အရှင်းဆုံးပါပဲ။ ပရောဂျက်ဖိုဒါနေရာ သိရင် ဖိုင်ခိုင်ယာလော်ကနေ ဘယ်လိုမဆိုရောက်အောင် သွားလိုပါတယ်။

၂၁



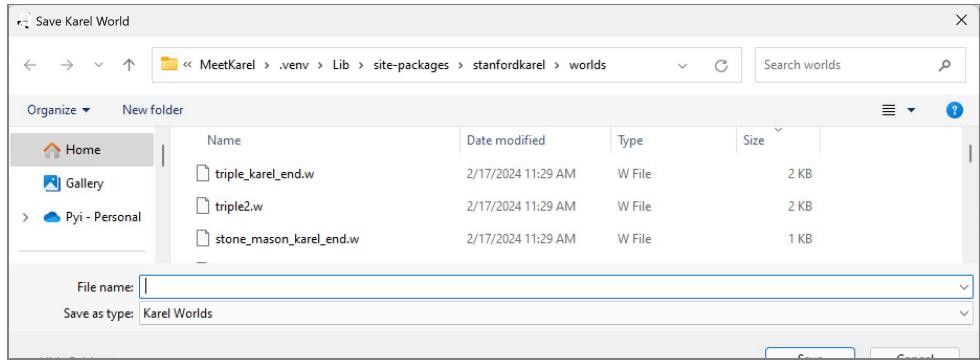
ပုံ ၅/၃



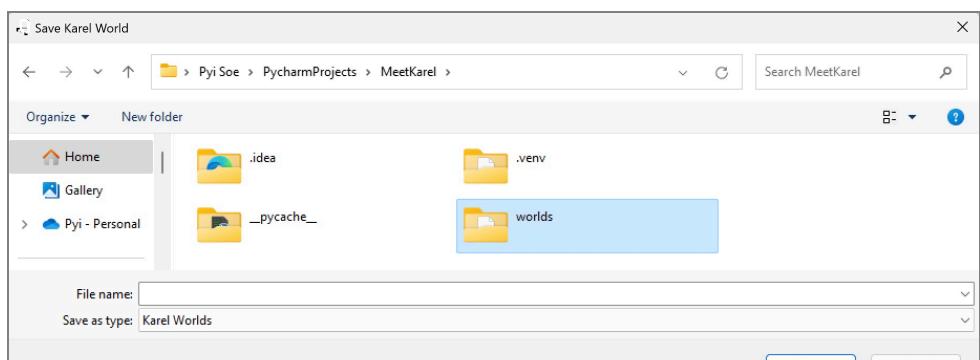
ပုံ ၆/၄

## ကိုယ်ပိုင် ကားရဲလ်ကမ္ဘာ ဆောက်ခြင်း

ကားရဲလ်ရဲ ကမ္ဘာ အသစ်တစ်ခုဆောက်မယ်ဆိုရင် world\_editor.py ဖိုင်ကို ညာကဗျာစွဲနှင့် Run ပါ။ Would you like to load an existing world? လို ပေါ်လာပြီး Yes/No ရွှေးချိုင်ပါလိမ့်မယ်။ No ကို နှိပ်ပါ။ ကမ္ဘာအရွယ်အစားအတွက် ကော်လံ ဘယ်နှစ်ခုလဲ row ဘယ်နှစ်ခုလဲ ထည့်ပေးပါ။ ကိုယ်ပိုင် ကားရဲလ်ကမ္ဘာ တည်ဆောက်လိုရတဲ့ ဝင်းဒီးပွင့်လာပါလိမ့်မယ်။ ကားရဲလ် မျက်နှာမူရာအရပ်၊ ဘိပါအိတ်ထဲရှိ ဘိပါအရေအတွက်၊ နံရုံဆောက်/ဖျက်တာ၊ ဘိပါထည့်/ဖယ်ထုတ်တာ စတာတွေ လုပ်နိုင်ပါတယ်။ Save World နှိပ်ပြီး သိမ်းခိုင်ပါတယ် ဖိုင်ကိုသိမ်းတဲ့အခါ သုန္ဓိ သိမ်းခိုင်းတဲ့ဖို့ (default world folder) ထဲမှာ သို့မဟုတ် ပင်မ ပရောဂျက်ဖို့တဲ့က worlds ဖို့ဒါထဲမှာ သိမ်းရပါမယ်။



ပုံ ၉/၅



ပုံ ၉/၆



Python programming language

**import, from, def, if**

New term

#### fEnEmpstatement

ပါရာမီတာမပါရင်လည်း စိုက်ကွင်းအလွတ် တစ်ဖုံး ‘()’ တော့ပါရမယ်။

colon ‘:’

ဖန်ရှင်နဲ့ သက်ဆိုင်တဲ့ ကုဒ်ဘလောက် (*code block*)

quote သုံးခုတွဲ ‘“”’

ပရိုဂရမ်ဝင်းဒီးမှာ **Run Program** ခလုတ်

**meet\_karel.w** ကမ္မာကို