Begin Modern Programming with Python

Pyi Soe

မာတိကာ

၁ စက်ရုပ်ကားရဲလ်ဖြင့် ပရိုဂရမ်းမင်းမိတ်ဆက် ၁

- ၁.၁ စက်ရုပ် ကားရဲလ် ၁
- ၁.၂ Meet Karel ပရိုဂရမ် ၃
- ၁.၃ ကားရဲလ် ပရိုဂရမ် run ခြင်း ၈
- ၁.၄ Python အင်စတောလ်လုပ်တဲ့အခါ ဘာတွေပါလဲ ၉
- ၁.၅ Move Beeper to Other Side ၁၁

၂ ကားရဲလ်နှင့် ကွန်ထရိုးလ် စတိတ်မန့်များ ၁၃

- ၂.၁ **for** loop ၁၃
- ၂.၂ အင်ဒန့်ထ်လုပ်ခြင်းနှင့် ကုဒ်စထရက်ချာ ၁၆
- ၂.၃ **while** loop ၁၆
- ၂.၄ **if** စတိတ်မန့် ၁၉
- ၂.၅ **if...else** စတိတ်မန့် ၂၁
- ၂.၆ Nested ကွန်ထရိုးလ် စတိတ်မန့်များ ၂၂

၃ ဖန်ရှင်များ (Functions) ၂၉

- ၃.၁ ဖန်ရှင် သတ်မှတ်ခြင်း ၂၉
- ၃.၂ ပရီကွန်ဒီရှင်နှင့် ပို့စ်ကွန်ဒီရှင် ၃၂
- ၃.၃ ဖန်ရှင်များဖြင့် abstraction များ တည်ဆောက်ခြင်း ၃၄
- ၃.၄ Bottom-up Programming ၃၅
- ၃.၅ Top-down Programming ၄၃

၄ ကားရဲလ်နှင့် ရီကားဆစ်ဖ် ဖန်ရှင်များ ၅၁

- ၄.၁ ရီကားဆစ်ဖ် ဖန်ရှင် ဘယ်လို အလုပ်လုပ်လဲ ၅၁
- ၄.၂ ရီကားဆစ်ဖ်နည်းဖြင့် ပုဏ္ဏာဖြေရှင်းခြင်း ၅၄

၅ ဒေတာများနှင့် ဖန်ရှင်များ ၆၅

- ၅.၁ ကိန်းဂဏန်းများ ၆၅

၅.၂ 'တိုက်ပ်' ဆိုတာ ဘာလဲ ၃၂
၅.၃ ဗေရီရေဘဲလ်များ ၃၃
၅.၄ စာသားများ ၃၄
၅.၅ အိပ်စ်ပရက်ရှင်များ ၃၈
၅.၆ ဘူလီယန် တိုက်ပ်နှင့် ဘူလီယန် အိပ်စ်ပရက်ရှင် ၃၉

၆ အော့ဘ်ဂျက်များ ၈၅

၆.၁ **date**, **time** and **datetime** ၈၆
၆.၂ **list** ၉၀

၇ ကွန်ထရိုးလ် စတိတ်မန့်များ ၉၃

၇.၁ **if** စတိတ်မန့် ၉၃
၇.၂ **for** Loop ၉၈
၇.၃ Python Arcade Library ၁၀၅
၇.၄ **while** loop ၁၀၉
၇.၅ လေ့ကျင့်ရန် ဥပမာများ ၁၁၄

၈ ဖန်ရှင်များ ၁၂၁

၈.၁ တန်ဖိုးပြန်ပေးတဲ့ ဖန်ရှင်များ ၁၂၁
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" ၁၄၃
၈.၆ ပါရာမီတာ ရွေးချယ်သတ်မှတ်ခြင်း ၁၅၂
၈.၇ တန်ဖိုး ပြန်ပေးခြင်း၊ မပေးခြင်း ၁၅၄
၈.၈ **return** ၁၅၆

၉ From Classes to Objects (ကလပ်စ်များမှ အော့ဘ်ဂျက်များသို့) ၁၅၉

၉.၁ **Account** Class (အကောင့် ကလပ်စ်) ၁၅၉

က လိုအပ်သည့် ဆော့ဖ်ဝဲများ ထည့်သွင်းခြင်း ၁၆၅

ခ ကားရဲလ်ပရိုဂရမ် ဖီချာများ ၁၉၁

အခန်း ၁

စက်ရုပ်ကားရဲလ်ဖြင့် ပရိုဂရမ်းမင်းမိတ်ဆက်

ကွန်ပျူတာတွေဟာ သက်မဲ့ စက်ပစ္စည်းတွေပါပဲ။ ကားတို့၊ လေယာဉ်တို့နဲ့ မတူတာက ကွန်ပျူတာတွေဟာ စက်ချည်းသက်သက် ဘာအစွမ်းမှ မယ်မယ်ရရ မရှိဘူး။ ဒါပေမဲ့ ဆောင်ရွက်လိုတဲ့ ကိစ္စအဝဝအတွက် ပရိုဂရမ်အမျိုးမျိုး ထည့်ပေးလိုက်တဲ့အခါမှာ သူ့ရဲ့အစွမ်းက အတိုင်းအဆမဲ့ပဲ။ နေရာမျိုးစုံ၊ နယ်ပါယ်မျိုးစုံမှာ အကူအညီပေးနိုင်တဲ့ စွယ်စုံသုံး ပစ္စည်းတစ်ခုဖြစ်သွားတယ်။ ဂီတသံစဉ်တွေကို ဖွင့်ပေးနိုင်သလို အသံလည်းသွင်းပေးနိုင်တယ်။ ရုပ်ရှင်တည်းဖြတ် လုပ်ချင်တာလား။ ပြဿနာမရှိဘူး၊ ကူညီပေးနိုင်တယ်။ နျူကလီးယား ဓါတ်ပေါင်းဖိုတွေကို စီမံနိုင်သလို မောင်းသူမဲ့ ဒုံးပျံတွေကိုလည်း ပဲ့ထိန်းပေးနိုင်တယ်။

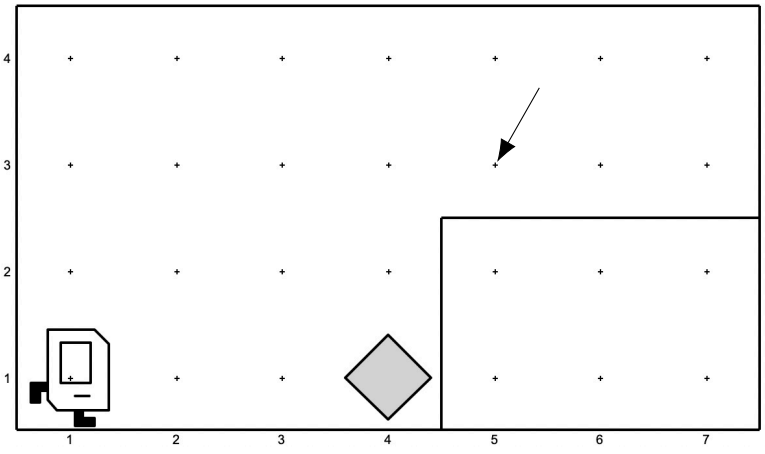
ကျွန်တော်တို့တွေ နိစ္စဓူဝ အသုံးပြုနေကြတဲ့ ကား၊ စမတ်ဖုန်း၊ လက်ပါတ်နာရီ၊ မိုက်ခရိုဝေဖ်မီးဖို၊ အဝတ်လျှော်စက် စတဲ့ စက်ပစ္စည်း အမျိုးမျိုးဟာလည်း ကွန်ပျူတာတွေနဲ့ မကင်းပြန်ပါဘူး။ "ကွန်ပျူတာ နည်းပညာ အကူအညီမပါတဲ့ ခေတ်မီဆန်းသစ်တီထွင်မှုဆိုတာ မရှိဘူး" လို့ ဆိုနိုင်ပါတယ်။

တစ်ချက်တစ်ချက် ရိုက်ခတ်လိုက်တဲ့ ကွန်ပျူတာနည်းပညာ လှိုင်းလုံးကြီးတွေဟာ ကမ္ဘာတစ်ဝှမ်းလုံး ပုံစံပြောင်းသွားလောက်အောင် အဟုန်ပြင်းထန်လှတယ်။ ဘီလီယံနဲ့ချီတဲ့ လူတွေ ဆိုရှယ်မီဒီယာတွေပေါ်ကနေ ရုပ်သံတွေနဲ့ ချိတ်ဆက်ပြောဆိုဆက်သွယ်လို့ ရစေတာဟာလည်း ကွန်ပျူတာစနစ်တွေပါပဲ။ Artificial Intelligence (AI) နည်းပညာကြောင့် သက်ရှိတွေမှာပဲတွေ့ရတဲ့ ညာဏ်ရည်မျိုးကို ကွန်ပျူတာတွေမှာလည်း တွေ့လာရပါပြီ။ သန်ဘုဏ္ဍာတွေ ဖြေရှင်းခြင်း၊ စစ်တုရင်ထိုးခြင်း စတဲ့ကိစ္စမျိုးတွေအပြင် ပန်းချီဆွဲခြင်း၊ ကဗျာရေးစပ်ခြင်း၊ သီချင်းရေးဖွဲ့ခြင်း ကဲ့သို့ အနုပညာဖန်တီးမှုတွေကိုပါ AI က လုပ်ဆောင်ပေးနိုင်ပါတယ်။ နှစ်ဆယ့်တစ်ရာစုရဲ့ အထူးခြားဆုံး AI နည်းပညာလှိုင်းဟာ အရှိန်အဟုန်ပြင်းပြင်း ရိုက်ခတ်ဖို့ အားယူစ ပြုနေပါပြီ။

'ကွန်ပျူတာ' လို့ပြောတဲ့အခါ စက်ပစ္စည်းသက်သက် မဟုတ်ဘဲ ကွန်ပျူတာမှတ်ညာဏ်ထဲက ပရိုဂရမ်တွေလည်း ပါဝင်တယ်ဆိုတာ သတိချပ်ရပါမယ်။ ကွန်ပျူတာတွေ တစ်စုံတစ်ရာ စွမ်းဆောင်နိုင်စေတဲ့ ပရိုဂရမ်တွေ ရေးတဲ့အလုပ်ကို ပရိုဂရမ်းမင်း (Programming) လို့ခေါ်တယ်။

၁.၁ စက်ရုပ် ကားရဲလ်

ပရိုဂရမ်းမင်းဆိုတာ ဘယ်လိုမျိုးလဲ သဘောပေါက်အောင် စာတွေတစ်သီကြီးရေး ရှင်းပြတာထက် ပရိုဂရမ်လေးတွေ လက်တွေ့ ရေးကြည့်လိုက်တာ ပိုပြီးထိရောက်ပါတယ်။ ဒါကြောင့် စက်ရုပ်ကားရဲလ်ကို ပရိုဂရမ်လေးတွေရေးပြီး အလုပ်တွေခိုင်းကြည့်ကြမယ်။ ပုံ (၁.၁) မှာ တွေ့ရတာက ကားရဲလ် ရောက်ရှိနေတဲ့ နမူနာကမ္ဘာတစ်ခုပါ။ မီးခိုးရောင် မှန်ကူကွက်ပုံလေးကို ဘီပါ (beeper) လို့ ခေါ်တယ်။ အဲဒီဘီပါကို မြှားပြထားတဲ့ နေရာကို ရွှေ့ခိုင်းချင်တယ်။ မျဉ်းမည်းအထူတွေက နံရံတွေပါ။ ကားရဲလ်ကို ကိစ္စတစ်ခု ဆောင်ရွက်စေ



ပုံ ၁.၁ စက်ရုပ်လေး ကားရဲလ်

ချင်တဲ့အခါ အခြေခံ ကားရဲလ်ကွန်မန်းတွေကို အသုံးပြုရပါတယ်။ ကွန်မန်းတွေကို နူတ်နဲ့ပြောပြီး ခိုင်းရတာ မဟုတ်ဘဲ ပရိုဂရမ်ရေးပြီး 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 ပရိုဂရမ်းမင်း စလေ့လာဖို့ ဒီလေးခုနဲ့ပဲ လုံလောက်ပါပြီ။

`move` ကွန်မန်းက ကားရဲလ်ကို ရှေ့တစ်ကွန်နာကို ရွှေ့ခိုင်းတာ။ ကားရဲလ်ကမ္ဘာထဲမှာ တစ်ခုနဲ့တစ်ခု အကွာအဝေးတူ ခြားထားတဲ့ အတန်းလိုက် အတန်းလိုက် အစက်ကလေးတွေဟာ ကွန်နာ (corner) တွေ ဖြစ်တယ်။ ကမ္ဘာကို မျဉ်းမည်းအထူ နံရံတွေနဲ့ ထောင့်မှန်စတုဂံပုံ ပါတ်လည် ဘောင်ခတ်ထားတယ်။ ကွန်နာတွေကြားမှာလည်း နံရံတွေရှိနိုင်တယ်။ နမူနာကမ္ဘာမှာ ဘေးတိုက် နံပါတ်စဉ် ၄ နဲ့ ၅ ကြား ထောင်လိုက် နံရံတစ်ခု၊ အထက်အောက် နံပါတ်စဉ် ၂ နဲ့ ၃ ကြား အလျားလိုက် နံရံတစ်ခုကို တွေ့ရပါမယ်။ ကွန်နာရှေ့မှာ နံရံကာနေရင် ကားရဲလ်ကို `move` ခိုင်းလို့မရပါဘူး။

`put_beeper` က ကားရဲလ် လက်ရှိ ရှိနေတဲ့ ကွန်နာမှာ 'ဘီပါတစ်ခုချ' ထားခိုင်းတာ၊ `pick_beeper` က ရပ်နေတဲ့ ကွန်နာမှာ 'ဘီပါတစ်ခုကောက်' ခိုင်းတာပါ။ ကွန်နာမှာ ဘီပါရှိနေမှ ကောက်ခိုင်းလို့ရမှာပါ။ မရှိရင် ကောက်ခိုင်းလို့ မရဘူး။ ဘီပါချခိုင်းရင်လည်း ကားရဲလ်မှာ ဘီပါရှိမှ ချခိုင်းလို့ရတယ်။ ကားရဲလ်ကို ဘီပါတွေ လိုသလောက် ဖြည့်ပေးထားတယ်လို့ ယူဆပါ။ `turn_left` က 'ဘယ်လှည့်' ခိုင်းတာ။

ဘီပါကို ဘယ်လိုရွှေ့ခိုင်းမလဲ

ပုံ (၁.၁) အနေအထားကနေ ရှေ့ကို သုံးနေရာရွှေ့၊ ဘီပါကောက်၊ ဘယ်ဘက်လှည့်၊ အပေါ် နှစ်နေရာရွှေ့၊ ညာဘက်လှည့်၊ ရှေ့တစ်နေရာထပ်ရွှေ့ပြီး ဘီပါချထားခိုင်းလိုက်ရင် အလုပ်ပြီးသွားပါပြီ။

ကားရဲလ်ကို ညာဘက်လှည့်ခိုင်းဖို့ `turn_right` ကွန်မန်း မရှိဘူး။ ဒါပေမဲ့ ဘယ်သုံးခါလှည့်တာဟာ ညာဘက်လှည့်တာနဲ့ တူတူပါပဲ။ ဒါကြောင့် ညာဘက်ချင်တဲ့အခါ ဘယ်သုံးခါလှည့်ခိုင်းလို့ရတယ်။

၁.၂ Meet Karel ပရိုဂရမ်

ပရိုဂရမ် ရေးတယ်ဆိုတာ ကွန်ပျူတာကို ကိစ္စတစ်ခုခု ဆောက်ရွက်ပေးဖို့ ခိုင်းစေတဲ့ ညွှန်ကြားချက်တွေ ရေးတာပါပဲ။ ဒီလို ညွှန်ကြားချက်တွေကို ပရိုဂရမ်ကုဒ် (*program code*) လို့ ခေါ်တယ်။ ပရိုဂရမ်ကုဒ်တွေကို ကွန်ပျူတာနားလည်တဲ့ programming language တစ်ခုခုနဲ့ ရေးရတယ်။ ဒီစာအုပ်မှာ အသုံးပြုမဲ့ programming language ကတော့ Python ပါ။ Programming language တစ်မျိုးပဲ ရှိတာ မဟုတ်ပါဘူး။ ရာနဲ့ချီပြီး ရှိတာပါ။ လူ့ဘာသာစကားတွေ အမျိုးမျိုးရှိသလိုပဲပေါ့။ Python ဟာ ဒီလို ရာနဲ့ချီတဲ့ထဲက လက်ရှိအသုံးအများဆုံး ထိပ်ဆုံးဆယ်ခု ထဲမှာ ပါဝင်တယ်။ Python နဲ့ ဘိပါရွှေခိုင်းတဲ့ ပရိုဂရမ်ကို လေ့လာကြည့်ရအောင်။ ကားရဲလ်နဲ့ ပထမဆုံး မိတ်ဆက်ပေးတဲ့ ပရိုဂရမ်မို့လို့ ဒီပရိုဂရမ် နံမည်ကို 'Meet Karel' လို့ 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' ပရိုဂရမ်အတွက် Python နဲ့ရေးထားတဲ့ ပရိုဂရမ်ကုဒ် တွေဖြစ်ပါတယ်။ 'Python ကုဒ်' လို့ အတိုချုံးပဲ ပြောတာများတယ်။ Python 'စာ/စကား' မတတ်ရင် ဒီ 'Python ကုဒ်' တွေကိုလည်း နားလည်မှာ မဟုတ်ဘူး။ ဒီတော့ Python 'စာ/စကား' အခြေခံက စပြီး လေ့လာဖို့ လိုပါမယ်။

ကွန်းမန့် (Comment)

ပထမဆုံး # သင်္ကေတနဲ့ စတဲ့ စာကြောင်းတွေက ကွန်းမန့်တွေပါ။ ကွန်းမန့်တွေက ကွန်ပျူတာ ဆောင်ရွက်ပေးရမဲ့ ညွှန်ကြားချက်တွေ မဟုတ်ဘူး။ ပရိုဂရမ်ကုဒ်နဲ့ ပါတ်သက်ပြီး ကုဒ် ဖတ်ရှုသူ အတွက် မှတ်ချက်

ရေးတာ သို့မဟုတ် ရှင်းပြထားတာပါ။ တနည်းအားဖြင့် ဖတ်ရှုသူ (လူ) ပရိုဂရမ်မာအတွက် ရည်ရွယ်တာ။ ကွန်ပျူတာ (စက်) အတွက် ရည်ရွယ်တာ မဟုတ်ဘူး။ ကွန်ပျူတာက ကုဒ်ထဲက ကွန်းမန့်တွေ အားလုံးကို လစ်လျူရှုမှာပါ။ ဒါပေမဲ့ ပရိုဂရမ်ကုဒ်ကို ဖတ်တဲ့လူ နားလည်ဖို့ အထောက်အကူ ဖြစ်စေတဲ့အတွက် ကွန်းမန့်ရေးတာကို ပေါ့ပေါ့တန်တန် အရေးမပါသလို သဘောထားလို့ မရပါဘူး။ မိမိရေးတဲ့ ကုဒ်ကို ရှင်းပြဖို့ လိုအပ်ရင် ကွန်းမန့်ရေးရပါမယ်။ ရေးသင့်တဲ့ နေရာတွေကိုလည်း မကြာခင်တွေ့ရမှာပါ။

import စတိတ်မန့်

```
from stanfordkarel import *
```

ကတော့ အင်ပို့စတိတ်မန့် ဖြစ်ပါတယ်။ "stanfordkarel လိုက်ဘရီမှ အာလုံးကို ထည့်သွင်းပေးပါ" လို့ တောင်းဆိုတဲ့ အဓိပ္ပါယ်။ * သင်္ကေတကို 'အားလုံး' လို့ ယူဆပါ။ stanfordkarel လိုက်ဘရီမှာ ကားရဲလ်ပရိုဂရမ်အတွက် လိုအပ်တာအားလုံး ပါဝင်တယ်။ ဒီလိုက်ဘရီကို အင်ပို့လုပ်ထားမှ ကားရဲလ်ကွန်မန်းတွေ သုံးလို့ရမှာပါ။ သီးခြား ကားရဲလ်ပရိုဂရမ် တစ်ခုစီတိုင်းအတွက် အင်ပို့လုပ်ရမှာ ဖြစ်တယ်။

လိုက်ဘရီ

လိုက်ဘရီ (*library*) ဆိုတာ ပညာရပ်နယ်ပယ် တစ်ခုအတွက် ရည်ရွယ်ရေးထားတဲ့ ပရိုဂရမ်ကုဒ်တွေပါပဲ။ သင်္ချာအတွက်အချက် လိုက်ဘရီ၊ ဂိမ်းရေးဖို့ လိုက်ဘရီ၊ 2D/3D ဂရပ်ဖစ်ဆွဲဖို့ လိုက်ဘရီ၊ အေအိုင်အတွက် လိုက်ဘရီ စသည်ဖြင့် နယ်ပယ်အသီးသီး၊ ကိစ္စရပ်အဖုံဖုံအတွက် သက်ဆိုင်ရာ ကျွမ်းကျင်ပညာရှင်တွေ ထုတ်လုပ်ဖြန့်ချိပေးထားတဲ့ လိုက်ဘရီတွေရှိတယ်။ Matrix အော်ပရေးရှင်းတွေအတွက် numpy ၊ ဂရပ်ဖစ်ဆွဲမယ်ဆိုရင် matplotlib စတဲ့ လိုက်ဘရီတွေကို အင်ပို့လုပ် အသုံးပြုနိုင်ပါတယ်။ မေ့ထရစ် A ကို B နဲ့ မြှောက်ရင် 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)
```

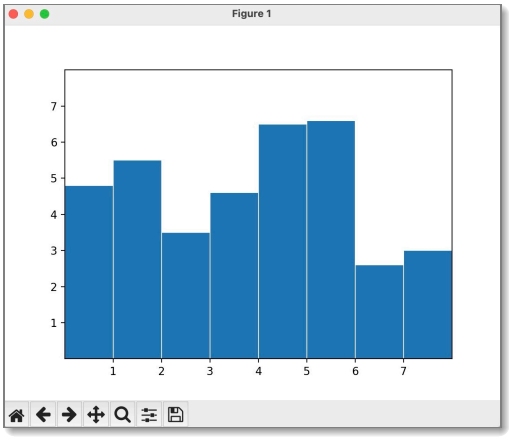
ရလဒ် အခုလိုထွက်ပါမယ်။

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

ဒါကတော့ ဘားချတ် 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()
```

ဘားချတ်ကို ဒီလို ထုတ်ပေးပါတယ်။



ပုံ ၁.၂

လိုက်ဘရီတွေဟာ ပရိုဂရမ်တွေ တည်ဆောက်ရာမှာ အင်မတန်မှ အရေးပါတယ်။ ဖော်ပြထားတဲ့ မေ့ထရစ် နဲ့ ဘားချတ် ကုဒ်တွေကို (အခုတော့) နားလည်မှာ မဟုတ်သေးဘူးပေါ့။ ဒါပေမဲ့ သက်ဆိုင်ရာ လိုက်ဘရီတွေနဲ့ ဒီလိုကိစ္စတွေကို သိပ်မခက်ခဲဘဲ လုပ်လို့ရနိုင်တယ်ဆိုတာ မြင်မယ် ထင်ပါတယ်။ လိုက်ဘရီတွေ သာမရှိရင် ပရိုဂရမ်တွေကို အခုထက် အဆပေါင်းများစွာ အချိန်ပေးပြီး ရှုပ်ရှုပ်ထွေးထွေး ခက်ခက်ခဲခဲ တည်ဆောက်ကြရမှာပါ။

တုံကင်၊ စတိတ်မန့်နှင့် ဆင်းတက်စ်

ဂျန်ပန်စာ၊ ပြင်သစ်စာ စတဲ့ လူ့ဘာသာစကားတွေဟာ စကားလုံးတွေ ဝါကျတွေနဲ့ ဖွဲ့စည်းထားသလို ပရိုဂရမ်ကုဒ်တွေဟာလည်း စကားလုံးတွေ၊ ဝါကျတွေနဲ့ ဖွဲ့စည်းထားတာပါပဲ။ Programming language တွေမှာ စကားလုံးတွေကို တုံကင် (*token*) လို့ခေါ်ပြီး ဝါကျတွေကိုတော့ စတိတ်မန့် (*statement*) လို့ ခေါ်ပါတယ်။ ဝါကျတွေကို စကားလုံးတွေနဲ့ ဖွဲ့စည်းထားသလို စတိတ်မန့်တွေကတော့ တုံကင်တွေနဲ့

ဖွဲ့စည်းထားတာပါ။ စတိတ်မန့် ပုံစံတစ်မျိုးကို တွေ့ခဲ့ပြီးပါပြီ။ အဲဒါကတော့ ရှေ့စာမျက်နှာက အင်ပို့စတိတ်မန့်ပဲ ဖြစ်ပါတယ်။

လူ့ဘာသာစကားတွေမှာ ဖွဲ့စည်းတည်ဆောက်ပုံ စထရက်ချာရှိသလို programming language တွေမှာလည်း စထရက်ချာရှိဖို့ လိုအပ်တာပေါ့။ ဖွဲ့စည်းပုံ စထရက်ချာ မှန်/မမှန်ကို သဒ္ဒါစည်းမျဉ်တွေနဲ့ ထိန်းကွပ်ထားတာပါ။ ပရိုဂရမ်ကုဒ် စထရက်ချာ မှန်/မမှန် ထိန်းကွပ်ပေးတဲ့ သဒ္ဒါစည်းမျဉ်တွေကိုတော့ ဆင်းတက်စ် (*syntax*) လို့ခေါ်တယ်။

မြန်မာလိုရေးရင် မြန်မာသဒ္ဒါကို လိုက်နာရသလို Python နဲ့ ရေးရင် Python ဆင်းတက်စ်ကို လိုက်နာရမှာပေါ့။ မြန်မာသဒ္ဒါမှားရင် ဖတ်တဲ့သူက သည်းခံနားလည် ပေးပေမဲ့ ဆင်းတက်စ်မှားရင်တော့ Python က လုံးဝ လက်ခံမှာ မဟုတ်ပါဘူး။ ဆင်းတက်စ် စည်းကမ်းတွေဟာ ပိုပြီး တင်းကျပ်တယ်။ လွဲချော်လို့ မရဘူး။ ဆင်းတက်စ်မှားနေတဲ့ ပရိုဂရမ်ကို Python က run ခွင့် ပြုမှာမဟုတ်ဘဲ အမှားနဲ့ သက်ဆိုင်တဲ့ အယ်ရာမက်ဆေ့ချ်တွေ ပြပေးမှာပါ။ ဖြစ်လေ့ရှိတဲ့ ဆင်းတက်စ်အမှားတွေကို မကြာခင် တွေ့ရပါမယ်။

Keywords

`from`, `import`, `def` , `if` စတာတွေဟာ *keyword* တွေဖြစ်တယ်။ Python ရေးတဲ့အခါ သူ့နေရာနဲ့ သူ အဓိပ္ပါယ်ကိုယ်စီနဲ့ အသုံးပြုရတဲ့ စကားလုံးတွေဖြစ်တယ်။ `from` နဲ့ `import` ကို လိုက်ဘရီ အင်ပို့လုပ်ဖို့ သုံးတယ်။ `def` ကို ဖန်ရှင် သတ်မှတ်တဲ့အခါ သုံးတယ်။ Python က သတ်မှတ်ထားတဲ့ နေရာတွေက လွဲလို့ အခြားကိစ္စတွေအတွက် keyword တွေကို အသုံးပြုလို့ မရပါဘူး။ ဒါကြောင့် keyword တွေကို reserved word လို့လည်း ခေါ်တယ်။

`main` ဖန်ရှင်

'Meet Karel' ပရိုဂရမ် အင်ပို့စတိတ်မန့် အပြီးမှာ တွေ့ရတာကတော့ `main` ဖန်ရှင်သတ်မှတ်ချက်ပါ။ ကြည့်ရအဆင်ပြေအောင် သူ့ချည်းသီးသန့် 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
```

ဖန်ရှင် (*function*) ဆိုတာ ကိစ္စတစ်ခု လုပ်ဆောင်ပေးဖို့အတွက် ယူနစ်တစ်ခုအနေနဲ့ ဖွဲ့စည်းထားတဲ့ ပရိုဂရမ်ကုဒ် အစုအဝေးတစ်ခုပါပဲ။ ဖန်ရှင်ကို အသုံးပြုတဲ့အခါ ၎င်းရဲ့ လုပ်ငန်းတာဝန်အတိုင်း ဖန်ရှင်က လုပ်ဆောင်ပေးမှာ ဖြစ်တယ်။ ဖန်ရှင်အသုံးပြုတာကို 'ဖန်ရှင်ကောလ်' (*function call*) လုပ်တာလို့ ပြောတယ်။

`main` ဖန်ရှင်သတ်မှတ်ချက်ကို အပိုင်းနှစ်ပိုင်းခွဲ ကြည့်နိုင်တယ်။ ပထမတစ်ပိုင်း

```
def main():
```

ကို ဖန်ရှင်ခေါင်းစီး (function header) လို့ခေါ်တယ်။ ဖန်ရှင်ခေါင်းစီးမှာ ဖန်ရှင်နံမည်နဲ့ ဖန်ရှင်ပါရာမီတာတွေကို ဝိုက်ကွင်းထဲမှာ သတ်မှတ်ပေးရပြီး colon ':' နဲ့ အဆုံးသတ်တယ်။ ဥပမာ x, y ပါရာမီတာ နဲ့ `myfun` ဖန်ရှင် အတွက်

```
def myfun(x, y):
```

ပါရာမီတာမပါရင်လည်း ဝိုက်ကွင်းအလွတ် တစ်စုံ '()' တော့ပါရမယ်။ `main` ဖန်ရှင်မှာ ပါရာမီတာ မပါဘူး။ ပါရာမီတာတွေအကြောင်း နောက်ပိုင်းအခန်းတွေမှာ အသေးစိတ် လေ့လာရမှာပါ။ ကားရဲလ်ပရိုဂရမ်တွေမှာ ပါရာမီတာအကြောင်း သိဖို့မလိုသေးပါဘူး။ ပါရာမီတာ မလိုတဲ့ ဖန်ရှင်တွေပဲ တွေ့ရမှာပါ။

ဖန်ရှင်ခေါင်းစည်းအောက် ဒုတိယပိုင်းကတော့ `main` ဖန်ရှင် ကုဒ်ဘလောက် (*code block*) ပါ။ ကုဒ်ဘလောက် ဆိုတာ ကုဒ်တွေကို အုပ်စုတစ်စု အဖြစ် ဖွဲ့စည်းထားတာကို ဆိုလိုတာပါ။ ဘလောက်လို့လည်း ခေါ်တယ်။ ဘလောက်တစ်ခုမှာ ပါဝင်တဲ့ ကုဒ်လိုင်းတွေကို ညာဘက် ခွာရေးရပါမယ်။ အင်ဒန့်ထ် (indent) လုပ်တာလို့ ခေါ်တယ်။ တက်ဘ်ကီးတစ်ချက် (သို့) စပေ့စ်လေးခုစာ ခွာလေ့ရှိတယ်။ ဘော်ဒီ ပထမတစ်ကြောင်း

```
"""Karel code goes here!"""
```

ကို *docstring* လို့ ခေါ်တယ်။ quote သုံးခုတွဲ '"""' တစ်စုံကြား ညှပ်ရေးတဲ့ ကွန်းမန့်တစ်မျိုးလို့ ယူဆနိုင်တယ်။ ဖန်ရှင်နဲ့ ပါတ်သက်တဲ့ ရှင်းလင်းဖော်ပြချက်တွေ ရေးဖို့အတွက် သုံးတာပါ။

Docstring အောက်မှာ တွေ့ရတာကတော့ ကားရဲလ်ကွန်မန်းတွေဆိုတာ သိပါလိမ့်မယ်။ ကားရဲလ်ကွန်းမန်းတွေဟာ `stanfordkarel` လိုက်ဘရီ ဖန်ရှင်တွေပါ။ တနည်းအားဖြင့် `stanfordkarel` လိုက်ဘရီမှာ ကားရဲလ်ကွန်းမန်းတွေအတွက် ဖန်ရှင်သတ်မှတ်ချက်တွေ ပါဝင်တယ်။ ဖန်ရှင်တစ်ခုကို အသုံးပြုဖို့ အတွက် အဲဒီဖန်ရှင်ကို ခေါ်ရပါတယ်။ ဒါကို ဖန်ရှင်ကောလ် (*function call*) လုပ်တယ်လို့ ပြောတယ်။ ကားရဲလ်ကို ဘယ်ဘက်လှည့်စေချင်ရင် `turn_left` ဖန်ရှင်ကောလ် လုပ်ရပါမယ်။ ဘီပါကောက်ခိုင်းချင်ရင် `pick_beeper` ဖန်ရှင်ကောလ် လုပ်ရပါမယ်။ ဖန်ရှင်ကောလ် လုပ်တဲ့ ပုံစံက

```
turn_left()
pick_beeper()
```

စသည်ဖြင့် ဖြစ်တယ်။

Python မှာ အင်ဒန့်ထ်ကို ဖြစ်ကတတ်ဆန်း လုပ်လို့မရဘူး။ ဘေးမျဉ်းကနေ ခွာတဲ့ အကွာအဝေး မညီတာနဲ့ ဆင်းတက်စ်အမှား ဖြစ်တယ်။ မလိုတဲ့နေရာမှာလည်း ခွာရေးလို့ မရဘူး။ ခေါင်းစီးကို ဘေးမျဉ်းနဲ့ ခွာထားကြည့်ပါ။ အယ်ရာဖြစ်တာကို တွေ့ရမယ်။ အင်ပို့စတိတ်မန့်လည်း ဘေးမျဉ်းနဲ့ ကွာနေလို့မရဘူး။ အခြား language တွေမှားလည်း အင်ဒန့်ထ်လုပ် ရေးကြပေမဲ့ Python မှာလောက် မတင်းကျပ်ဘူး။ အင်ဒန့်ထ်မလုပ်လည်း ဆင်းတက်စ်မှားတာ မဖြစ်ဘူး။

ကားရဲလ်ပရိုဂရမ်တစ်ခုမှာ `main` ဟာ အထူးတာဝန်တစ်ခု လုပ်ဆောင်ပေးရတယ်။ အဲဒါကတော့ ပရိုဂရမ်ဝင်းဒိုးမှာ Run Program ခလုတ် (ပုံ ၁.၅ မှာကြည့်ပါ) နှိပ်လိုက်ရင် တုံ့ပြန် လုပ်ဆောင်ပေးရတာပါ။ ဒါကြောင့် ကွန်မန်းတွေဟာ အဲဒီခလုတ် နှိပ်တော့မှပဲ စအလုပ်လုပ်တာ ဖြစ်တယ်။

Entry Point

'Meet Karel' ပရိုဂရမ်မှာ `main` ဖန်ရှင်နောက် အောက်ဆုံးအပိုင်းဟာ ပရိုဂရမ် run တဲ့အခါ ပထမဆုံး စတင်လုပ်ဆောင်ပေးရမဲ့ ဖန်ရှင်ကို ဖော်ပြပေးတာပါ။ အန်ထရီပွိုင့် (*entry point*) လို့ခေါ်တယ်။

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

`run_karel_program` ဖန်ရှင်ဟာ ကားရဲပရိုဂရမ် တစ်ခုအတွက် အန်ထရီပွိုင့် ဖြစ်တယ်။ ပရိုဂရမ် တက်လာတာနဲ့ တစ်ပါတည်း ခေါ်တင်ချင်တဲ့ ကမ္ဘာကို ဒီဖန်ရှင်မှာ ထည့်ပေးတယ်။ meet_karel.w ကမ္ဘာကို စစချင်းခေါ်တင်ထားချင်ရင် `"meet_karel"` ထည့်ပေးရမယ်။ ကမ္ဘာဖိုင်မရှိရင် အယ်ရာတက်ပြီး ပရိုဂရမ်ပွင့်လာမှာ မဟုတ်ဘူး။ ကမ္ဘာမထည့်ပေးထားဘဲ ဒီလို

```
run_karel_program()
```

ဆိုရင် 8×8 အရွယ် default ကမ္ဘာကို တင်ပေးပါတယ်။

ကားရဲလ်ကမ္ဘာတစ်ခုကို လိုချင်တဲ့ပုံစံ ဒီဇိုင်းဆွဲပြီး ဖိုင်နဲ့ သိမ်းထားရတာပါ။ ကမ္ဘာ ပုံစံချတဲ့ ပရိုဂရမ်လည်း ရှိတယ်။ ကမ္ဘာဖိုင်တွေက .w ဖိုင် အိပ်စ်တန်းရှင်းနဲ့ ဖြစ်တယ်။ ဒီစာအုပ်မှာပါတဲ့ ဥပမာတွေ၊ လေ့ကျင့်ခန်းတွေ အားလုံးအတွက် လိုအပ်တဲ့ ကမ္ဘာတွေကို အဆင်သင့်ပေးထားမှာပါ။ ကိုယ့်ဟာကို လုပ်ဖို့ မလိုဘူး။ စိတ်ဝင်စားရင် စမ်းကြည့်လို့ရအောင် စာမျက်နှာ (၁၉၁) နောက်ဆက်တွဲ (ခ) မှာ အကျဉ်းဖော်ပြပေးထားပါတယ်။

၁.၃ ကားရဲလ် ပရိုဂရမ် run ခြင်း

လိုအပ်တဲ့ဆော့ဖ်ဝဲတွေ ထည့်သွင်းနည်းကို စာမျက်နှာ (၁၆၅) နောက်ဆက်တွဲ (က) မှာ တစ်ဆင့်ချင်း ဖော်ပြပေးထားပါတယ်။ အခုက Python ပရိုဂရမ်တစ်ခုကို အရိုးရှင်းဆုံး (လွယ်တယ်လို့ မဆိုလို) run လို့ရတဲ့ နည်းကိုဖော်ပြပေးမှာပါ။ သဘောတရားပိုင်း နားလည်ဖို့ အထောက်အပံ့ဖြစ်မယ်။ အခုနည်းလမ်းကို အကြမ်းဖျဉ်း နားလည်အောင် ဖတ်ပြီးမှ နောက်ဆက်တွဲ (က) ကို ဖတ်စေချင်ပါတယ်။

မိုက်ခရိုဆော့ဖ် ဝင်းဒိုးမှာ Notepad ၊ အက်ပဲလ် မက်ခ်အိုအက်စ်မှာ TextEdit စတဲ့ တက်စ်အယ်ဒီတာတစ်ခုခုနဲ့ ပရိုဂရမ်ကုဒ်ရေးလို့ရတယ်။ ကုဒ်ဖိုင်ကို .py အိပ်စ်တန်းရှင်းနဲ့ သိမ်းရပါမယ်။ ပလိန်းတက်စ် (plain text) ဖိုင် ပါပဲ။ Python ကုဒ်ဖိုင်မို့လို့ .txt အစား .py နဲ့ သိမ်းတာပါ။ Python ဖိုင်နံမည်ကို စာလုံးအသေးနဲ့ပဲ ပေးတဲ့ ထုံးစံရှိတယ်။ စပေ့စ်နေရာမှာ _ (underscore) သုံးတဲ့ ထုံးစံရှိတယ်။ ဒါကြောင့် 'Meet Karel' ပရိုဂရမ်ကုဒ်ကို meet_karel.py ဖိုင်မှာ သိမ်းသင့်တယ်။

Python နဲ့ ရေးထားတဲ့ ပရိုဂရမ်ကို run မယ်ဆိုရင် Python ဆော့ဖ်ဝဲရှိရမှာပါ။ ဒီဆော့ဖ်ဝဲ အင်စတောလ် လုပ်နည်းကို နောက်ဆက်တွဲ (က) စာမျက်နှာ (၁၇၇) မှာ ဖော်ပြပေးထားပါတယ်။ Python ကုဒ်တွေကို ကွန်ပျူတာက တိုက်ရိုက် နားမလည်ပါဘူး။ Python ဆော့ဖ်ဝဲဟာ Python ကုဒ်တွေကို တိုက်ရိုက် နားလည်ပြီး ကွန်ပျူတာပေါ်မှာ run လို့ရအောင် ကြားခံဆောင်ရွက်ပေးတဲ့ ဆော့ဖ်ဝဲလို့ ယေဘုယျအားဖြင့် ယူဆနိုင်တယ်။

Python ထည့်ပြီးရင် `stanfordkarel` လိုက်ဘရီကို အောက်ပါကွန်မန်းနဲ့ အင်စတောလ် လုပ်ရပါမယ်။ အင်တာနက်ပေါ်ကနေ ဒေါင်းလုဒ် လုပ်ရတာမို့လို့ ကွန်နက်ရှင်ရှိရမယ်။

```
pip install stanfordkarel
```

ကားရဲလ်ပရိုဂရမ်မှာ ခေါ်တင်ချင်တဲ့ ကမ္ဘာဖိုင်တွေလည်း ရှိရပါမယ်။ meet_karel.w ကမ္ဘာဖိုင်က meet_karel.zip ဖိုင် worlds ဖိုဒါထဲမှာ ရှိပါတယ်။ `http://tinyurl.com/3mmm9c7j` လင့်ကနေ meet_karel.zip ဖိုင်ကို ဒေါင်းလုဒ်လုပ်ပါ။ ဒီ zip ဖိုင်ထဲက worlds ဖိုဒါကို meet_karel.py ဖိုင်ရှိတဲ့ နေရာမှာ ကော်ပီကူးထည့်ပါ။ ဝင်းဒိုးမှာ Command Prompt ၊ မက်ခ်အိုအက်စ်မှာ Terminal ဖွင့်ပြီး cd ကွန်မန်းနဲ့ ကုဒ်ဖိုင်ထားတဲ့ ဖိုဒါထဲကို သွားပြီး အောက်ပါအတိုင်း python ကွန်မန်းနဲ့ ကုဒ်ဖိုင်ကို run ပေးရပါမယ် (လာမဲ့စာမျက်နှာမှာ နမူနာပြထားတာ ကြည့်ပါ)။

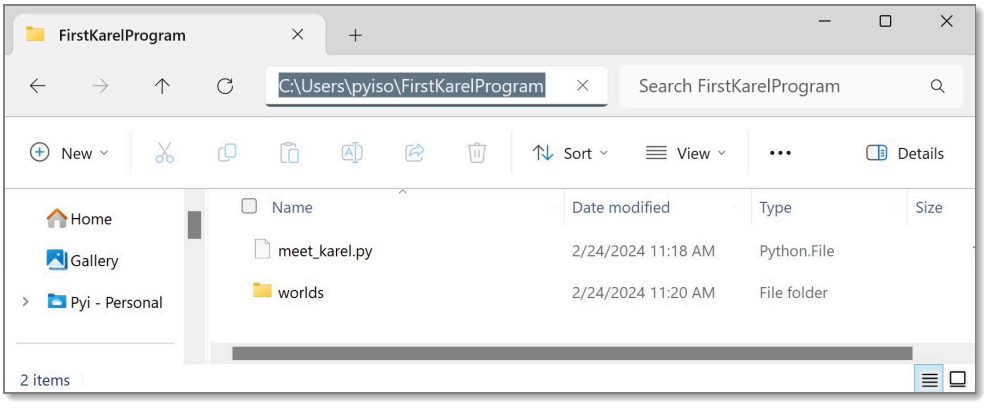
```
python meet_karel.py
```

ကုဒ်ရေးထားတာ ဆင်းတက်စ်အမှား မရှိဘူးဆိုရင် ကားရဲပရိုဂရမ် ပွင့်လာမှာပါ။

အခုဖော်ပြခဲ့တာက ကားရဲလ်ပရိုဂရမ်တစ်ခု run ဖို့ မဖြစ်မနေလုပ်ရမဲ့ အနည်းဆုံးလိုအပ်ချက်ပါ။ Python ဆော့ဖ်ဝဲ ရှိရမယ်။ `stanfordkarel` လိုက်ဘရီ အင်စတောလ် လုပ်ရမယ်။ ကမ္ဘာဖိုင်ပါတဲ့ worlds ဖိုဒါရှိရမယ်။ .py ဖိုင် တစ်ခုနဲ့ ပရိုဂရမ်ကုဒ်ကို သိမ်းရမယ်။ worlds ဖိုဒါနဲ့ ကုဒ်ဖိုင်ကို တစ်နေရာတည်းမှာ ထားရမယ်။ ပြီးရင် ကွန်မန်းလိုင်းမှာ

```
python your_karel_program.py
```

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



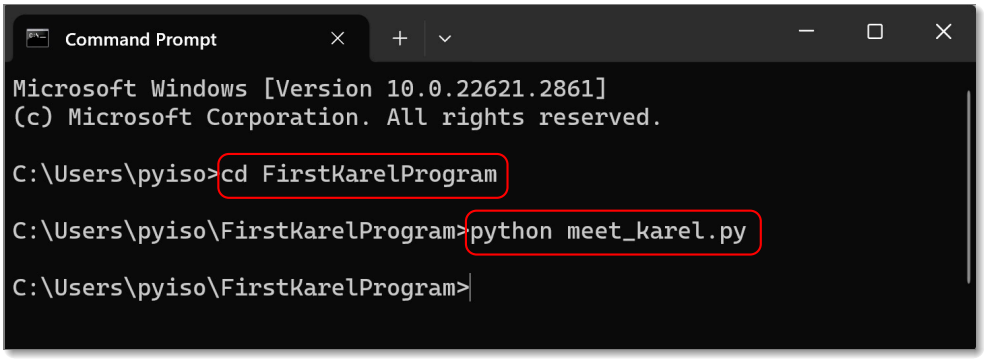
ပုံ ၁.၃

၁.၄ Python အင်စတောလ်လုပ်တဲ့အခါ ဘာတွေပါလဲ

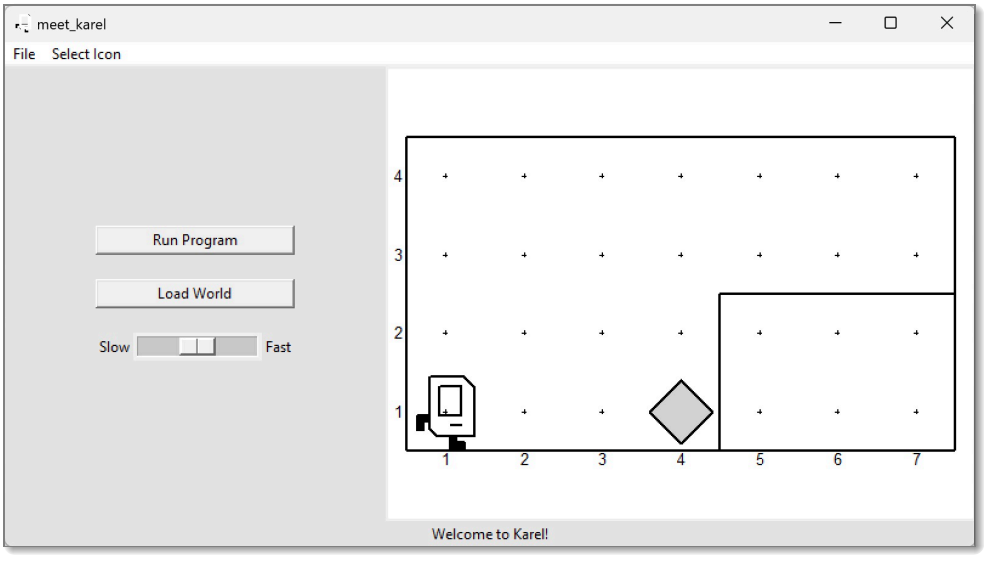
Python အင်စတောလ်လုပ်တယ်လို့ ယေဘုယျ ပြောပေမဲ့ အင်စတောလ်လုပ်တဲ့အခါ တကယ်တမ်းက ပရိုဂရမ်တစ်ခုတည်း ထည့်သွင်းပေးသွားတာ မဟုတ်ပါဘူး။ Python *interpreter* ပရိုဂရမ်၊ Python *standard library* နဲ့ အခြားလိုအပ်တဲ့ ပရိုဂရမ်တွေကို ထည့်ပေးသွားတာပါ။

Python Interpreter

အင်တာပရက်တာ ဆိုတာ ပရိုဂရမ်ကုဒ်တွေကို ဖတ်ပြီး ပရိုဂရမ်ကုဒ်ထဲက ညွှန်ကြားချက်တွေအတိုင်း လုပ်ဆောင်ပေးတဲ့ ပရိုဂရမ်ပါ။ Python အင်တာပရက်တာကတော့ Python နဲ့ရေးထားတဲ့ ကုဒ်တွေကို ဖတ်နိုင်တဲ့အပြင် ညွှန်ကြားထားတဲ့အတိုင်းလည်း လုပ်ဆောင်ပေးနိုင်ပါတယ်။ Python ကုဒ်တွေကို



ပုံ ၁.၄



ပုံ ၁.၅

ကွန်ပျူတာက တိုက်ရိုက် နားလည်တာ မဟုတ်ပါဘူး။ အင်တာပရက်တာကသာ တိုက်ရိုက်နားလည်တာပါ။ ၎င်းက တစ်ဆင့်ခံ၍ Python ကုဒ်တွေကို ကွန်ပျူတာပေါ်မှာ run ပေးရတာဖြစ်တယ်။ အင်တာပရက်တာကို python ကွန်မန်းနဲ့ အသုံးပြု run ရပါတယ်။

```
python meet_karel.py
```

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

Python Standard Library

Python အင်စတောလ်လုပ်တဲ့အခါ standard library လည်း တစ်ပါတည်း ထည့်သွင်းပေးသွားတယ်။ Standard library မှာ ကွဲပြားခြားနားတဲ့ ရည်ရွယ်ချက်အမျိုးမျိုးအတွက် ကွန်ပိုးနန့်တွေ အမြောက်အများ ပါဝင်ပါတယ်။ Standard library ကွန်ပိုးနန့်တွေကို နောက်ပိုင်းမှာ အသုံးပြုကြရမှာပါ။ ၎င်းတို့ကို အလျှင်းသင့်သလို ဖော်ပြပေးသွားပါမယ်။

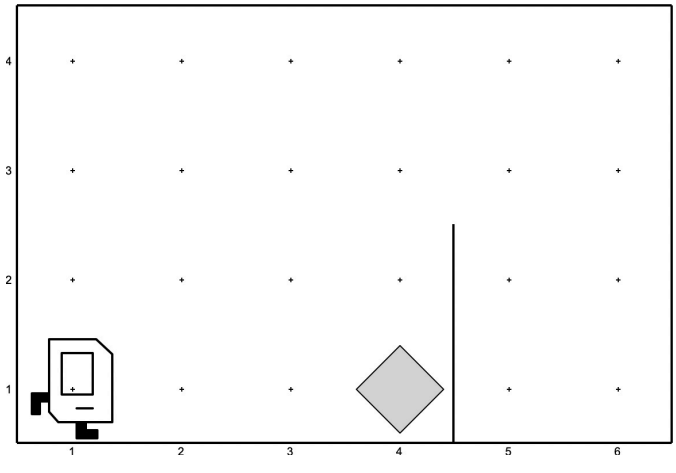
Standard အပြင် အခြားရရှိနိုင်တဲ့ လိုက်ဘရီတွေလည်း အများအပြားရှိပါတယ်။ pip ဟာ လိုက်ဘရီတွေ အင်စတောလ်လုပ်ဖို့ အသုံးပြုတဲ့ ပရိုဂရမ်တစ်ခုပါပဲ။ Python documentation မှာတော့ pip ကို Python *package* တွေ အင်စတောလ် လုပ်ဖို့ အသုံးအများဆုံး tool (ထောက်ကူပြုပစ္စည်း) လို့ ဆိုထားပါတယ်။ Programming language တွေနဲ့ပါတ်သက်ပြီး မော်ဂျူး (module)၊ ပက်ကေ့ချ် (package)၊ လိုက်ဘရီ (library) စတဲ့ အသုံးအနှုန်းတွေ တွေ့ရကြားရပါတယ်။ ဒီစကားလုံးတွေ၊ အခေါ်အဝေါ်တွေရဲ့ အဓိပ္ပါယ်သတ်မှတ်ချက်ကလည်း သက်ဆိုက်ရာ programming language အလိုက် ကွာခြားလေ့ရှိတယ်။ ဘီဂင်နာအနေနဲ့ ဒါနဲ့ပါတ်သက်ပြီး သိပ်ပြီးခေါင်းရှုပ်ခံစရာ မလိုသေးပါဘူး။ ပရိုဂရမ်ကုဒ် သိမ်းဆည်း၊ ဖွဲ့စည်း၊ ထားသို၊ ဖြန့်ဖြူးတဲ့ ကိစ္စတွေနဲ့ သက်ဆိုင်တဲ့ အခေါ်အဝေါ်တွေလို့ သိထားရင် လုံလောက်ပါပြီ။ နောက်ပိုင်းမှာ ဒီအခေါ်အဝေါ်တွေနဲ့ သက်ဆိုင်တဲ့ အဓိပ္ပါယ်သတ်မှတ်ချက်တွေကို အလျဉ်းသင့်ရင် သင့်သလို ဖော်ပြပေးပါမယ်။

၁.၅ Move Beeper to Other Side

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

ပုံ (၁.၆) မှ ဘိပါကို နံရံအခြားတစ်ဘက် အောက်ခြေကို ရွှေ့ပေးတဲ့ ပရိုဂရမ် ရေးကြည့်ပါ။ Python ထုံးစံအရ move_beeper_to_other_side.py ဖိုင်နဲ့ သိမ်းသင့်ပါတယ်။meet_karel.zip ဖိုင်ထဲမှာပါတဲ့ worlds ဖိုဒါမှာပဲ အခု ကမ္ဘာဖိုင် ထည့်ပေးထားပါတယ်။ move_beeper_to_other_side.w နံမည်နဲ့ပါ။ အန်ထရီပို့ဇ်အတွက် အခုလိုရေးရပါမယ်။

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



ပုံ ၁.၆

အခန်း ၂

ကားရဲလ်နှင့် ကွန်ထရိုးလ် စတိတ်မန့်များ

ကားရဲလ်ပရိုဂရမ်တစ်ခု ဖွဲ့စည်းတည်ဆောက်ပုံကို ရှေ့အခန်းမှာ လေ့လာခဲ့ကြပြီး ကွန်ထရိုးလ် စတိတ်မန့်တွေ ဖြစ်ကြတဲ့ **for** loop, `while` loop, **if** နဲ့ **if...else** တို့ကို အခုဆက်လက် လေ့လာကြပါမယ်။ ပရိုဂရမ်တစ်ခုကို run တဲ့အခါ ပရိုဂရမ်ကုဒ်ထဲက ညွှန်ကြားချက်တွေအတိုင်း ကွန်ပျူတာက လုပ်ဆောင်ပေးတာပါ။ ဒီလိုလုပ်ဆောင်ပေးတာကို အကွာစီကျူ (execute) လုပ်တယ်လို့ ခေါ်တယ်။ ကွန်ထရိုးလ် စတိတ်မန့်တွေဟာ ပရိုဂရမ်တစ်ခုရဲ့ execution flow ကို ထိန်းချုပ်ပေးတာ ဖြစ်တဲ့အတွက် *control flow statements* တွေလို့လည်း ခေါ်ပါတယ်။

၂.၁ **for** loop

ကွန်ထရိုးလ်စတိတ်မန့် တစ်မျိုးဖြစ်တဲ့ **for** loop ဟာ စတိတ်မန့် တစ်ခု (သို့) စတိတ်မန့် တစ်စုကို သတ်မှတ်ထားတဲ့ အကြိမ်အရေအတွက် ပြည့်အောင် ထပ်ခါထပ်ခါ ပြန်ကျော့ပေးပါတယ်။ `move` ကို နှစ်ဆယ့်ငါးကြိမ် ကျော့ပေးဖို့ `for` loop နဲ့ အခုလို

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

ရေးရပါတယ်။ `put_beeper`, `move`, `turn_left` စတိတ်မန့် သုံးကြောင်းတစ်စုကို အကြိမ်တစ်ရာ ကျော့ချင်ရင် ဒီလိုပါ

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

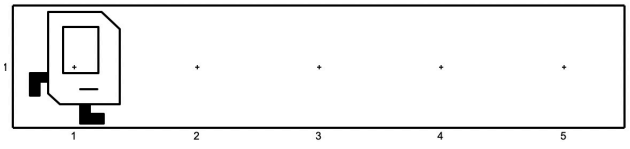
နှစ်ဆယ့်ငါးကြိမ်ကို `range(25)`, အကြိမ်တစ်ရာကို `range(100)` စသည်ဖြင့် တွေ့ရတယ်။ ယေဘုယျအားဖြင့် အကြိမ်အရေအတွက် N ကြိမ် ကျော့ချင်ရင်

```
for x in range(N):
    statement_1
    statement_2
    statement_3 etc.
```

ပုံစံနဲ့ သတ်မှတ်ရတာ။ N က အကြိမ်အရေအတွက်ကို ဖော်ပြတဲ့ ကိန်းပြည့်ဂဏန်းဖြစ်တယ်။ for loop ရေးတဲ့အခါ ကော်လံ : မကျန်ခဲ့ဖို့ သတိပြုရပါမယ်။ x ဟာ ဗေရီရေဘဲလ်တစ်ခုဖြစ်ပြီး i, j, k စတဲ့ အက္ခရာတစ်ခုနဲ့ ကိုယ်စားပြုလေ့ရှိတယ်။

ပြန်ကျော့စေချင်တဲ့ စတိတ်မန့်တွေကို for ရဲ့ ညာဘက်ကို အင်ဒန့်ထ် လုပ်ပေးရပါမယ်။ အောက်ပါ အတိုင်းဆိုရင် put_beeper ကိုပဲ အကြိမ်တစ်ရာ လုပ်မှာပါ။ move နဲ့ turn_left ပြန်ကျော့မဲ့ထဲမှာ မပါတော့ဘူး။

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

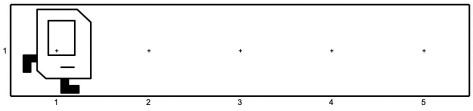


ပုံ ၂.၁

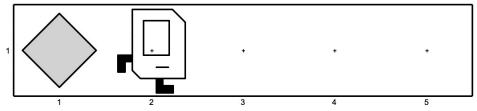
ပုံ (၂.၁) ကားရဲလ်ကမ္ဘာမှာ ကွန်နာအားလုံးမှာ ဘီပါတစ်ခုစီ ချထားပေးရမယ်။ ကွန်နာငါးခုရှိတာမို့ လို့ စုစုပေါင်း ဘီပါငါးခု ချထားရမှာပါ။ move ကတော့ လေးကြိမ်ပဲ လုပ်ရမယ် (ကားရဲလ်ရှေ့မှာ ကွန်နာ လေးခုပဲ ရှိတယ်)။ ဒီအတွက်ကို အခုလိုရေးနိုင်ပါတယ်။

```
# File: make_row_of_5_beepers.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_beepers")
```

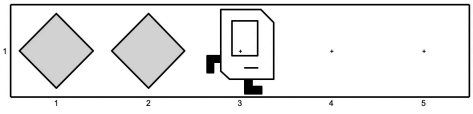
for loop ဟာ put_beeper နဲ့ move ကို လေးကြိမ်ကျော့ပေးမှာပါ။ တစ်ကြိမ်ပြီးတိုင်း ရှိနေမဲ့ အနေအထား တစ်ခုချင်းစီကို ပုံ (၂.၂) မှာ တွေ့နိုင်ပါတယ်။ လေးကြိမ်မြောက် နောက်ဆုံးတစ်ကျော့အပြီး ပုံ (၂.၂) (c) မှာ ဘီပါတစ်ခုလိုနေသေးတာ တွေ့ရမယ်။ for loop ပြီးတဲ့အခါ put_beeper တစ်ခါ ထပ်လုပ်ပေးတော့မှ တစ်တန်းလုံးအပြည့်ဖြစ်သွားမှာပါ။



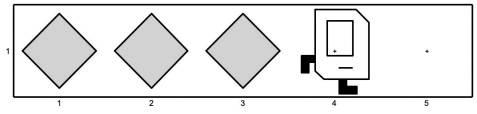
(က) မူလ အနေအထား



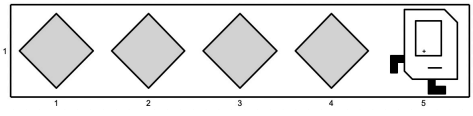
(ခ) ပထမ တစ်ကျော့ပြီး



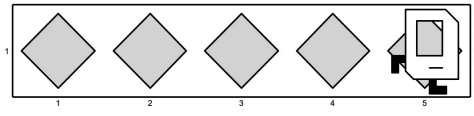
(ဂ) ဒုတိယ တစ်ကျော့ပြီး



(ဃ) တတိယ တစ်ကျော့ပြီး



(င) စတုတ္ထမြောက် ကျော့ပြီး



(စ) နောက်ဆုံး putBeeper လုပ်ပြီး

ပုံ ၂.၂

Off-by-one bug

ပြီးခဲ့တဲ့ဥပမာမှာ `for` loop အပြီး `put_beeper` မလုပ်မိရင် ပရိုဂရမ်ဟာ လိုချင်တဲ့အတိုင်း မဖြစ်ပါဘူး။ ဘိပါတစ်ခုလိုနေမယ်။ ဒီလိုပြဿနာမျိုးဟာ အဖြစ်များတဲ့ အမှားဖြစ်ပြီး *off-by-one bug* လို့ခေါ်ပါတယ်။ Loop သုံးတဲ့အခါ ကြိုရလေ့ရှိတဲ့ ဖြစ်တတ်တဲ့ အမှား (*bug*) ဖြစ်တယ်။ သတ်မှတ်ထားတဲ့အတိုင်း၊ ဖြစ်သင့်တဲ့အတိုင်း ပရိုဂရမ်က အလုပ်မလုပ်ဘဲ ဖြစ်နေတဲ့ အမှားကို ကွန်ပျူတာအသုံးအနှုန်းမှာ bug လို့ခေါ်တာပါ။ Bug ကိုလိုက်ရှာပြီး မှန်အောင်ပြင်ပေးတာကိုတော့ debug လုပ်တယ်လို့ခေါ်ပါတယ်။

ဘိပါတစ်ခုကျန်ခဲ့တဲ့ အမှားမျိုးဟာ off-by-one bug သာဓကတစ်ခုမျှသာ ဖြစ်တယ်။ ခုနှစ်နဲ့ ဆယ့်သုံးကြား ဂဏန်း ခုနှစ်လုံးရှိပါတယ် (ခုနှစ်နဲ့ ဆယ့်သုံးအပါဝင် ဆိုလျှင်)။ $13 - 7 = 6$ လို့တွက်မိရင် တစ်လုံး လိုနေပါလိမ့်မယ်။ သုံးပေခြား တစ်တိုင် အလျားပေသုံးဆယ်ရှိ တဖြောင့်တည်း ခြံစည်းရိုးတစ်ခု အတွက် တိုင်စုစုပေါင်း တစ်ဆယ့်တစ်တိုင် လိုပါမယ်။ $30 \div 3 = 10$ လို့ တွက်ရင် မမှန်ပါဘူး။ တစ်ခုလိုတာ အပြင် တစ်ခုပိုနေတာလည်း ဖြစ်တတ်တယ်။ စောနက ခြံစည်းရိုးမှာပဲ ထရံအချပ်အရေအတွက်ကတော့ ကိုးချပ်ဖြစ်ရမှာပါ။ $30 \div 3 = 10$ လို့ တွက်ရင် တစ်ခုပိုနေပါမယ်။ ခုနှစ်နဲ့ ဆယ့်သုံးကို ထည့်မစဉ်းစားရင် ကြားမှာ ဂဏန်းငါးလုံးရှိတာပါ။ $13 - 7 = 6$ လို့တွက်မိရင် တစ်လုံး ပိုနေပါလိမ့်မယ်။ ဒီဥပမာ အားလုံးဟာ အခြေခံသဘောတရားအားဖြင့် သိပ်မကွာခြားတဲ့ off-by-one bug ပုံစံကွဲအမျိုးမျိုး ဖြစ်တယ်။

'Make Row of Five Beepers' ဒုတိယ ဗားရှင်း

ပရိုဂရမ်ရေးတဲ့အခါ ပရိုဂရမ်မာတွေ တစ်ယောက်နဲ့တစ်ယောက် စဉ်းစားပုံ စဉ်းစားနည်း၊ ဖြေရှင်းနည်း ထပ်တူကျလေ့မရှိပါဘူး။ ဘိပါငါးခု အတန်းလိုက် ချပေးတဲ့ ပရိုဂရမ်ကိုပဲ နောက်ဗားရှင်းတစ်မျိုးနဲ့ 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 ဗားရှင်းနဲ့ ပြောင်းပြန်ဖြစ်နေတာ) သတိထားကြည့်ပါ။

၂.၂ အင်ဒန့်ထ်လုပ်ခြင်းနှင့် ကုဒ်စထရက်ချာ

Python ဟာ အင်ဒန့်ထ်လုပ်ခြင်းဖြင့် ပရိုဂရမ်ကုဒ် ဖွဲ့စည်းပုံ စထရက်ချာကို ဖော်ပြတဲ့ programming language ဖြစ်ပါတယ်။ Python ကုဒ်တွေကို ဘလောက် (*block*) တွေနဲ့ ဖွဲ့စည်းထားတယ်လို့ ရှုမြင်နိုင်တယ်။ ဘလောက်ဆိုတာ ကုဒ်တွေကို အုပ်စုတစ်စု အဖြစ် ဖွဲ့စည်းထားတာကို ဆိုလိုတာပါ။

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

အခု Python ကုဒ်မှာ အင်ဒန့်ထ်မလုပ်ထားဘဲ ဘယ်ဘက်စွန်း ကပ်ရေးထားတဲ့ **def** main(): (ဖန်ရှင်ခေါင်းစည်း) ဟာ အပေါ်ဆုံးအဆင့် (top level) ဖြစ်တယ်လို့ ယူဆတယ်။ ဖန်ရှင်ခေါင်းစည်းအောက် အင်ဒန့်ထ်လုပ်ထားတဲ့ ကုဒ်လိုင်းအားလုံးဟာ main ဖန်ရှင် ဘလောက်ထဲမှာ အကျုံးဝင်တယ်။ pick_beeper အထိပါတယ်။ put_beeper (ပထမတစ်ခု)၊ for loop နဲ့ pick_beeper တို့ဟာ ပထမအဆင့် အင်ဒန့်ထ်ဖြစ်တယ်။

တစ်ခါ for အောက်မှာ ဒုတိယတစ်ဆင့် အင်ဒန့်ထ်လုပ်ထားတဲ့ ကုဒ်လိုင်းအားလုံး for ဘလောက်ထဲမှာ အကျုံးဝင်တယ်။ move နဲ့ put_beeper ပါဝင်တယ်။ pick_beeper ကတော့ ပထမအဆင့် ပြန်ဖြစ်သွားတဲ့အတွက် for ဘလောက်ထဲမှာမပါဘူး။

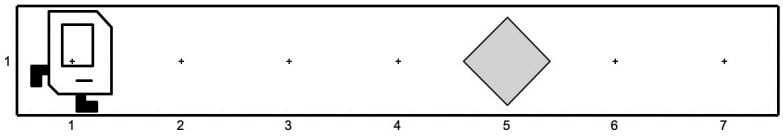
အောက်ဆုံး turn_left() ကလည်း အင်ဒန့်ထ်လုပ် မထားတဲ့အတွက် အပေါ်ဆုံးအဆင့်ပဲ။ **def** main(): နဲ့ အဆင့်တူတာပေါ့။ main ဖန်ရှင်ဘလောက် အပြင်ဘက်မှာလို့ ယူဆရမယ်။

အင်ဒန့်ထ်လုပ်ထားတဲ့ အဆင့်ကို ကြည့်ပြီး ဘလောက်တွေရဲ့ ဖွဲ့စည်းပုံကို ကွက်ကွက်ကွင်းကွင်း ထင်းကနဲ မြင်နိုင်တယ်။ အပေါ်ကကုဒ်ကို 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 အင်ဒန့်ထ်လုပ်ထားတဲ့ အဆင့်က ဖော်ပြနေတယ်။

၂.၃ **while** loop

အခြေအနေတစ်ရပ် မှန်နေသ၍ စတိတ်မန့်တွေကို တစ်ကြိမ်ပြီးတစ်ကြိမ် ပြန်ကျော့ လုပ်ဆောင်စေချင်ရင် **while** loop ကို အသုံးပြုပါတယ်။ for နဲ့ while loop နှစ်ခုလုံးက ပြန်ကျော့ပေးတာ ဖြစ်ပေမဲ့ for ကို အကြိမ်အရေအတွက် အတိအကျသိတဲ့ကိစ္စမျိုးမှာ သုံးလေ့ရှိပြီး while ကိုတော့ ဘယ်နှစ်ကြိမ်လဲ ကြိုတွက်လို့မရဘဲ အခြေအနေ အပေါ်မူတည်ပြီး လုပ်ဆောင်ပေးရမဲ့ အကြိမ်အရေအတွက် ကွာခြားနိုင်တဲ့ ကိစ္စမျိုးတွေမှာ သုံးလေ့ရှိတယ်။ ဥပမာတစ်ခုနဲ့ ကြည့်ရင် ပိုနားလည်ပါလိမ့်မယ်။

(၁) လမ်းပေါ်မှာ ဘီပါတစ်ခုရှိမယ်။ ဘီပါ ဘယ်ကွန်နာမှာရှိမှာလဲ၊ လမ်းဘယ်လောက်အရှည်ဖြစ်မလဲ ကြိုတင်မသိဘူးလို့ ယူဆပါ။ နမူနာကမ္ဘာ တစ်ခုကို ပုံ (၂.၃) မှာ တွေ့နိုင်တယ်။ ဘီပါကို သွားပြီး ကောက်ခိုင်းရပါမယ်။ အလားတူ မည်သည့်ကမ္ဘာအတွက်မဆို ဘီပါကောက်ပေးနိုင်ရမှာ ဖြစ်တယ်။ ဘီပါရှိ



ပုံ ၂.၃

မဲ့ကွန်နာကို ကြိုတင်မသိထားတဲ့အတွက် ဘယ်နှစ်ကြိမ် move လုပ်ခိုင်းရမှာလဲ မသိဖြစ်နေတယ်။ အကြိမ်အရေအတွက် မသိတဲ့အတွက် for loop နဲ့ အဆင်မပြေတော့ပါဘူး။

ဘိပါမရှိသ၍ တစ်ကြိမ်ပြီးတစ်ကြိမ် move လုပ်ခိုင်းမယ်ဆိုရင် နောက်ဆုံးမှာ ဘိပါရှိတဲ့ကွန်နာကို ရောက်သွားမှာ သေချာပါတယ်။ အဲဒီလို လုပ်ခိုင်းဖို့ရာအတွက် while loop နဲ့ ရေးထားတာကို အခုလို တွေ့ရပါမယ်။

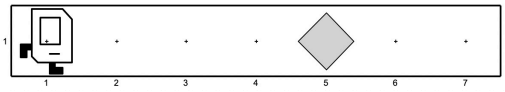
```
while no_beepers_present():
    move()
```

ကားရဲလ်ဟာ အခြေခံကွန်မန်း လေးခုအပြင် လက်ရှိ ကွန်နာမှာ ဘိပါရှိ/မရှိ၊ ရှေ့မှာ နံရံပိတ်နေ/မနေ၊ အရှေ့ဘက်ကို မျက်နှာမူထား/မထား စတဲ့ သူနဲ့ (သို့) သူ့ရဲ့ကမ္ဘာနဲ့ သက်ဆိုင်တဲ့ အခြေအနေတစ်ရပ်ရပ် မှန်/မမှန် စစ်နိုင်ပါတယ်။ no_beepers_present က လက်ရှိ သူရှိနေတဲ့ ကွန်နာမှာ 'ဘိပါ မရှိဘူးလား' ဆိုတဲ့ အခြေအနေ စစ်ခိုင်းတာပါ။ လက်ရှိကွန်နာမှာ ဘိပါ 'မရှိ' ရင် ဒီအခြေအနေက မှန်တယ်လို့ ယူဆရမှာပေါ့။ အကယ်၍ 'ရှိ' နေရင်တော့ မှားတယ်လို့ ယူဆရမယ်။ အမှန် (သို့) အမှား ရလဒ်ထွက်မဲ့ ဒီလိုမျိုး အခြေအနေစစ် ကွန်မန်းတွေကို ကွန်ဒီရှင် (condition) လို့ခေါ်ပါတယ်။

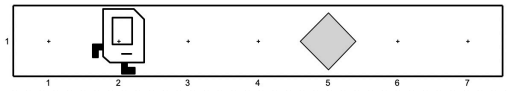
while loop အလုပ်လုပ်ပုံက ဒီလိုပါ။ no_beepers_present ကွန်ဒီရှင် စစ်ပါတယ်။ မှန်ရင် move လုပ်ပါတယ်။ ပြီးရင် ကွန်ဒီရှင်ပြန်စစ်တယ်။ မှန်ရင် move ကို နောက်ထပ်တစ်ကြိမ် ထပ်ကျော့ပါတယ်။ ကွန်ဒီရှင်စစ်လိုက်၊ မှန်ရင် တစ်ခါထပ်ကျော့လိုက်၊ ကွန်ဒီရှင်ပြန်စစ်လိုက်၊ မှန်ရင် နောက်တစ်ခါ ထပ်ကျော့လိုက်၊ ... ဒီအတိုင်းဆက်ပြီး ကွန်ဒီရှင်မှန်နေသ၍ while loop က move ကို ပြန်ကျော့ပေးမှာပါ။ ကွန်ဒီရှင် စစ်လိုက်လို့ မှားသွားပြီဆိုရင်တော့ ထပ်မကျော့တော့ဘဲ ရပ်သွားမှာဖြစ်တယ်။ while loop တစ်ကြိမ်ကျော့ပြီးတိုင်း ရှိနေမဲ့ အနေအထား တစ်ခုချင်းကို ပုံ (၂.၄) မှာ လေ့လာကြည့်ပါ။ လေးကြိမ်မြောက် ကျော့ပြီးနောက် ကွန်ဒီရှင်စစ်လိုက်တဲ့အခါ မှားနေပြီ။ ဒီအခါ while loop က ထပ်မကျော့တော့ဘူး။ ဒီလို loop က ပြန်ကျော့နေတာ ရပ်သွားရင် loop ကနေ ထွက်တယ် (loop exits) လို့ ပြောလေ့ရှိတယ်။

အပြည့်အစုံ ဖော်ပြပေးထားတဲ့ ပရိုဂရမ်ကို လေ့လာကြည့်ပါ။

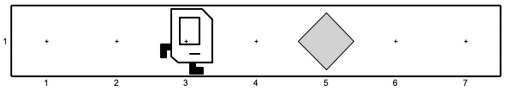
```
# File: go_pick_beeper.py
from stanfordkarel import *


def main():
    while no_beepers_present():
        move()

    pick_beeper()
```



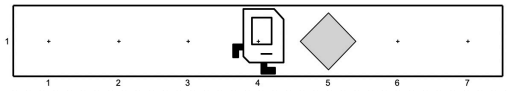
(က) မူလ အနေအထား



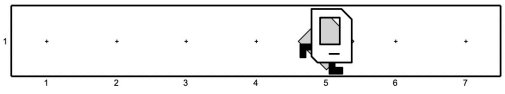
(ခ) ပထမ တစ်ကျော့ပြီး



(ဂ) ဒုတိယ တစ်ကျော့ပြီး



(ဃ) တတိယ တစ်ကျော့ပြီး



(င) စတုတ္ထမြောက် ကျော့ပြီး

ပုံ ၂.၄

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

ကားရဲလ် ကွန်ဒီရှင်များ

beepers_present ကွန်ဒီရှင် ကျတော့ လက်ရှိကွန်နာမှာ ဘီပါရှိရင် အမှန်၊ မရှိရင် အမှား ရလဒ်ထွက်တယ်။ no_beepers_present နဲ့ ဆန့်ကျင်ဘက်ပေါ့။ ကားရဲလ် ကွန်ဒီရှင်တွေအားလုံးကို တေဘဲလ် (၂.၃) မှာ ကြည့်ပါ။ ပုံမှန်စစ်တာနဲ့ အငြင်းပုံစံစစ်တာ နှစ်မျိုးကို ယှဉ်တွဲပြထားပါတယ်။

ကွန်ဒီရှင်	ဆန့်ကျင်ဘက် ကွန်ဒီရှင်	စစ်ပေးသည့် အခြေအနေ
front_is_clear	front_is_blocked	ရှေ့မှာ နံရံကပ်လျက် ရှိမရှိ
left_is_clear	left_is_blocked	ဘယ်ဘက်မှာ နံရံကပ်လျက် ရှိမရှိ
right_is_clear	right_is_blocked	ညာဘက်မှာ နံရံကပ်လျက် ရှိမရှိ
beepers_present	no_beepers_present	လက်ရှိကွန်နာမှာ ဘီပါရှိမရှိ
beepers_in_bag	no_beepers_in_bag	ကားရဲလ်၏ ဘီပါအိပ်ထဲ ဘီပါရှိမရှိ
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

တေဘဲလ် ၂.၁ ကားရဲလ် စစ်နိုင်သည့် ကွန်ဒီရှင်များ

while loop ဆင်းတက်စ်

while loop ရေးတဲ့ ပုံစံက ယေဘုယျအားဖြင့် ဒီလိုပါ။

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

condition က ကားရဲလ်ကွန်ဒီရှင် တစ်ခုဖြစ်တယ်။ ကော်လံ : မကျန်ခဲ့အောင် ဂရုစိုက်ပါ။ *condition* မှန်နေသေးသ၍ ပြန်ကျော့စေချင်တဲ့ စတိတ်မန့်တွေကို `while` အောက်မှာ အင်ဒန့်ထ်လုပ်ထား ရပါမယ်။ ရှေ့မှာရှင်းနေသ၍ `move` လုပ်တဲ့ `while` loop ကို အခုလို

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

ရေးပါတယ်။

'Make Beeper Row' ဥပမာ

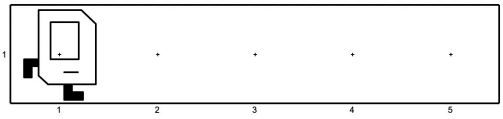
'Make Row of Five Beepers' ဥပမာမှာ လမ်းရဲ့အလျားဟာ မပြောင်းလဲဘူး ယူဆတာမို့လို့ `for` loop ကို အသုံးပြုတာ ဆီလျော်ပါတယ်။ လမ်းရဲ့အရှည်ကို ကြိုမသိထားဘူး၊ တစ်လမ်းလုံး ဘိပါတွေ ဖြန့်ထားပေးရမယ်ဆိုရင် `while` နဲ့ ရေးရမှာပါ။

```
# 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()
```

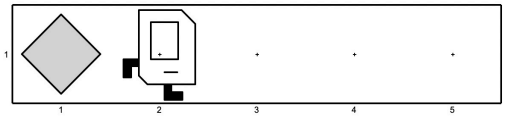
တစ်ကြိမ်ကျော့ပြီးတိုင်း ရှိနေမဲ့ အနေအထားကို ပုံ (၂.၅) မှာ ကြည့်ပါ။ လေးကြိမ်မြောက်အပြီး `front_is_clear` စစ်တဲ့အခါ မှားနေပြီဖြစ်လို့ ထပ်မကျော့တော့ဘဲ loop ကနေ ထွက်သွားမယ်။ ဒီအခါမှာ ဘိပါတစ်ခု လိုနေသေးတယ်။ `put_beeper` ထပ်လုပ်ရမယ်။ တစ်ခါပဲ လုပ်ရမှာပါ။ ဒါကြောင့် `while` loop ထဲမပါအောင် ပထမအဆင့် အင်ဒန့်ထ်ပဲ ဖြစ်ရမယ်။

၂.၄ `if` စတိတ်မန့်

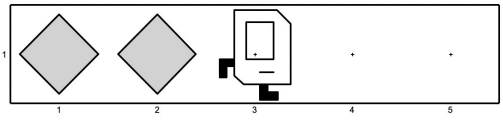
တစ်ခု (သို့) တစ်ခုထက်ပိုတဲ့ စတိတ်မန့်တွေကို အခြေအနေတစ်ရပ် မှန်တော့မှပဲ လုပ်ဆောင်စေချင်တဲ့ အခါ **`if`** ကို အသုံးပြုနိုင်တယ်။ ဥပမာ



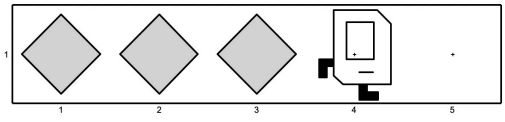
(က) မူလ အနေအထား



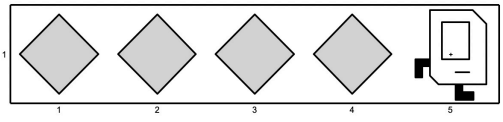
(ခ) ပထမ တစ်ကျော့ပြီး



(ဂ) ဒုတိယ တစ်ကျော့ပြီး



(ဃ) တတိယ တစ်ကျော့ပြီး



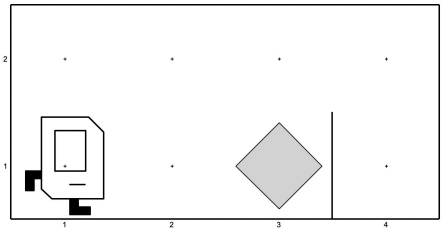
(င) စတုတ္ထမြောက် ကျော့ပြီး

ပုံ ၂.၅

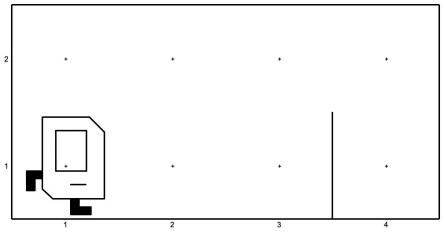
```
if beepers_present():
    pick_beeper()
```

`if` စတိတ်မန့်ဟာ `beepers_present` ကွန်ဒီရှင် မှန်တော့မှပဲ `pick_beeper` လုပ်မှာပါ။ မှားရင် မ လုပ်ပါဘူး။

ပုံမှာတွေ့ရတဲ့ ကမ္ဘာနှစ်ခုထဲက တစ်ခုမှာ ကားရဲလ်ရှိနေမယ် ဆိုပါစို့။ ဘိပါရှိတဲ့ ကမ္ဘာဆိုရင် ဘိပါကို



(က)



(ခ)

ပုံ ၂.၆

နံရံအခြားဘက် အောက်ခြေကို ရွှေ့ပေးရမယ်။ မရှိတဲ့ကမ္ဘာဆိုရင် 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()
        # 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 အင်ဒန့်ထ်လုပ်ထားတာ သတိပြုပါ။ အဲဒီ စတိတ်မန့်အားလုံး beepers_present ဖြစ်မှ လုပ်ဆောင်မှာ ဖြစ်တယ်။ ဘိပါမရှိတဲ့ ကမ္ဘာမှာ စမ်းကြည့်ဖို့အတွက် Load World ခလုတ်နှိပ်ပြီး ခေါ်တင်ပါ။ ဒါမှမဟုတ် ပရိုဂရမ်ကုဒ်မှာ "mbtos2" လို့ပြင်ပြီး ပြန် run ပါ။

if စတိတ်မန့် ယေဘုယျစထရက်ချာကို အခုလိုတွေ့ရတယ်။

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

condition က front_is_clear, beepers_present စတဲ့ ကားရဲလ် ကွန်ဒီရှင် တစ်ခုခုဖြစ်မယ်။ တေဘဲလ် (၂.၃) မှာ ကားရဲလ်ကို စစ်ခိုင်းလို့ရတဲ့ ကွန်ဒီရှင်အားလုံး ပြထားပါတယ်။

၂.၅ if...else စတိတ်မန့်

အခြေအနေတစ်ရပ် မှန်တဲ့အခါ လုပ်ဆောင်ရမှာနဲ့ မှားတဲ့အခါ လုပ်ဆောင်ရမှာ မတူကွဲပြားနေတဲ့အခါ **if...else** ကို သုံးပါတယ်။ ရှေ့က if စတိတ်မန့် ဥပမာ ပုံ (၂.၆) မှာ ဘိပါမရှိရင် နဂိုမူလနေရာကို ပြန်လာခိုင်းချင်တယ်ဆိုပါစို့။ if...else နဲ့ အခုလို 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` စတိတ်မန့်က ကွန်ဒီရှင်မှန်ရင် `if` ဘလောက်ကိုလုပ်ဆောင်ပေးပြီး ကွန်ဒီရှင်မှားရင်တော့ `else` ဘလောက်ကို လုပ်ဆောင်ပေးတာပါ။ ယေဘုယျပုံစံက ဒီလိုပါ

if $condition$:
    $statement_{1a}$
    $statement_{2a}$
    $statement_{3a}$ etc.
else:
    $statement_{1b}$
    $statement_{2b}$
    $statement_{3b}$ etc.

၂.၆ Nested ကွန်ထရိုးလ် စတိတ်မန့်များ

ကွန်ထရိုးလ် စတိတ်မန့် ဘလောက်ထဲမှာ ကွန်ထရိုးလ် စတိတ်မန့် ရှိလို့ရတယ်။။ Nested ကွန်ထရိုးလ် စတိတ်မန့်လို့ ခေါ်ပါတယ်။

Nested **for** loop

ဘိပါသုံးခုချလိုက်၊ ရှေ့တိုးလိုက် လုပ်တာကို လေးကြိမ်ကျော့ချင်ရင် for loop နဲ့ အခုလို ရေးရမယ်ဆိုတာ သိခဲ့ပြီးပါပြီ။

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

ဒီ for loop ထဲက put_beeper သုံးကြောင်းကိုလည်း နောက်ထပ် for တစ်ခုနဲ့ ရေးလို့ ရရမှာပေါ့။ ဒီလို ဖြစ်သွားမှာပါ

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

for loop နှစ်ခု ဆင့်ရေးထားလို့ nested for loop လို့ခေါ်ပါတယ်။ Loop တစ်ခုချင်းကို သီးခြားရည်ညွှန်းချင်တဲ့အခါ အပြင် for loop နဲ့ အတွင်း for loop လို့ ပြောလေ့ရှိတယ်။ အင်ဒန့်ထဲ လုပ်ထားပုံအရ အပြင် for loop ဘလောက်ထဲမှာ အတွင်း for loop နဲ့ move ပါဝင်တယ်။ အတွင်း for loop ဘလောက်ထဲမှာတော့ put_beeper ပဲပါတယ်၊ move မပါ ပါဘူး။ အပြင် for loop မှာ ဗေရီ ရေဘဲလ် i ဆိုရင် အတွင်းမှာ i မဖြစ်ရပါဘူး။ j, ဒါမှမဟုတ် k မတူတာ တစ်ခုခုဖြစ်ရပါမယ်။

အတန်းလိုက် ကွန်နာ ငါးခုမှာ တစ်ကွန်နာ ဘိပါ (၂၅) 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**

ကမ္ဘာပါတ်လည် ဘိပါခြံစည်းရိုး ခတ်ပေးဖို့အတွက် for နဲ့ while nest လုပ်ထားတာကို လေ့လာကြည့်ပါ။ (၁, ၁) ကွန်နာမှာ အရှေ့ဘက်လည့် အနေအထားကနေ စမယ်လို့ ယူဆပါ။ ကမ္ဘာအရွယ်အစား အမျိုးမျိုးကို ခြံစည်းရိုး ခတ်ပေးနိုင်ရပါမယ်။

တောင်ဘက် နံရံတလျှောက် ဘိပါတွေချသွားမယ် (နောက်ဆုံး ကွန်နာမှာ ဘိပါမချဘဲ ချန်ထားမယ်)။ ပြီးရင် အရှေ့ဘက် နံရံအတွက် အဆင်သင့်ဖြစ်အောင် ဘယ်ဘက်လှည့်မယ်။ ပုံ (၂.၇) (က) နဲ့ (ခ) ကို ကြည့်ပါ။ အရှေ့၊ မြောက် နဲ့ အနောက်ဘက် နံရံတွေအတွက်လည်း ဒီအတိုင်း လုပ်သွားရုံပါပဲ။ ပုံ (၂.၇) (ဂ)၊ (ဃ)၊ (င) အသီးသီးကို ကြည့်ပါ။

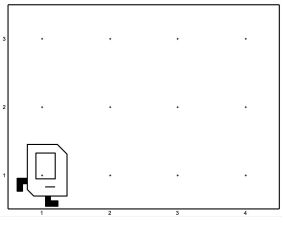
နံရံတစ်ဘက် အတွက် ဆိုရင် အခုလို

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

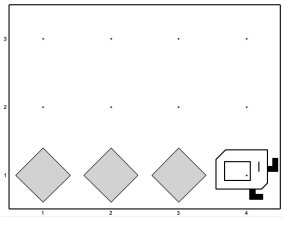
ဖြစ်မယ်။ (while ဘလောက်မှာ turn_left မပါတာ ဂရုပြုပါ)။ နံရံလေးဘက်အတွက် လေးကြိမ် လုပ်ရမှာဆိုတော့ for နဲ့ အခုလို

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

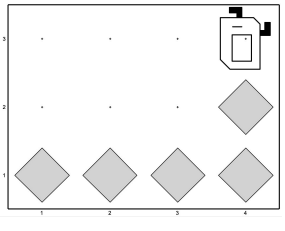
ရေးရပါမယ်။



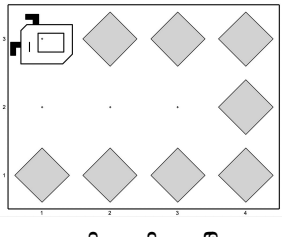
(က) မူလ အနေအထား



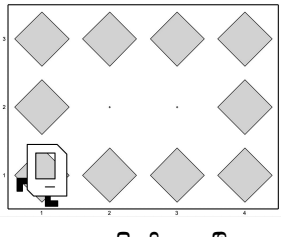
(ခ) ပထမ တစ်ကျော့ပြီး



(ဂ) ဒုတိယ တစ်ကျော့ပြီး



(ဃ) တတိယ တစ်ကျော့ပြီး



(င) စတုတ္ထမြောက် ကျော့ပြီး

ပုံ ၂.၇

ပရိုဂရမ် တစ်ခုလုံး အတွက်ဆိုရင် အခုလိုပါ

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


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


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

Nested **while** and **if**

while နဲ့ if nest လုပ်လို့လည်း ရတာပေါ့။ လမ်းပေါ်မှာ ဘီပါတွေက ကြိုရာကျပန်း ရှိနေမယ်။ ဘီပါတွေကို အမှိုက်တွေလို့ ယူဆပြီး ရှင်းပေးရပါမယ်။ လမ်းအလျားကို ကြိုမသိဘူး၊ ကွန်နာတစ်ခုမှာ ဘီပါတစ်ခုထက် ပိုမရှိနိုင်ဘူးလို့ ယူဆပါ။ လမ်းအဆုံးထိ while loop နဲ့ ရွှေ့ခိုင်းလို့ ရမယ်။ ရောက်တဲ့ ကွန်နာတိုင်းမှာ ဘီပါရှိရင် 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 ဘီပါရှိတော့မှပဲ ကောက်ပေးဖို့အတွက်။

ပြီးခဲ့တဲ့ဟာနဲ့ အလားတူတဲ့ နောက်ထပ် ဥပမာ တစ်ခုကတော့ လမ်းတစ်လျှောက် ကွန်နာတွေမှာ ဘီပါရှိရင် ကောက်ခိုင်းပြီး မရှိရင် တစ်ခုချပေးရပါမယ်။ တနည်းအားဖြင့် ကွန်နာတစ်ခုချင်းရဲ့ ဘီပါရှိ/မရှိ အခြေအနေကို ဆန့်ကျင်ဘက် ပြောင်းတာပေါ့။

```
# File: toggle_beepers.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။ ကျန်တာအားလုံး လမ်းရှင်းတဲ့ ပရိုဂရမ်နဲ့ တူတူပဲ။

Nested `if`

အခြေအနေ တစ်ရပ် မှန်တော့မှပဲ လုပ်ချင်ရင် `if` ကို သုံးတယ်။ အခြေအနေ 'နှစ်ရပ်' လုံးနဲ့ ကိုက်ညီမှ လုပ်ဆောင်စေချင်တဲ့အခါ nested `if` သုံးနိုင်ပါတယ်။ ညာဘက်လည်းရှင်း လက်ရှိကွန်နာမှာလည်း ဘီပါမရှိမှ ဘီပါတစ်ခု ချထားခိုင်းမယ်ဆိုပါစို့။ အခုလိုရေးနိုင်ပါတယ်

```
if right_is_clear():
    if no_beepers_present():
        put_beeper()
```

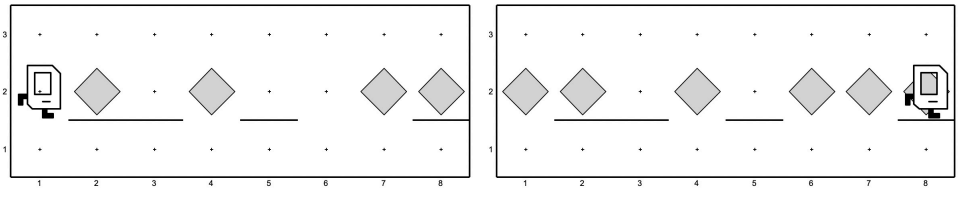
ညာဘက်မှာ ရှင်းနေမှ အပြင် `if` က အတွင်း `if` ကို လုပ်ဆောင်စေမှာပါ။ အတွင်း `if` ကလည်း ဘီပါမရှိမှပဲ `put_beeper` လုပ်မှာပါ။ ညာဘက်မှာ ပိတ်နေရင်သော်လည်းကောင်း ဘီပါရှိနေရင်သော်လည်းကောင်း `pick_beeper` လုပ်မှာ မဟုတ်ပါဘူး။ `right_is_clear` နဲ့ `no_beepers_present` အခြေအနေ နှစ်ခုလုံး မှန်မှပဲ လုပ်မှာပါ။

ပုံ (၂.၈) (က) မှာ ဒုတိယ လမ်းတစ်လျှောက် လမ်းပြင်တဲ့အလုပ်ကို ကားရဲလ်ကို တာဝန်ပေးတယ်လို့ ယူဆပါ။ ဘီပါရော ညာဘက်နံရံပါ မရှိတဲ့ ကွန်နာတွေက လမ်းအခြေအနေ တော်တော်ဆိုးနေတဲ့ နေရာတွေ။ ဒီလိုနေရာတွေမှာ ဘီပါတစ်ခု ဖြည့်ပြီး လမ်းပြင်ပေးရမှာပါ။

```
# 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 ဘလောက်ထဲမှာ nested if နဲ့ move ပါဝင်တယ်။ nested if က ညာဘက်လည်းရှင်း ဘိပါလည်းမရှိမှ လက်ရှိကွန်နာမှာ ဘိပါတစ်ခု ချခိုင်းထားတာပါ။ while ထဲမှာ if၊ အဲဒီ if ထဲမှာမှ နောက်ထပ် if တစ်ခု ဆင့်ထားတဲ့အတွက် သုံးဆင့် nest လုပ်ထားတာပါ။ အင်ဒန့်ထ် လုပ်ထားတာတွေ ဂရုစိုက်ကြည့်ဖို့ လိုတယ်။ ဘယ်ဘလောက်ထဲမှာ ဘာရှိနေလဲဆိုတာ မြင်အောင် လုပ်ရမှာပါ။ ကိုယ်တိုင် ရေးရင်လည်း အင်ဒန့်ထ် ဂရုစိုက်ပြီး 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 ပရိုဂရမ်းမင်းမှာ အရေးကြီးဆုံး အခြေခံသဘောတရားတစ်ခု ဖြစ်တယ်။ ဖန်ရှင်ဆိုတာ ဘာလဲ၊ ဘာကြောင့် အရေးပါရတာလဲ၊ ဖန်ရှင်တွေကို ပရိုဂရမ် ဒီဇိုင်းပြုလုပ် ရေးသားတဲ့အခါ ဘယ်လိုအသုံးချတာလဲ စတာတွေကို ဒီအခန်းမှာ လေ့လာကြပါမယ်။

၃.၁ ဖန်ရှင် သတ်မှတ်ခြင်း

ညာဘက် လှည့်ခိုင်းချင်တိုင်း `turn_left` သုံးခါရေးနေရတာ ရေရှည်အဆင်မပြေပါဘူး။ `turn_right` လို့ပဲ တိုက်ရိုက် ရေးလို့ရရင် ပိုပြီးတော့ အဆင်ပြေမှာပါ။ ဒီလို လိုအပ်ချက်မျိုးကို ဖြည့်ဆည်း ပေးဖို့အတွက်ဟာ ဖန်ရှင်တွေရဲ့ အဓိက ရည်ရွယ်ချက်တွေထဲက တစ်ခုဖြစ်တယ်။ `turn_right` ဖန်ရှင်ကို အခုလို သတ်မှတ်နိုင်ပါတယ်။

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

ဒီလို သတ်မှတ်ထားပြီးရင် ညာဘက်လှည့်ချင်တဲ့အခါ

```
turn_right()
```

လို့ တိုက်ရိုက်ပြောလို့ ရသွားမှာ ဖြစ်ပါတယ်။ `turn_left` သုံးခါ ရေးဖို့ မလိုတော့ပါဘူး။

ဖန်ရှင်တစ်ခု သတ်မှတ်တယ် (*defining a function*) ဆိုတာ ကိစ္စတစ်ခု ဖြေရှင်းဆောင်ရွက်ဖို့အတွက် စတိတ်မန့်တွေကို ယူနစ်တစ်ခုအဖြစ် ဖွဲ့စည်းထားလိုက်တာပါပဲ။ ၎င်းယူနစ်အတွက် အမည်တစ်ခုကိုလည်း သတ်မှတ်ပေးတယ်။

ဖန်ရှင် သတ်မှတ်မယ် ဆိုရင် **def** keyword သုံးရပါတယ်။ အထက်ပါ `turn_right` ဖန်ရှင် သတ်မှတ်ချက် (*function definition*) မှာ

```
def turn_right():
```

ကို ဖန်ရှင် ဟက်ဒ်ဒါ (*function header*) လို့ ခေါ်တယ် (မြန်မာလို ဆိုရင်တော့ ဖန်ရှင် ခေါင်းစည်းပေါ့)။ `turn_right` က ဖန်ရှင် အမည်။ ပါရာမီတာပါတဲ့ ဖန်ရှင်ဆိုရင် ဝိုက်ကွင်းထဲမှာ ပါရာမီတာတွေ

သတ်မှတ်ရတယ်။ ဥပမာ (x, y)။ ပါရာမီတာ မပါရင်တော့ () ပဲဖြစ်မယ်။ ကားရဲလ်မှာ ဖန်ရှင်အားလုံးဟာ ပါရာမီတာ မပါတဲ့အတွက် () ပဲ ဖြစ်မှာပါ။ ဖန်ရှင်ဟက်ဒ်ဒါ လိုင်းအဆုံးမှာ ကော်လံ ':' ထည့်ပေးဖို့ လိုပါတယ်။

ဖန်ရှင် ဟက်ဒ်ဒါအောက် အင်ဒန့်ထ်လုပ်ထားတဲ့ လိုင်းအားလုံးဟာ ၎င်းဖန်ရှင်နဲ့ သက်ဆိုင်တဲ့ ကုဒ်ဘလောက် ဖြစ်တယ်။ အခု `turn_right` ဖန်ရှင် ဘလောက်မှာ `turn_left` သုံးကြိမ်ပါတယ်။

```
turn_right()
```

လုပ်ခိုင်းတာက (သတ်မှတ်ထားတဲ့) ဖန်ရှင်ကို အသုံးပြုတာ ဖြစ်တယ်။ ဒီအခါမှာ ၎င်းဖန်ရှင်နဲ့ 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 အထက်ဆိုရင် ကြားမှာ underscore (_) ခြားပေးလေ့ ရှိတယ်။

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

name_of_function()
```

ဖန်ရှင်နံမည် အဓိပ္ပါယ် အရေးကြီးပါတယ်

စာရေးတာပဲဖြစ်ဖြစ်၊ ပရိုဂရမ်ကုဒ် ရေးတာပဲဖြစ်ဖြစ် စိတ်ထဲ တွေးတဲ့အတိုင်း၊ စဉ်းစားတဲ့အတိုင်း ပေါ်လွင်အောင် ဖော်ပြနိုင်တာဟာ အားသာချက်တစ်ခုပါပဲ။ ဖန်ရှင် သတ်မှတ်ထားခြင်း အားဖြင့် ညာဘက်လှည့်ခိုင်းရင် `turn_right` ဘိပါ နှစ်ဆယ့်ငါးခု ချရင် `put_25_beepers` တိုက်ရိုက် ဖော်ပြလို့ ရတာဟာ အရေးပါတဲ့ ကိစ္စဖြစ်ပါတယ်။ ဘာသာစကားတစ်ခုရဲ့ ဖော်ပြနိုင်စွမ်း 'အား' (expressive power) ကို ထပ်လောင်းအားဖြည့်ပေးတာလို့ ဆိုရမှာပါ။

ဖန်ရှင်လုပ်ဆောင်ပေးတဲ့ ကိစ္စကို သိသာစေမဲ့၊ နားလည်ရလွယ်မဲ့ နံမည်မျိုး ဂရုစိုက်ရွေးချယ်တာကလည်း အရေးပါပါတယ်။ ကားရဲလ်ပရိုဂရမ်တွေမှာ အမိန့်ပေးခိုင်းစေတဲ့ ပုံစံနဲ့ ဖန်ရှင်နံမည်ပေးလေ့ရှိတယ်။ ဥပမာ `turn_north`, `pick_all_beepers` ။

ဖန်ရှင်သတ်မှတ်ချက်နဲ့ ဖန်ရှင်ကောလ် ဥပမာ တချို့ကို လေ့လာကြည့်ပါ။ ဘိပါ နှစ်ဆယ့်ငါးခု ချပေးတဲ့ `put_25_beepers` ဖန်ရှင်ပါ

```
def put_25_beepers():
    for i in range(25):
        put_beeper()

put_25_beepers()
```

ဒါကတော့ ကွန်နာတစ်ခုမှာ ရှိတဲ့ ဘိပါအားလုံးကောက်ပေးတဲ့ ဖန်ရှင်ဖြစ်ပါတယ်

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

```
pick_all_beepers()
```

ဖန်ရှင် အသုံးပြုတဲ့အခါ ကိစ္စတစ်ခုကို ပိုပြီး အလွယ်တကူ လုပ်လို့ရသွားတယ်။ ဘိပါအားလုံး ကောက်မယ်ဆိုရင် pick_all_beepers ဖန်ရှင်ခေါ်လိုက်ရုံပဲ။ ဘိပါ နှစ်ဆယ့်ငါးခု ချချင်ရင် put_25_beepers ဖန်ရှင်ခေါ်လိုက်ရင်ရပြီ။ ဖန်ရှင်တစ်ခုကို အသုံးပြုတဲ့အခါ အဲဒီဖန်ရှင်ကို ဘယ်လိုသတ်မှတ်ထားလဲ ပြန်စဉ်းစားနေဖို့ မလိုပါဘူး။ ဥပမာ ...

အခန်း (၁) မှာ လိုက်ဘရီဆိုတာ ဘာလဲ အကျဉ်း ဖော်ပြခဲ့တယ်။ မေ့ထရစ် A နဲ့ B မြှောက်လဒ် $A \times B$ ကို numpy လိုက်ဘရီ ဖန်ရှင် matmul နဲ့ အဖြေရှာခဲ့ပါတယ်။

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

matmul ဖန်ရှင်ကို လိုက်ဘရီ ထုတ်လုပ်တဲ့ ပညာရှင်တွေက ရေးပေးထားတာ။ အဲဒီဖန်ရှင်ကို ဘယ်ပုံ ဘယ်နည်း သတ်မှတ်ထားတယ်ဆိုတာ အသုံးပြုသူအတွက် အရေးမကြီးဘူး။ မေ့ထရစ်တွေကို ဒီဖန်ရှင်နဲ့ မြှောက်လို့ရတယ်ဆိုတာ သိရင် သုံးလို့ရနေတာပါပဲ။ အခြား မေ့ထရစ် လုပ်ထုံးလုပ်နည်း ဖန်ရှင်တွေလည်း numpy လိုက်ဘရီမှာ ပါဝင်ပါတယ်။ မိမိ လုပ်ချင်တဲ့ မေ့ထရစ် လုပ်ထုံးလုပ်နည်းအတွက် ဖန်ရှင်ကို သိရင် (လိုက်ဘရီ documentation ဖတ်ပြီး ရှာလို့ရတယ်) လိုအပ်တဲ့အခါ ခေါ်သုံးလိုက်ရုံပါပဲ။ ပရိုဂရမ်တွေ တည်ဆောက်ရာမှာ လိုက်ဘရီတွေ မရှိမဖြစ်လိုအပ်တယ်။ ဖန်ရှင်တွေဟာ လိုက်ဘရီတွေရဲ့ အဓိက အစိတ်အပိုင်းတွေ ပဲဖြစ်ပါတယ်။

ဖန်ရှင် အခြေခံအုတ်ချပ်များ

pick_all_beepers ဖန်ရှင်ကို အခြေခံအုတ်ချပ်သဖွယ် အသုံးပြုပြီး အခြားဖန်ရှင်တွေကို သတ်မှတ်နိုင်ပါတယ်။ လမ်းတစ်လျှောက် ဘိပါ အားလုံး ရှင်းပေးမဲ့ clean_the_street ဖန်ရှင်ကို လေ့လာကြည့်ပါ။

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

    pick_all_beepers()
```

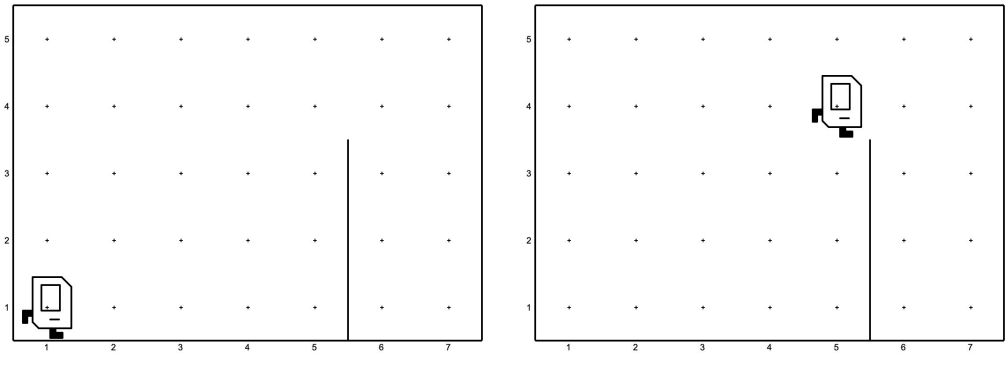
လမ်းတစ်လမ်းလုံး ရှင်းဖို့အတွက် ကွန်နာတစ်ခုက ဘိပါအားလုံး ကောက်ပေးတဲ့ pick_all_beepers ကို အခြေခံ အုတ်ချပ်သဖွယ် အသုံးပြုထားတာပါ။

ရိုးရှင်းတဲ့ အခြေခံ ဖန်ရှင်လေးတွေကနေ ပိုပြီး ရှုပ်ထွေးခက်ခဲတဲ့ ကိစ္စတွေ ဖြေရှင်း ဆောင်ရွက်ပေးနိုင်တဲ့ ဖန်ရှင်တွေကို တစ်ဆင့်ပြီး တစ်ဆင့် တည်ဆောက်ယူလို့ရတဲ့ သဘောကို တွေ့ရပါတယ်။ ကားရဲလ် ကမ္ဘာထဲက ရှိသမျှ ဘိပါအားလုံး ရှင်းပေးမဲ့ clean_the_world ဖန်ရှင် သတ်မှတ်မယ် ဆိုပါစို့။ clean_the_street ကို အခြေခံအုတ်ချပ်သဖွယ် ဆက်လက် အသုံးပြုနိုင်မှာ ဖြစ်တယ်။

၃.၂ ပရီကွန်ဒီရှင်နှင့် ပို့စ်ကွန်ဒီရှင်

ပရီကွန်ဒီရှင် (*precondition*) နဲ့ (*postcondition*) ဟာ ဖန်ရှင်နဲ့ ပါတ်သက်ပြီး ဂယန္ထ နားလည်ဖို့ လိုအပ်တဲ့ အရေးကြီးတဲ့ သဘောတရားနှစ်ခုပါ။ ဖန်ရှင် မလုပ်ဆောင်မီ ကြိုတင်ရှိနေရမဲ့ အခြေအနေကို ပရီကွန်ဒီရှင်လို့ ခေါ်ပြီး လုပ်ဆောင်ပြီး ရှိရမဲ့ အခြေအနေကို ပို့စ်ကွန်ဒီရှင်လို့ ခေါ်ပါတယ်။ သတ်မှတ်ထားတဲ့ ပရီကွန်ဒီရှင်နဲ့ ကိုက်ညီမှသာ ဖန်ရှင်တစ်ခုဟာ သူလုပ်ဆောင်ပေးရမဲ့ ကိစ္စကို မှန်ကန်အောင် ဆောင်ရွက်ပေးနိုင်မှာပါ။ ဖန်ရှင် အသုံးပြုတဲ့အခါမှာရော တည်ဆောက်တဲ့အခါမှာပါ ပရီကွန်ဒီရှင် ပို့စ်ကွန်ဒီရှင်တွေ အပေါ် အခြေခံပြီး တိတိကျကျစဉ်းစားဖို့ ပဓာနကျပါတယ်။

ပုံ ၃.၁ (က) မှ (ခ) အနေအထားသို့ ကားရဲလ်က တိုင်ထိပ်အရောက် တက်သွားရပါမယ်။ တိုင်အကွာအဝေး၊ အမြင့် အမျိုးမျိုးနဲ့ အလားတူ ကမ္ဘာတွေမှာလည်း အလုပ်လုပ်ရပါမယ်။ (နံရံကို တိုင်ဟု ယူဆပါ)။ တိုင်အောက်ခြေကိုသွားတာနဲ့ တိုင်ထိပ်တက်တာ အလုပ်နှစ်ခု ပါဝင်တယ်လို့ မြင်နိုင်တယ်။ ဒီအတွက် ဖန်



(က) မတိုင်မီ 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()
```

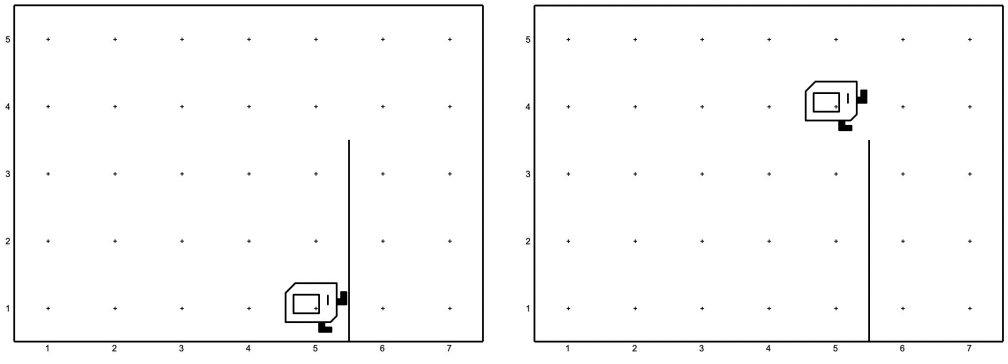
အထက်ပါ ဖန်ရှင်နှစ်ခုနဲ့ go_to_top ကို ဆက်လက် သတ်မှတ်ပါမယ်

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

go_to_pole အပြီးမှာ ကားရဲလ်ဟာ တိုင်ခြေမှာ အရှေ့ဘက်မျက်နှာမူပြီး ရှိနေမှာပါ။ ascend_pole က တိုင်ခြေမှာ ကားရဲလ် အပေါ်ဘက်ကို မျက်နှာမူတဲ့ အနေအထားကနေ စရပါမယ်။ go_to_pole ပြီးရင် ascend_pole အတွက် အသင့်အနေအထားဖြစ်အောင် turn_left လုပ်ပေးရပါမယ်။

ဖန်ရှင် စတင်မလုပ်ဆောင်မီ ကြိုတင်ရှိနေရမဲ့ အခြေအနေကို ပရီကွန်ဒီရှင်လို့ ပြောခဲ့ပါတယ်။ ဖန်ရှင် သတ်မှတ်တဲ့အခါ ပရီကွန်ဒီရှင်ကို တိတိကျကျ စဉ်းစားဖို့ လိုပါတယ်။ ဖန်ရှင်အသုံးပြုတဲ့အခါမှာလည်း သတ်မှတ်ထားတဲ့ ပရီကွန်ဒီရှင် အတိုင်းကိုက်ညီဖို့ လိုတယ်။ `ascend_pole` ပရီကွန်ဒီးရှင်းဟာ တိုင်ခြေမှာ ကားရဲလ် အပေါ်ဘက်ကို မျက်နှာမူတဲ့ အနေအထား ဖြစ်ပါတယ်။ ပုံ ၃.၂ (က) ကို ကြည့်ပါ။

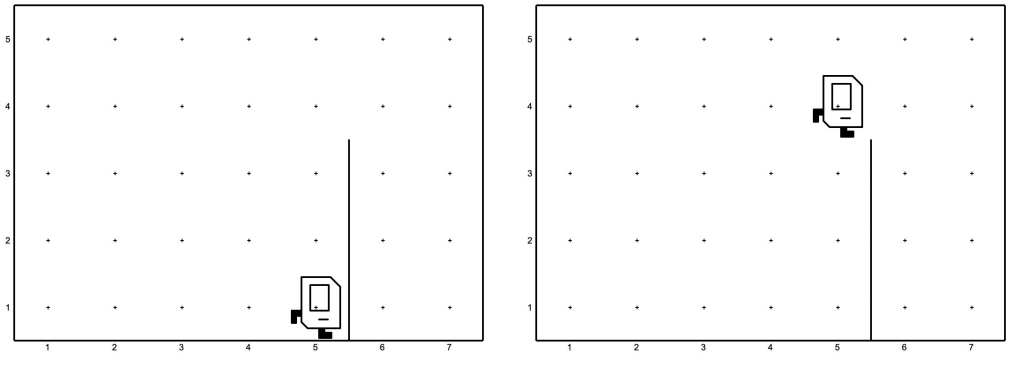
ဖန်ရှင် လုပ်ဆောင်အပြီးမှာ ရှိနေရမဲ့ အခြေအနေကို ပို့စ်ကွန်ဒီရှင်လို့ ဖော်ပြခဲ့တယ်။ ဖန်ရှင်တစ်ခုဟာ ပရီကွန်ဒီရှင်နဲ့ ကိုက်ညီတဲ့ အနေအထားကနေ စတင်ရင် ပို့စ်ကွန်ဒီရှင်နဲ့ ကိုက်ညီအောင်လုပ်ဆောင်ပေးပြီး အဆုံးသတ်ရမှာပါ။ ပုံ ၃.၂ (ခ) မှာ `ascend_pole` ပို့စ်ကွန်ဒီရှင်ကို တွေ့နိုင်ပါတယ်။



(က) (ခ)

ပုံ ၃.၂ **`ascend_pole` ပရီကွန်ဒီးရှင်းနှင့် ပို့စ်ကွန်ဒီးရှင်း**

ဖန်ရှင် ပရီကွန်ဒီရှင် ပို့စ်ကွန်ဒီရှင်ကို သင့်တော်သလို သတ်မှတ်နိုင်ပါတယ်။ ပုံ (၃.၃) တွင် `ascend_pole` အတွက် အခြား ရွေးချယ်နိုင်တဲ့ ပရီ နဲ့ ပို့စ် ကွန်ဒီရှင်ကို ကြည့်ပါ။



(က) (ခ)

ပုံ ၃.၃ **`ascend_pole` ပရီ နဲ့ ပို့စ် ကွန်ဒီရှင်**

အထက်ပါ ပရီ နဲ့ ပို့စ် ကွန်ဒီရှင်အရ `ascend_pole` ကို အခုလို

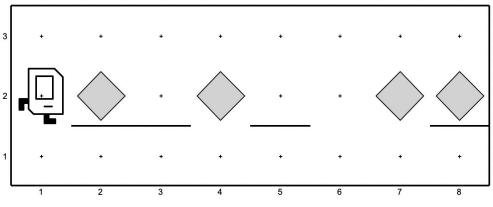
```
def ascend_pole():
    turn_left()
    while right_is_blocked():
        move()
    turn_right()
```

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 ပရိုဂရမ်မှာ ကွန်နာတစ်ခုဟာ ဘိပါလည်းမရှိ၊ အောက်ဘက် ကပ်လျက် နံရံလည်းမရှိရင် ဘိပါတစ်ခု ချထားပေးရပါတယ်။ ဒီကိစ္စ ဆောင်ရွက်ပေးဖို့အတွက် ဖန်ရှင်တစ်ခု သတ်မှတ်နိုင်ပါတယ်။



ပုံ ၃.၄

```
def repair_corner():
    if right_is_clear():
        if no_beepers_present():
            put_beeper()
```

အခြေအနေနှစ်ခုစလုံး မှန်တော့မှ ဘိပါချပေးဖို့ nested if သုံးထားတာက နားလည်ရ အတန်အသင့် ခက်ခဲနိုင်ပါတယ်။ ဒါပေမဲ့ repair_corner ကို အသုံးပြုတဲ့အခါ ဘယ်လိုရေးထားလဲ စဉ်းစားစရာ မလိုပါဘူး။ ဘိပါလည်းမရှိ၊ ညာဘက် နံရံလည်းမရှိတဲ့ ကွန်နာမှာ ဘိပါချချင်ရင် repair_corner ဖန်ရှင်နဲ့ လုပ်လို့ရတယ်ဆိုတာ သိထားရင် သုံးလို့ရပြီ။

ဖန်ရှင် သတ်မှတ်တဲ့အခါ ဆောင်ရွက်ပေးစေချင်တဲ့ ကိစ္စကို 'ဘယ်လို လုပ်ရမှာလဲ' အသေးစိတ် စဉ်းစားရမှာဖြစ်ပေမဲ့ အသုံးပြုတဲ့အခါမှာတော့ ဒီလိုအသေးစိတ်တွေကို ထပ်ပြီး စဉ်းစားဖို့ မလိုတော့ပါဘူး။ ဖန်ရှင် 'ဘာလုပ်ပေးတာလဲ' ကပဲ အရေးကြီးတယ်။ 'ဘယ်လို' တည်ဆောက်ထားလဲ သိစရာမလိုဘဲ 'ဘာ' လုပ်ပေးလဲ သိရုံနဲ့ အသုံးပြုလို့ရစေတာကို abstraction လုပ်တယ်လို့ခေါ်ပါတယ်။ Abstraction လုပ်ခြင်းအတွက် ဖန်ရှင်တွေဟာ အဓိက အကျဆုံး အခြေခံ အုတ်ချပ်တွေပါပဲ။

Abstraction လုပ်ထားလိုက်ခြင်းအားဖြင့် ရှုပ်ရှုပ်ထွေးထွေးတွေ ထပ်ခါထပ်ခါ ခေါင်းရှုပ်ခံ စဉ်းစားနေဖို့ မလိုအပ်တော့ဘဲ ကိစ္စတစ်ခုကို အလွယ်တကူ လုပ်ဆောင်လို့ရသွားတယ်။ ကုဒ်တွေဖတ်ရတာလည်း ပိုရှင်းပြီး နားလည်ရ လွယ်ကူသွားတယ်။ ဒါကြောင့် ပရိုဂရမ်တွေ တည်ဆောက်ရာမှာ abstraction လုပ်ခြင်းဟာ အရေးပါပါတယ်။ အရေးကြီးဆုံးလို့ ဆိုရင်လည်း မမှားဘူး။ ကြီးမားရှုပ်ထွေးတဲ့ ဆော့ဖ်ဝဲတွေကို လေ့လာကြည့်ရင် abstraction ပေါင်းများစွာနဲ့ တစ်ဆင့်ပြီးတစ်ဆင့်၊ တစ်လွှာပြီးတစ်လွှာ တည်ဆောက်ထားတယ်ဆိုတာ တွေ့ရမှာပါ။

repair_corner ဟာ အနိမ့်ဆုံးအလွှာက အခြေခံ abstraction တစ်ခု ဖြစ်တယ် ဆိုပါစို့။ ၎င်းကို 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 အခြေခံအစိတ်အပိုင်းအဖြစ် အသုံးပြုနိုင်ပါတယ်။ ဒီလို abstraction အလွှာတွေ အဆင့်ဆင့်နဲ့ ပရိုဂရမ် တည်ဆောက်နည်းတွေကို ဆက်လက်လေ့လာကြရပါမယ်။

၃.၄ Bottom-up Programming

ကြီးကျယ်ခမ်းနားတဲ့ အဆောက်အအုံကြီးတွေ၊ လျှို့ဝှက်ဆန်းကြယ်တဲ့ သဘာဝဖြစ်စဉ်တွေ၊ အံ့ဖွယ်သိပ္ပံနှင့် နည်းပညာ ဆန်းသစ်တီထွင်မှုတွေ စတဲ့အရာတွေအားလုံးဟာ ရိုးရှင်းတဲ့ အစိတ်အပိုင်းလေးတွေနဲ့ ဖွဲ့စည်းထားတာပါပဲ။ ဒါကြောင့် အရွယ်အစား ကြီးမား ရှုပ်ထွေးတဲ့ ပရိုဂရမ်တွေကိုလည်း ရိုးရှင်းတဲ့ အစိတ်အပိုင်းလေးတွေနဲ့ အခြေပြု ဖွဲ့စည်းတည်ဆောက်သင့်တယ်လို့ ယူဆမယ်ဆိုရင် ကျိုးကြောင်းဆီလျော်တယ်ပဲ ဆိုရမှာပါ။

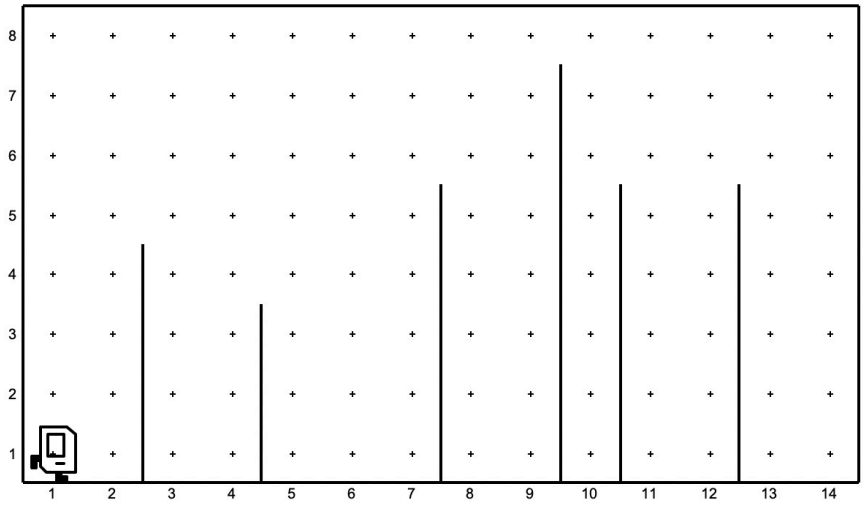
ခက်ခဲရှုပ်ထွေးတဲ့ ကိစ္စတစ်ခုကို ဖြေရှင်းတဲ့အခါ အပိုင်းတွေခွဲပြီး တစ်ပိုင်းချင်းကို သီးခြားဖြေရှင်းလေ့ရှိတယ်။ အပိုင်းတွေခွဲလိုက်ခြင်းအားဖြင့် တစ်ပိုင်းစီဟာ မူလဖြေရှင်းရမဲ့ ကိစ္စလောက် မခက်ခဲတော့ပါဘူး။ အရွယ်အစားအားဖြင့်လည်း နဂိုထက် ငယ်သွားမယ်။ အပိုင်း တစ်ပိုင်းချင်းစီကို ဖြေရှင်းတဲ့အခါမှာလည်း အလားတူပဲ လုပ်ဆောင်နိုင်တယ်။ အပိုင်းတစ်ပိုင်းကို သူ့ထက်သေးငယ်တဲ့ အပိုင်းတွေအဖြစ် ထပ်မံခွဲထုတ်နိုင်ပါတယ်။ ဒီတိုင်း တစ်ဆင့်ပြီးတစ်ဆင့် လုပ်သွားမယ်ဆိုရင် နောက်ဆုံးမှာ အလွယ်တကူ ဖြေရှင်းလို့ရတဲ့ အစိတ်အပိုင်းလေးတွေ ဖြစ်သွားရမှာပါပဲ။ *Bottom-up programming* ဆိုတာကတော့ ပရိုဂရမ်ရေးပြီး ကိစ္စတစ်ခုကို ဖြေရှင်းရာမှာ ရိုးရှင်းသထက်ရိုးရှင်း၊ သေးငယ်သထက်သေးငယ်တဲ့ အပိုင်းလေးတွေဖြစ်လာအောင် အဆင့်ဆင့်ပိုင်းခွဲပြီး အောက်ခြေအရိုးရှင်းဆုံးအလွှာကနေ အပေါ်ကိုတစ်ဆင့်ချင်းတက် ဖြေရှင်းတဲ့နည်း ဖြစ်တယ်။ လေ့လာကြည့်ကြရအောင်။

ပုံ (၃.၅) ကမ္ဘာမှာ ကားရဲလ် တန်းကျော်ပြေးပြိုင်ရပါမယ်။ ထောင်လိုက်နံရံတွေကို တန်းတွေလို့ယူဆပါ။ တန်းတွေဟာ အနိမ့်အမြင့် အမျိုးမျိုးဖြစ်နိုင်ပါတယ်။ ကမ္ဘာရဲ့ အပေါ်ဘက် နံရံကို ထိတဲ့အထိတော့ မြင့်လို့မရပါဘူး။ ဒါမှ တန်းကိုကျော်ပြီး အခြားဘက်ကို သွားလို့ရမှာပါ။ $(1, 1)$ ကွန်နာကနေ တာစထွက်မှာဖြစ်ပြီး $(14, 1)$ မှာ ပန်းဝင်ရမှာပါ။

တန်းကျော်ပြေးတဲ့ ကိစ္စမှာ ပါဝင်တဲ့ အပိုင်းတစ်ခုက တန်းတစ်ခုကျော်တာ။ တန်းတစ်ခု ကျော်တာကို ထပ်ခွဲကြည့်ရင် အပေါ်တက်တာနဲ့ အောက်ဆင်းတာ နှစ်ပိုင်းရှိမယ်။ အခုလို နဂိုမူလကိစ္စတစ်ခုကနေ တစ်ဆင့်ထက်တစ်ဆင့် ပိုသေးငယ်တဲ့ အပိုင်းလေးတွေဖြစ်လာအောင် ခွဲထုတ်တာကို *problem decomposition* လုပ်တယ်လို့ ခေါ်ပါတယ်။

Bottom-up programming နည်းနဲ့ ပရိုဂရမ်ရေးတဲ့အခါ problem decomposition အရင်ဦးဆုံး လုပ်ရတယ်။ ပြီးရင် အောက်ဆုံးအလွှာမှ အသေးဆုံးအပိုင်းလေးတွေကနေ စတင်ဖြေရှင်းတယ်။ ပြီးတဲ့အခါ အဲဒီ အသေးဆုံး အပိုင်းလေးတွေကို အခြေခံအစိတ်အပိုင်းအဖြစ် အသုံးပြုပြီး အောက်ဆုံးအလွှာထက် တစ်ဆင့်မြင့်တဲ့ အပေါ်အလွှာက အပိုင်းတွေကို ဆက်လက်ဖြေရှင်းတယ်။ ဒီအလွှာက အပိုင်းတွေကို အခြေခံအစိတ်အပိုင်းအဖြစ် အသုံးပြုပြီး နောက်ထပ်တစ်ဆင့် ပိုမြင့်တဲ့အလွှာက အပိုင်းတွေကို ဆက်လက်ဖြေရှင်းမှာဖြစ်တယ်။ ဒီလိုမျိုး အောက်ကနေ အပေါ်ကို တစ်လွှာပြီးတစ်လွှာ တက်သွားပြီး ဖြေရှင်းတဲ့ နည်းစနစ်ဟာ bottom-up programming ပါပဲ။

တန်းကျော်ပြေးပွဲမှာ Problem Decomposition လုပ်ထားတာကို တစ်လွှာချင်း ခွဲကြည့်မယ်ဆို



ပုံ ၃.၅

ရင် အခုလိုတွေ့ရမှာ ဖြစ်ပါတယ်။

- တန်းကျော်ပြေးပြိုင်ပွဲဝင်ခြင်း
  - ➡ တန်းတစ်ခုကျော်ခြင်း
    - ➡ အပေါ်တက်ခြင်း
    - ➡ အောက်ဆင်းခြင်း

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

```
def ascend():
    """
    တန်းနဲ့လွှတ်တဲ့အထိ အပေါ်သို့ တက်သည်
    Precondition: တန်းဘယ်ဘက် ခြေရင်းမှာ အရှေ့ဘက်မျက်နှာမူပြီး ရှိနေ
    Postcondition: တန်းထိပ်အပေါ် ဘယ်ဘက်ကွန်နာမှာ အရှေ့ဘက်မျက်နှာမူပြီး ရှိနေ
    """
    turn_left()
    while right_is_blocked():
        move()
    turn_right()
```

```
def descend():
    """
    တန်းအောက်သို့ ပြန်ဆင်းသည်
    Precondition: တန်းထိပ်အပေါ် ညာဘက်ကွန်နာမှာ အရှေ့ဘက်မျက်နှာမူပြီး ရှိနေ
    Postcondition: တန်းညာဘက် ခြေရင်းမှာ အရှေ့ဘက်မျက်နှာမူပြီး ရှိနေ
    """
    turn_right()
    while front_is_clear():
```

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

အထက်ပါ ဖန်ရှင်နှစ်ခုကို အသုံးပြုပြီး အပေါ်တစ်ဆင့် ပိုမြင့်တဲ့ အလွှာက တန်းတစ်ခုကျော်တာကို ဆက်လက်ဖြေရှင်းရပါမယ်။ ascend နဲ့ descend ပရီနဲ့ ပို့စ် ကွန်ဒီရှင်တွေအရ move လုပ်ပေးဖို့လိုပါတယ်။

```
def jump_over_hurdle():
    """
    တန်းတစ်ခုကို ကျော်သည်
    Precondition: တန်းဘယ်ဘက် ခြေရင်းမှာ အရှေ့ဘက်မျက်နှာမူပြီး ရှိနေ
    Postcondition: တန်းညာဘက် ခြေရင်းမှာ အရှေ့ဘက်မျက်နှာမူပြီး ရှိနေ
    """
    ascend()
    move()
    descend()
```

ဒီဖန်ရှင်နဲ့ နောက်တစ်ဆင့်ကို ဆက်လက်တည်ဆောက်ရပါမယ်။ ရှေ့မှာရှင်းနေလျှင် ရှေ့တိုးမယ်၊ မရှင်းလျှင် တန်းကိုကျော်ရမယ်။ ပန်းဝင်ဖို့အတွက်ဆိုလျှင် ဒီအလုပ်ကို ဆယ့်သုံးခါလုပ်ပေးရုံပါပဲ။

```
def compete_hurdle_race():
    """
    တန်းကျော်ပြေးပြိုင်ပွဲ ပန်းဝင်သည်ထိ ယှဉ်ပြိုင်သည်
    Precondition: (၁,၁) ကွန်နာတွင် အရှေ့ဘက်မျက်နှာမူနေ
    Postcondition: (၁၄,၁) ကွန်နာတွင် အရှေ့ဘက်မျက်နှာမူနေ
    """
    for i in range(13):
        if front_is_clear():
            move()
        else:
            jump_over_hurdle()
```

ပရိုဂရမ် အစအဆုံးကို လေ့လာကြည့်ပါ။ ဖန်ရှင်တစ်ခုချင်း ပရီနဲ့ ပို့စ် ကွန်ဒီရှင်တွေကို တိတိကျကျ မြင်အောင် ဂရုစိုက်ကြည့်ဖို့လည်း အရေးကြီးတယ်။ ဖန်ရှင် docstring မှာ ၎င်းဖန်ရှင် ပရီနဲ့ ပို့စ် ကွန်ဒီရှင်ကို တိတိကျကျ ဖော်ပြထားတာဟာ အလေ့အထကောင်း တစ်ခုပါ။ ဖန်ရှင်တိုင်းမှာ ပါသင့်တယ်။

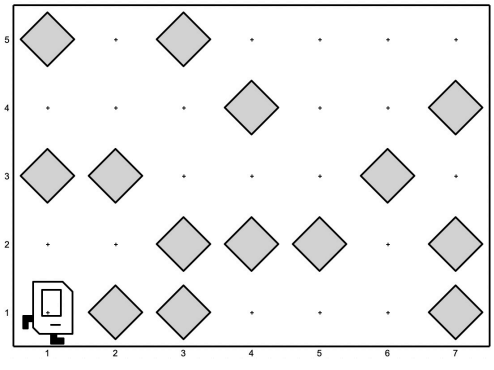
```
# File: hurdle_jumping.py
from stanfordkarel import *


def main():
    """Hurdle Jumping Program"""
    compete_hurdle_race()


def ascend():
    """
```

```
    တန်းနဲ့လွှတ်တဲ့အထိ အပေါ်သို့ တက်သည်
    Precondition: တန်းဘယ်ဘက် ခြေရင်းမှာ အရှေ့ဘက်မျက်နှာမူပြီး ရှိနေ
    Postcondition: တန်းထိပ်အပေါ် ဘယ်ဘက်ကွန်နာမှာ အရှေ့ဘက်မျက်နှာမူပြီး ရှိနေ
    """
    turn_left()
    while right_is_blocked():
        move()
    turn_right()


def descend():
    """
    တန်းအောက်သို့ ပြန်ဆင်းသည်
    Precondition: တန်းထိပ်အပေါ် ညာဘက်ကွန်နာမှာ အရှေ့ဘက်မျက်နှာမူပြီး ရှိနေ
    Postcondition: တန်းညာဘက် ခြေရင်းမှာ အရှေ့ဘက်မျက်နှာမူပြီး ရှိနေ
    """
    turn_right()
    while front_is_clear():
        move()
    turn_left()


def jump_over_hurdle():
    """
    တန်းတစ်ခုကို ကျော်သည်
    Precondition: တန်းဘယ်ဘက် ခြေရင်းမှာ အရှေ့ဘက်မျက်နှာမူပြီး ရှိနေ
    Postcondition: တန်းညာဘက် ခြေရင်းမှာ အရှေ့ဘက်မျက်နှာမူပြီး ရှိနေ
    """
    ascend()
    move()
    descend()


def compete_hurdle_race():
    """
    တန်းကျော်ပြေးပြိုင်ပွဲ ပန်းဝင်သည်ထိ ယှဉ်ပြိုင်သည်
    Precondition: (၁,၁) ကွန်နာတွင် အရှေ့ဘက်မျက်နှာမူနေ
    Postcondition: (၁၄,၁) ကွန်နာတွင် အရှေ့ဘက်မျက်နှာမူနေ
    """
    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

အခုတစ်ခါမှာ ကားရဲလ်က အမှိုက်တွေရှင်းပေးရပါမယ်။ ပုံ (၃.၆) နမူနာ ကမ္ဘာမှာ ကြုံရာကျပန်း (random) ပြန့်ကျဲနေတဲ့ ဘီပါတွေကို အမှိုက်လို့ ယူဆပါ။ ကမ္ဘာအရွယ်အစား အမျိုးမျိုးအတွက် အမှိုက်ရှင်းပေးတဲ့ ပရိုဂရမ်တစ်ခု ရေးပေးရမှာပါ။

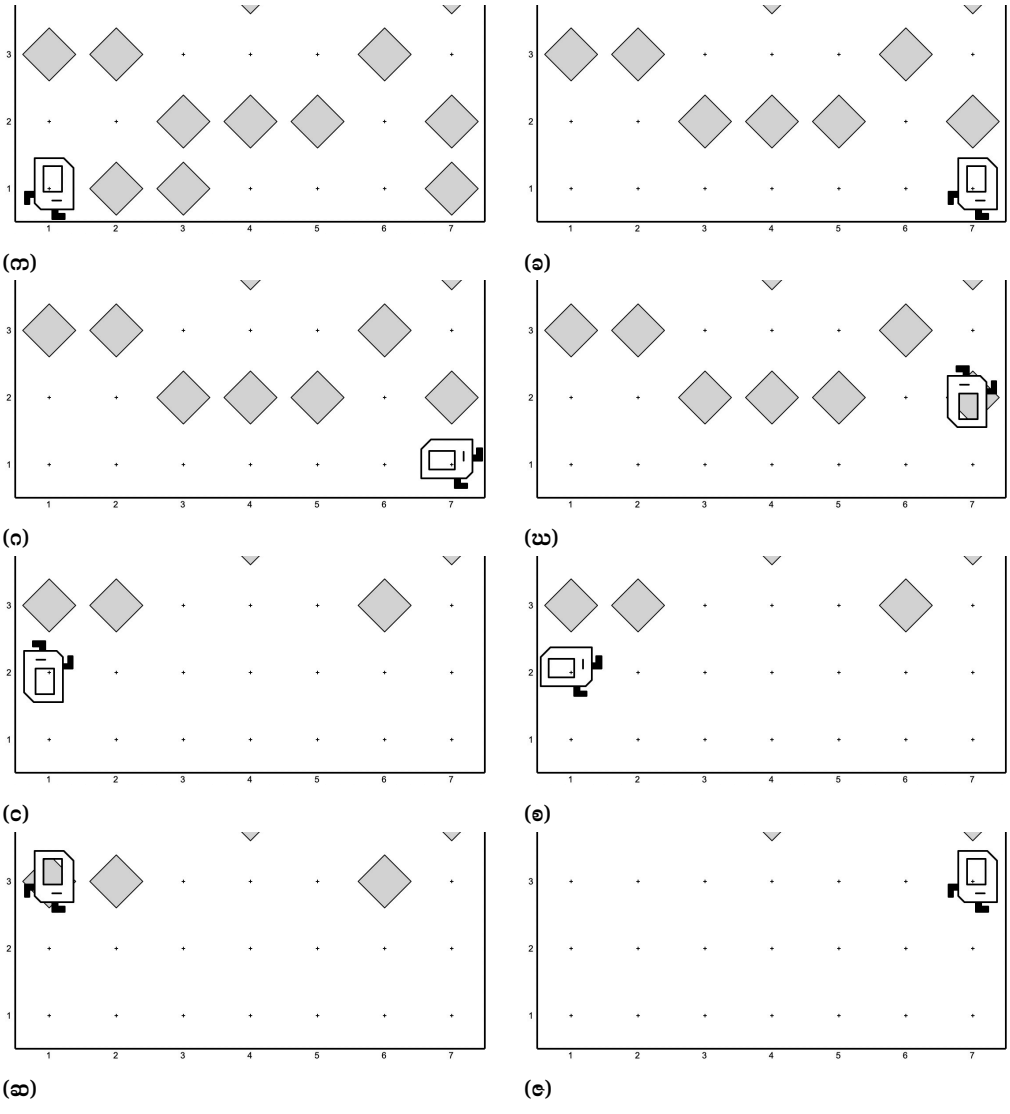


ပုံ ၃.၆

ဒီကိစ္စကို ဖြေရှင်းဖို့က နည်းလမ်းတစ်ခုတည်း ရှိတာ မဟုတ်ပါဘူး။ နည်းလမ်းတစ်ခုက ဒီလိုပါ။ (၁) လမ်းကနေ စပြီး ရှင်းတယ် [[စာမျက်နှာ ၄၀ ပုံ (၃.၇) (က) နှင့် (ခ) တွင် ကြည့်ပါ]]။ ပြီးရင် မြောက်ဘက်လှည့်ထားမယ် [[ပုံ (၃.၇) (ဂ)]]။ ရှင်းနေသေးတယ်ဆိုရင် ဒုတိယလမ်းကို တက်၊ အနောက်ဘက် မျက်နှာမူထားမယ် [[ပုံ (၃.၇) (ဃ)]]။ ဒုတိယလမ်းကို ဆက်ရှင်းပြီးတော့လည်း မြောက်ဘက်ကိုပဲ မျက်နှာမူထားပါမယ် [[ပုံ (၃.၇) (င)၊ (စ)]]။ ရှင်းနေသေးရင် တတိယလမ်းကို ကူး၊ အရှေ့ဘက်မျက်နှာ မူထားမယ် [[ပုံ (၃.၇) (ဆ)]]။ ဒီအတိုင်း တစ်လမ်းပြီးတစ်လမ်း ရှင်းသွားမှာဖြစ်တယ်။ လမ်းတစ်လမ်းရှင်းပြီး မြောက်ဘက်မျက်နှာမူလို့ ပိတ်နေရင် နောက်ထပ် လမ်းမရှိတော့ဘူး။ လမ်းအားလုံး ရှင်းပြီးသွားပြီ။ စာမျက်နှာ ၄၁ ပုံ (၃.၈) (က)၊ (ခ)၊ (ဂ) တို့ကို ကြည့်ပါ။ Problem decomposition လုပ်ကြည့်ရင် အောက်ပါအတိုင်း တွေ့နိုင်ပါတယ်။

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

လမ်းတစ်လမ်း ရှင်းတဲ့ ကိစ္စကို ထပ်ခွဲကြည့်ရင် ကွန်နာတစ်ခုရှင်းတာကို တွေ့ရမှာပါ။ Bottom-up နည်းအရ အောက်ဆုံးအလွှာက ကွန်နာတစ်ခု အမှိုက်ရှင်းတဲ့ ကိစ္စအတွက် ဖန်ရှင်ကို ပထမဆုံး သတ်မှတ်

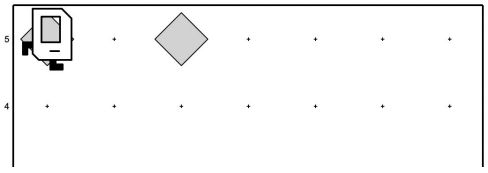


ပုံ ၃.၇

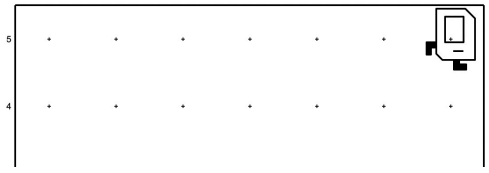
ရပါမယ်။

```
def clean_corner():
    """
    ကွန်နာတစ်ခုမှာ ဘီပါရှင်းပေးသည်
    Precondition: ကွန်နာတစ်ခုမှာ ကားရဲလ် ရှိနေမယ်။ အများဆုံး ဘီပါတစ်ခု ရှိနိုင်တယ်။
    Postcondition: ဘီပါ မရှိတဲ့ ကွန်နာဖြစ်ရမယ်
    """
    if beepers_present():
        pick_beeper()
```

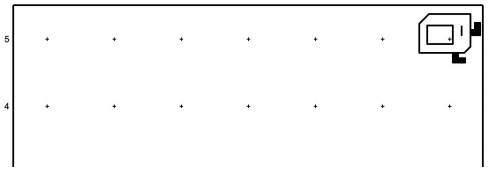
`clean_corner` နဲ့ လမ်းတစ်လမ်းရှင်းတဲ့ ဖန်ရှင် သတ်မှတ်ပါမယ်။



(၃)



(၁)



(၅)

ပုံ ၃.၈

```
def clean_street():
    """
    လမ်းတလျှောက် ဘိပါအားလုံးရှင်းပေးသည်
    Precondition: လမ်းအစွန်းတစ်ဖက်မှာ အခြားဘက်စွန်းကို မျက်နှာမူ၍ ရှိမယ်
    Postcondition: လမ်းပေါ်ဘိပါအားလုံး 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 မျက်နှာမူ မနေသ၍ ဘယ်ဘက်လှည့်ခိုင်းထားတာ သတိထားကြည့်ပါ။ မြောက်ဘက် မျက်နှာမူနေတဲ့ အနေအထားမှာ ရပ်သွားမှာ ဖြစ်တယ်။

လမ်းတစ်လမ်းရှင်းပြီး နောက်တစ်လမ်းကူးတဲ့ ဖန်ရှင်

```
def change_street():
    """
    လမ်းတစ်လမ်း၏ အစွန်းတစ်ဖက်မှာ အပေါ်လမ်းသို့ ကူးသည်
    Precondition: လမ်းအစွန်းတစ်ဖက်မှာ ကပ်လျက် နံရံကို မျက်နှာမူ၍ ရှိမယ်
    Postcondition: အပေါ်တစ်လမ်းကူးပြီး အခြားဘက်စွန်းကို မျက်နှာမူ၍ ရှိမယ်
    """
    move()
    if right_is_blocked():
        turn_left()
    else:
        turn_right()
```

တစ်လမ်းချင်း အမှိုက်အကုန် ရှင်းတဲ့ ဖန်ရှင်ကို သတ်မှတ်လို့ ရပါပြီ

```
def clean_world():
    """
    လမ်းအားလုံးမှ ဘိပါတွေ ရှင်းပေး
    Precondition: (၁,၁) ကွန်နာမှာ အရှေ့ဘက် မျက်နှာမူ၍ ရှိနေမယ်
    Postcondition: လမ်းအားလုံး အမှိုက်ရှင်းပြီး ဖြစ်မယ်
    """
    clean_street()
    turn_north()
    while front_is_clear():
        change_street()
        clean_street()
        turn_north()
```

ပရိုဂရမ် အစအဆုံး ဆက်လက်လေ့လာကြည့်ပါ။ (ပရီနဲ့ ပို့စ် ကွန်ဒီရှင်တွေကို ချန်ထားတယ်။ နေရာသက်သာအောင်ပါ။ 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 တည်ဆောက်သွားမှာပါ။ နည်းလမ်းနှစ်ခုလုံးမှာ problem decomposition လုပ်ရမှာဖြစ်ပေမဲ့ လုပ်ငန်းစဉ် ကွာခြားချက်ရှိတယ်။ Top-down နည်းအတွက် problem decomposition လုပ်တဲ့အခါ bottom-up မှာလို အောက်ဆုံးအလွှာထိ တစ်ခါတည်း ခွဲခြမ်းစိတ်ဖြာစရာမလိုဘူး။ တစ်ဆင့်ချင်းပဲ ခွဲရပါတယ်။ ပြီးခဲ့တဲ့ဥပမာ အမှိုက်သိမ်းတဲ့ ကိစ္စကို ခွဲခြမ်းစိတ်ဖြာမယ်ဆိုရင် အခုလို

- ကမ္ဘာတစ်ခုလုံး အမှိုက်တွေရှင်း (clean world)
  - ➥ လမ်းတစ်လမ်း အမှိုက်ရှင်း (clean street)
  - ➥ မြောက်ဘက်လှည့် (turn north)
  - ➥ လမ်းပြောင်း/နောက်တစ်လမ်းကူး (change street)

ဖြစ်မှာပါ။ စာမျက်နှာ (၃၉) က ခွဲခြမ်းစိတ်ဖြာတာနဲ့ တူတယ်ဆိုပေမဲ့ သတိပြုရမှာက လမ်းတစ်လမ်း ရှင်းတဲ့ကိစ္စကို လောလောဆယ် ထပ်ပြီး အသေးစိတ် မခွဲသေးဘူး။ ဒီအဆင့်မှာပဲ ရပ်ထားတယ်။

ပြီးတဲ့အခါ အပေါ်ဆုံး အဆင့်ကို စပြီးဖြေရှင်းတယ်။ Top-down နည်းရဲ့ အဓိက ထူးခြားချက်က လက်ရှိဖြေရှင်းမဲ့ ကိစ္စရဲ့ အောက်အလွှာကို 'ဖြေရှင်းနိုင်ပြီး' ဖြစ်တယ်လို့ မှတ်ယူရတာပါ။ တစ်နည်းအားဖြင့် `clean_world` ဖန်ရှင် သတ်မှတ်တဲ့အခါ `clean_street`, `turn_north`, `change_street` ဖန်ရှင်တွေ ရှိထားပြီးဖြစ်တယ်လို့ ယူဆရမှာပါ။ ဖန်ရှင်တွေ အမှန်တကယ် မရှိသေးဘဲ ရှိပြီးဖြစ်တယ်လို့ ယူဆရတာဖြစ်တယ်။

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

ဒီနေရာမှာ ပရီနဲ့ ပို့စ် ကွန်ဒီရှင်တွေ တိတိကျကျ သတ်မှတ်ထားဖို့ အလွန်အရေးကြီးပါတယ်။ clean_street, turn_north, change_street ဖန်ရှင်တစ်ခုချင်းရဲ့ ပရီနဲ့ ပို့စ် ကွန်ဒီရှင်တွေ တိတိကျကျ ရှိထားမှပဲ clean_world ကို မှန်ကန်အောင် ရေးဖို့ ဖြစ်နိုင်မှာပါ။ ဥပမာအားဖြင့် clean_street ပို့စ်ကွန်ဒီရှင်ကသာ မြောက်ဘက်လှည့် အနေအထားနဲ့ အဆုံးသတ်မယ်ဆိုရင် clean_world ကို အခုလို ရေးရပါလိမ့်မယ်။

```
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။ တစ်ခုချင်း လိုအပ်သလို ထပ်မံခွဲခြမ်းစိတ်ဖြာ ရပါမယ်။ ဒါပေမဲ့ နောက်ထပ်တစ်ဆင့်ပဲ ခွဲခြမ်းစိတ်ဖြာ ရမှာပါ။ ဆိုလိုတာက ဒုတိယအလွှာကို ဖြေရှင်းတဲ့အခါ တတိယအလွှာထိပဲ ခွဲဖို့ စဉ်းစားတယ်။ တတိယ အလွှာကို ဘယ်လို ခွဲမလဲ ဆက်မစဉ်းစားသေးဘူး။ လမ်းတစ်လမ်း ရှင်းတဲ့ကိစ္စကို ဖြေရှင်းတဲ့အခါ အခုလို

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

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

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

ဒုတိယအလွှာမှာ ပါဝင်တဲ့ မြောက်ဘက်လှည့်တာနဲ့ လမ်းကူးတာကို ဆက်လက်ဖြေရှင်းပါမယ်။ နှစ်ခုလုံး အတန်အသင့် ရိုးရှင်းတဲ့အတွက် ထပ်မခွဲတော့ဘူး။ (လိုအပ်ရင်တော့ အောက်တစ်ဆင့် ထပ်ပြီး ဒီကွန်ပို့စ် 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။

> ညာဘက်လှည့်တာကို လမ်းပြောင်းတဲ့ကိစ္စရဲ့ အခွဲလို့ ယူဆကောင်း ယူဆနိုင်ပါတယ်။ ဒါပေမဲ့ အရမ်းအခြေခံကျတဲ့အတွက် နဂိုပါပြီး turn_left နဲ့ အဆင့်တူလို့ ယူဆရင် ပိုပြီး ကျိုးကြောင်းဆီလျော်မယ်လို့ ယူဆတယ်။ ဒီလိုဖန်ရှင်မျိုးတွေဟာ ဘုံ (common) သုံး ဖြစ်လေ့ ရှိတယ်။ အခြားဖန်ရှင်တွေ (နှစ်ခုနဲ့အထက်) ကနေ ခေါ်သုံးရလေ့ ရှိတယ်။ ကားရဲလ်ပရိုဂရမ် အားလုံးလိုလိုမှာလည်း လိုအပ်လေ့ရှိတယ်။ ဒီသဘောအရ turn_north ကိုလည်း turn_left, turn_right တို့လို အခြေခံ အကျဆုံး ဘုံဖန်ရှင်အနေနဲ့ ယူဆမယ်ဆိုရင်လည်း မမှားပါဘူး။
>
> (ယူဆချက်ကို ဖော်ပြတာသာဖြစ်ပါတယ်။ ဘယ်လိုမှ မှန်တယ်၊ ဘယ်လိုဆိုရင်ဖြင့် မှားတယ် မဆိုလိုပါ။ မိမိကိုယ်တိုင် စဉ်းစားဆင်ခြင် သုံးသပ်ပြီးမှ ဖြစ်သင့်တယ်ထင်တာကို ဆုံးဖြတ်လို့ရပါတယ်)။

အောက်တစ်လွှာ ဆက်ဆင်းရပါမယ်။ ဒုတိယ အဆင့်ကို ဒီကွန်ပျူတာ လုပ်လိုက်တာတွေဟာ တတိယ အလွှာမှာ ပါဝင်တယ်။ ကွန်နာတစ်ခု ရှင်းတဲ့ကိစ္စပဲ ရှိပါတယ်။ ရိုးရှင်းတဲ့အတွက် နောက်တစ်ဆင့် ထပ်မခွဲတော့ဘဲ တစ်ခါတည်း ဖန်ရှင်သတ်မှတ်ပါမယ်။

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

အပေါ်ဆုံးကနေ တစ်ဆင့်ပြီးတစ်ဆင့် ဖြေရှင်းလာတာ တတိယအလွှာမှာ ထပ်မခွဲတော့တဲ့အတွက် ဒီမှာပဲ ပြီးသွားပြီဖြစ်တယ်။ အကယ်၍ ထပ်ခွဲထားတာ ရှိတယ်ဆိုရင် အောက်တစ်ဆင့် ထပ်ဆင်းပြီး ပြီးခဲ့တဲ့ တစ်ဆင့်ချင်းမှာ လုပ်ခဲ့သလိုပဲ ဆက်သွားရမှာပါ။ ပရိုဂရမ် run မယ်ဆိုရင် ကျန်နေသေးတဲ့ turn_right ၊ main နဲ့ အန်ထရီပွိုင့် ဖြည့်ရုံပါပဲ

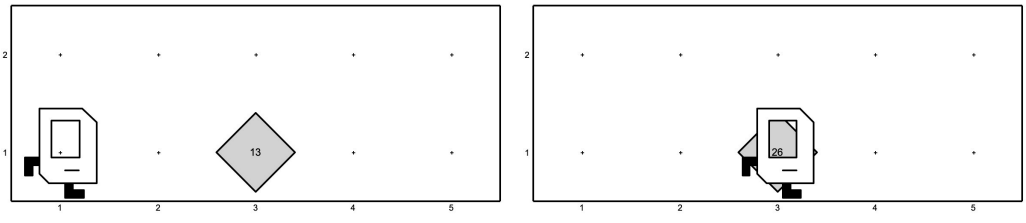
```
def main():
    clean_world()

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

(turn_right ဖန်ရှင်ကို ထပ်မဖော်ပြတော့ပါ)

ဒုတိယ top-down ဥပမာ (Double Beeper Pile)

အမှန်တကယ် မရှိသေးတဲ့ ဖန်ရှင်တွေကို ရှိထားပြီးဖြစ်တယ်လို့ ယူဆထားရတာဟာ top-down နည်းရဲ့ အဓိက သော့ချက်ဖြစ်သလို ပရိုဂရမ်းမင်း စလေ့လာသူတွေအတွက် နားလည်ရ အခက်ဆုံး သဘောတရား တစ်ခုဆိုရင်လည်း မမှားဘူး။ အောက်တစ်ဆင့်အလွှာ ဖန်ရှင်တွေ 'ဘာလုပ်ပေးလဲ'၊ ပရီနဲ့ ပို့စ် ကွန်ဒီရှင်တွေ ဘယ်လိုဖြစ်သင့်လဲ တိတိကျကျ စိတ်ကူးပုံဖော်ထားပြီး လက်ရှိဖန်ရှင်ကို ဘယ်လိုရေးမလဲ စဉ်းစားရတာ ဦးနှောက်ခြောက်စရာ ဖြစ်နေတာပါ။ ဥပမာ နောက်တစ်ခုလောက် ထပ်ကြည့်ရင် ပိုပြီး သဘောပေါက်သွားမယ် ထင်ပါတယ်။

အခုပရိုဂရမ်မှာ ကားရဲလ်က ကွန်နာတစ်ခုမှာ စုပုံထားတဲ့ ဘိပါတွေကို နှစ်ဆဖြစ်အောင် လုပ်ပေးရပါမယ်။ ပုံ (၃.၉) တွင်ကြည့်ပါ။ နဂို ဆယ့်သုံးကနေ နှစ်ဆယ့်ခြောက်ခု ဖြစ်သွားပါတယ်။ ကားရဲလ်က ကိန်းဂဏန်းတွေအကြောင်း နားမလည်သလို ရေတွက်တာလည်း မလုပ်တတ်ပါဘူး။ ဒါကြောင့် ဘိပါနှစ်ဆပွားဖို့ အခြားနည်းလမ်းရှာရပါမယ်။



(ဂ) (ခ)

ပုံ ၃.၉

ဘိပါပုံ (အစုအပုံကို ဆိုလို) ထဲကနေ ဘိပါတစ်ခုကောက်လိုက်၊ ရှေ့ကွန်နာမှာ နှစ်ခုချထားလိုက်ဆက်တိုက် လုပ်မယ်ဆိုရင် နဂိုဘိပါတွေ ကုန်သွားတဲ့အခါ နှစ်ဆရှိတဲ့ ဘိပါပုံတစ်ခု (ရှေ့ကွန်နာမှာ) ရမှာပါ။ တစ်ခါတည်း ဘိပါအားလုံး ကောက်လို့ မရပါဘူး။ ကားရဲလ်က ဘယ်နှစ်ခု ကောက်ထားလဲ မမှတ်မိတဲ့ အတွက်ကြောင့်ပါ။ တစ်ခုကောက်လိုက် ရှေ့ကွန်နာမှာ နှစ်ခုပြန်ချထားလိုက် လုပ်ရပါမယ်။

ရှေ့ကွန်နာမှာ နဂိုဘိပါပုံရဲ့ နှစ်ဆဖြစ်အောင် လုပ်လို့ရတဲ့နည်းတော့ စဉ်းစားလို့ရပြီ။ ပြီးတဲ့အခါ အဲ့ဒီ ဘိပါတွေကို နဂိုဘိပါပုံနေရာကို ရွှေ့ရုံပါပဲ။ ဒီကွန်ပို့စ်လုပ်ကြည့်ရင်

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

ဘိပါပုံဆီကို သွားတာက လွယ်တယ်

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

နဂိုနေရာမှာ ဘိပါပုံ နှစ်ဆလုပ်တဲ့ ကိစ္စကို နောက်တစ်ဆင့် ထပ်ခွဲထားတယ်။ ရှေ့ကွန်နာမှာ နှစ်ဆပုံတာနဲ့ နဂိုနေရာ ပြန်ရွှေ့တာ ကိစ္စနှစ်ခု ပါဝင်တယ်။ ဒီအတွက် ဖန်ရှင်နှစ်ခု ရှိတယ်လို့ မှတ်ယူရပါမယ်။ ၎င်းတို့ လုပ်ဆောင်ချက်က ဘာလဲ၊ ၎င်းတို့ရဲ့ ပရီနဲ့ ပို့စ် ကွန်ဒီရှင်တွေ ဘယ်လိုဖြစ်သင့်လဲ တိတိကျကျ စဉ်းစားထားရမှာပါ။ ဒီအတွက်ကို စိတ်ထဲမှာပဲ လုပ်လို့ရသလို အောက်ပါအတိုင်း ဗလာဖန်ရှင် (*empty*

function) သတ်မှတ်ပြီး docstring နဲ့ တိတိကျကျ ချရေးထားတာဟာလည်း ပရိုဂရမ်မာ အများစု လုပ်လေ့လုပ်ထရှိတဲ့ အလေ့အထတစ်ခုပါ။

```
def double_beeper_pile_next():
    """
    ဘိပါပုံကို ရှေ့ကွန်နာတွင် နှစ်ဆဖြစ်အောင် ရွှေ့ပေးသည်
    Precondition: ဘိပါပုံပေါ်တွင် အရှေ့ဘက် မျက်နှာမူ၍ ရှိနေမယ်
    Postcondition: နှစ်ဆဘိပါပုံပေါ်တွင် အရှေ့ဘက် မျက်နှာမူ၍
    """
    pass
```

```
def move_beeper_pile_next():
    """
    ဘိပါပုံကို ရှေ့ကွန်နာသို့ ရွှေ့ပေးသည်
    Precondition: ဘိပါပုံပေါ်တွင် အနောက်ဘက်မျက်နှာမူ ရှိနေမယ်
    Postcondition: ရှေ့ကို ရွှေ့ပြီးဘိပါပုံပေါ်တွင် အနောက်ဘက်မျက်နှာမူ ရှိနေမယ်
    """
    pass
```

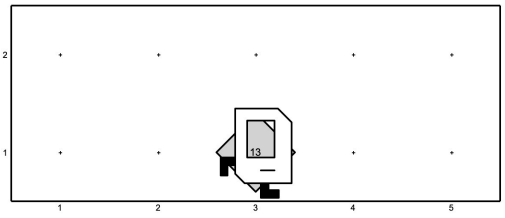
Python မှာ ဗလာဖန်ရှင်ဆိုပေမဲ့ စတိတ်မန့်တစ်ခုတော့ ပါရမယ်။ မဟုတ်ရင် ဆင်းတက်စ် အမှားဖြစ်မှာပါ။ ဒါကြောင့် pass စတိတ်မန့်ကို ယာယီအနေနဲ့ သုံးလေ့ရှိတယ်။ ဒမ်မီစတိတ်မန့် *(dummy statement)* လို့ ခေါ်ပါတယ်။ ဖန်ရှင်နှစ်ခုရဲ့ ပရီနဲ့ ပို့စ် ကွန်ဒီရှင်တွေကို ပုံ (၃.၁၀) နဲ့ (၃.၁၁) မှာ တွေ့နိုင်ပါတယ်။ နှစ်ခုလုံး မျက်နှာမူတဲ့ဘက် မပြောင်းတာကို ဂရုပြုပါ။ ဘိပါပုံက မူလနေရာမဟုတ်ဘဲ ရှေ့ကိုရောက်သွားတာကိုလည်း ဂရုပြုပါ။

မူလနေရာမှာပဲ ဘိပါနှစ်ဆဖြစ်အောင် လုပ်တဲ့ ဖန်ရှင်က ဒီလိုဖြစ်ပါမယ်

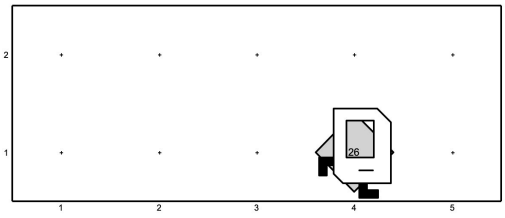
```
def double_beeper_pile():
    """
    ဘိပါပုံကို မူလနေရာတွင်ပင် နှစ်ဆတိုးပေးသည်
    Precondition: ဘိပါပုံပေါ်တွင် အရှေ့ဘက်မျက်နှာမူလျက် ရှိနေ
    Postcondition: နှစ်ဆတိုးပြီး ဘိပါပုံတွင် မူလအတိုင်း အရှေ့ဘက်မျက်နှာမူလျက် ရှိနေ
    """
    double_beeper_pile_next()
    turn_left()
    turn_left()
    move_beeper_pile_next()
    turn_left()
    turn_left()
```

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

အထက်ပါဖန်ရှင်မှာ ဘယ် နှစ်ခါလှည့်ကို နှစ်ကြိမ်လုပ်ထားပါတယ်။ ဆန့်ကျင်ဘက် အရပ်ကို လှည့်ဖို့အတွက် ရည်ရွယ်တာပါ။ ဒီအတွက် turn_around ဖန်ရှင် ရှိသင့်ပါတယ်။ ရှိမယ်ဆိုရင်

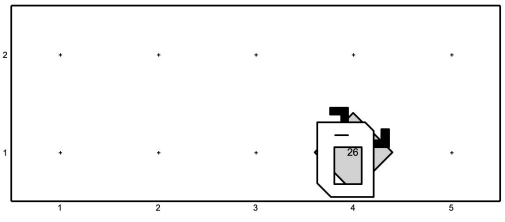


(ဂ)

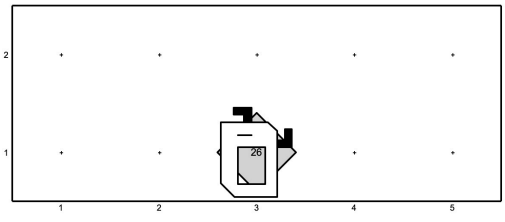


(ခ)

ပုံ ၃.၁၀



(ဂ)



(ခ)

ပုံ ၃.၁၁

```
def double_beeper_pile():
    """
    ဘီပါပုံကို မူလနေရာတွင်ပင် နှစ်ဆတိုးပေးသည်
    Precondition: ဘီပါပုံပေါ်တွင် အရှေ့ဘက်မျက်နှာမူလျက် ရှိနေ
    Postcondition: နှစ်ဆတိုးပြီး ဘီပါပုံတွင် မူလအတိုင်း အရှေ့ဘက်မျက်နှာမူလျက် ရှိနေ
    """
    double_beeper_pile_next()
    turn_around()
    move_beeper_pile_next()
    turn_around()
```

ဒီနေရာမှာ မှတ်သားသင့်တာတစ်ခုက ဒီကွန်ပြူစ် လုပ်တဲ့အခါ တစ်ခါတည်းနဲ့ ပြီးပြည့်စုံတဲ့ ရလဒ် ဖြစ်ချင်မှ ဖြစ်မှာပါ။ တစ်ခါတစ်ရံ ပရိုဂရမ် ရေးနေရင်း စိတ်ကူးသစ် သို့မဟုတ် ပိုကောင်းတဲ့နည်းလမ်း ခေါင်းထဲ ပေါ်လာတတ်ပါတယ်၊ တဖြည်းဖြည်း သဘောပေါက်လာတတ်တယ်။ ဒီအခါမှာ ဒီကွန်ပြူစ် လုပ်တာကို လိုအပ်သလို အလိုက်သင့် ပြောင်းပေးနိုင်ပါတယ်။ အထက်က ဥပမာမှာ `turn_around` လိုအပ်မယ်ဆိုတာ ကြိုမသိခဲ့ပါဘူး။

`double_beeper_pile_next` နဲ့ `move_beeper_pile_next` အတွက် ဆက်လက်စဉ်းစားပါမယ်။ အတန်အသင့် လွယ်ကူမယ် ယူဆတဲ့အတွက် နောက်တစ်ဆင့် ထပ်မခွဲတော့ဘူး။

```
def double_beeper_pile_next():
    """
    ဘီပါပုံကို ရှေ့ကွန်နာတွင် နှစ်ဆဖြစ်အောင် ရွှေ့ပေးသည်
    Precondition: ဘီပါပုံပေါ်တွင် အရှေ့ဘက် မျက်နှာမူ၍ ရှိနေမယ်
    Postcondition: နှစ်ဆဘီပါပုံပေါ်တွင် အရှေ့ဘက် မျက်နှာမူ၍
```

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

    move()
```

```
def move_beeper_pile_next():
    """
    ဘိပါပုံကို ရှေ့ကွန်နာသို့ ရွှေ့ပေးသည်
    Precondition: ဘိပါပုံပေါ်တွင် အနောက်ဘက်မျက်နှာမူ ရှိနေမယ်
    Postcondition: ရှေ့ကို ရွှေ့ပြီးဘိပါပုံပေါ်တွင် အနောက်ဘက်မျက်နှာမူ ရှိနေမယ်
    """
    while beepers_present():
        pick_beeper()
        move()
        put_beeper()
        turn_around()
        move()
        turn_around()

    move()
```

အခုနောက်ဆုံး turn_around, main နဲ့ အန်ထရီပ္ပိုင့် ဖြည့်ဖို့ပဲ ကျန်ပါတယ်။ ပရိုဂရမ် 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()
```

```python
def go_to_beeper_pile():
    while no_beepers_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

ကားရဲလ်နှင့် ရီကားဆစ်ဖ် ဖန်ရှင်များ

ဖန်ရှင်တစ်ခုကနေ 'အခြား' ဖန်ရှင်တွေ ခေါ်သုံးတာကို အခန်း (၃) မှာ တွေ့ခဲ့ပြီးပါပြီ။ ဒါပေမဲ့ ဖန်ရှင်တစ်ခုက ၎င်းကိုယ်၎င်း ပြန်ခေါ်ထားတာကိုတော့ မကြုံဖူးသေးပါဘူး။ ဖန်ရှင်တွေဟာ ဆော့ဖ်ဝဲ အဆောက်အဦးတည်ဆောက်ရာမှာ မရှိမဖြစ်တဲ့ အခြေခံအုတ်ချပ်တွေလို့ ဆိုရမှာပါ။ ၎င်းကိုယ်တိုင်ကို ပြန်လည်အသုံးပြု၍ ဖန်ရှင်အုတ်ချပ်တစ်ခု ဖန်တီးလို့ ရနိုင်ပါမလား။ ဒီမေးခွန်းဟာ ထူးဆန်းကောင်း ထူးဆန်းနေပါလိမ့်မယ်။ အခြေခံကျပြီး စိတ်ဝင်စားစရာကောင်းတဲ့ ဖီလော်ဆော်ဖီ မေးခွန်းလည်း ဖြစ်တယ်။

ဖန်ရှင် သတ်မှတ်ချက်ထဲမှာ ၎င်းဖန်ရှင်ကိုယ်တိုင်ကို ပြန်ခေါ်လို့ ရပါတယ်။ ရီကားဆစ်ဖ် ဖန်ရှင် (*recursive function*) လို့ ခေါ်တယ်။ ရီကားဆစ်ဖ် ဖန်ရှင်တွေဟာ ဘီဂင်နာ ပရိုဂရမ်မာအတွက် နားလည်ဖို့ ခက်ခဲတဲ့ သဘောတရားအဖြစ် ယူဆကြတာကြောင့် စာအုပ် အတော်များများမှာ နောက်ကျပြီး ဖော်ပြလေ့ရှိတယ်။ တကယ်က လူများစု ထင်/ပြောသလို နားမလည်နိုင်လောက်အောင် ရှုပ်ထွေး ခက်ခဲတဲ့ သဘောတရား မဟုတ်ပါဘူး။ သာမန်လူအားလုံး နားလည်နိုင်ပါတယ်။ ဒါကြောင့် စောစောစီးစီး အခုပဲ မိတ်ဆက်ပေးလိုက်ပါတယ်။ အကယ်၍ နားမလည်ခဲ့ရင်လည်း ပြဿနာမရှိပါဘူး။ အကြမ်းဖျဉ်းလောက် ဖတ်ကြည့်ပြီး နောက်လာမဲ့ အခန်းတွေကို ကျော်သွားနိုင်ပါတယ်။ နောင်တစ်ချိန်ကျမှ ပြန်လာဖတ်ပေါ့။

၄.၁ ရီကားဆစ်ဖ် ဖန်ရှင် ဘယ်လို အလုပ်လုပ်လဲ

ရီကားဆစ်ဖ် ဖန်ရှင် ဥပမာတစ်ခုကို လေ့လာကြည့်ပါမယ်။ အောက်ဖော်ပြပါ ဖန်ရှင်ဟာ ၎င်းကိုယ်တိုင်ကို ၎င်း ပြန်ခေါ်ထားတာ ဂရုပြုကြည့်ပါ။ *recursive call* လို့ ကွန်းမန့်ရေးထားတဲ့ လိုင်းမှာပါ။

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

ဒီဖန်ရှင်ကို ခေါ်လိုက်ရင် ဘာဆက်ဖြစ်မလဲဆိုတာ စိတ်ဝင်စားစရာပါ။ ဖန်ရှင်မစတင်မီ အနေအထားကို ပုံ (၄.၁) မှာ ကြည့်ပါ။ ဖန်ရှင်ကို ကနဦး စခေါ်လိုက်တာမို့လို့ *initial call* လို့ ရည်ညွှန်းပါမယ်။

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

ပုံ ၄.၁

ဖန်ရှင် စတင် လုပ်ဆောင်ပါမယ်။ ရှေ့မှာရှင်းနေတဲ့ အတွက် if ဘလောက်ကို လုပ်မှာပါ

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

ဘိပါချ။ ရှေ့တိုး [[ပုံ (၄.၂) (က) ကြည့်ပါ]] ပြီးရင် သူ့ကိုယ်သူ ပြန်ခေါ်ထားတယ်။ ဒါဟာ ပထမဆုံး တစ်ကြိမ်ပါ။ ဖန်ရှင်ခေါ်ရင် ဖြစ်မြဲအတိုင်းပဲ ဖန်ရှင်နဲ့ သက်ဆိုင်တဲ့ ဘလောက်ကို ဆောက်ရွက်တာပေါ့။ ဒီတော့ make_beeper_row ဖန်ရှင်ဘလောက်ကိုပဲ တစ်ခါထပ်လုပ်မှာပါ။ ရှေ့မှာ ရှင်းနေတဲ့အတွက် if အပိုင်းကို လုပ်တယ်

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

ဘိပါချ။ ရှေ့တိုး [[ပုံ (၄.၂) (ခ)]] ပြီး သူ့ကိုယ်သူ ပြန်ခေါ်ထားတဲ့ကိစ္စ တစ်ခါထပ်ဖြစ်ပြန်တယ်။ ဒါနဲ့ဆိုနှစ်ကြိမ်။ ဖန်ရှင်ဘလောက် အလုပ် ပြန်လုပ်မယ်။ ရှေ့မှာရှင်းတယ်၊ if ကိုပဲ ထပ်လုပ်

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

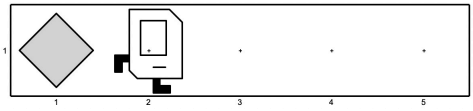
ပုံ (၄.၂) (ဂ) နေရာရောက်ပြီး သူ့ကိုယ်သူ ထပ်ခေါ်ထားပြန်တယ်။ သုံးကြိမ်ရှိပြီ။ ဒီတစ်ခါလည်း if အပိုင်းပဲ ထပ်လုပ်

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

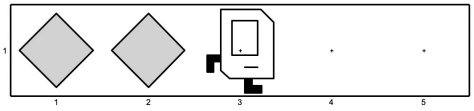
ရှေ့တိုးပြီးရင် နံရံပိတ်နေပြီ [[ပုံ (၄.၂) (ဃ)]]။ သူ့ကိုယ်သူ ခေါ်တယ်။ ရှေ့မှာ ပိတ်နေတဲ့အတွက် else အပိုင်းကို လုပ်မှာပါ။

```
put_beeper()
```

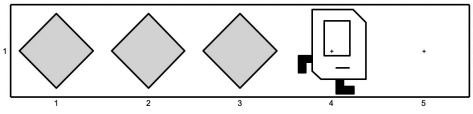
သူ့ကိုယ်သူ ပြန်ခေါ်တဲ့ကစ္စ ထပ်မဖြစ်တော့ဘူး။ ဒီမှာပဲ ပြီးဆုံးသွားတယ်။ ရီကားဆစ်ဖ် ဖန်ရှင် အလုပ်လုပ်ပုံ အခြေခံသဘောတရားက ဒါပါပဲ။ loop တွေ မသုံးဘဲ ပြန်ကျော့နေတဲ့ သဘောကို ရီကားဆစ်ဖ် ဖန်ရှင်မှာ တွေ့ရပါတယ်။ ဖန်ရှင်က သူ့ကိုယ်သူ (သို့ ကိုယ့်ကိုကိုယ်) ပြန်ခေါ်တာကို *recursive call* လို့



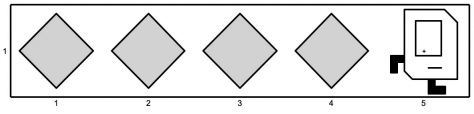
(က) ပထမ တစ်ကျော့ပြီး



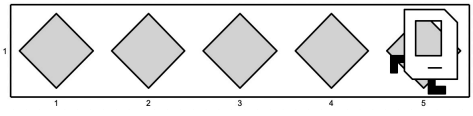
(ခ) ဒုတိယ တစ်ကျော့ပြီး



(ဂ) တတိယ တစ်ကျော့ပြီး



(ဃ) စတုတ္ထမြောက် ကျော့ပြီး



(င) နောက်ဆုံး putBeeper လုပ်ပြီး

ပုံ ၄.၂

ခေါ်ပါတယ်။ ဒက်ဖ်နေးရှင်းအရ recursive function တွေမှာ recursive call အနည်းဆုံး တစ်ခု ပါရမှာ ဖြစ်ပါတယ်။

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

make_beeper_row()
```

ရီကားဆစ်ဖ်ကောလ် ဖြစ်တာကို စိတ်ကူးပုံဖော်ကြည့်ဖို့ ပြထားတာပါ။ အပြင်ဆုံး မြှားအနက်က ကနဦး ဖန်ရှင်ကောလ် စတင်တာဖြစ်ပေါ်တာကို ဖော်ပြတယ်။ ရီကားဆစ်ဖ်ကောလ် မဟုတ်သေးဘူး။ မြှားအပြာက ရီကားဆစ်ဖ်ကောလ်ကြောင့် ဖန်ရှင်အစ ပြန်ရောက်သွားတာကို ပြတယ်။ ပထမနဲ့ ဒုတိယ ရီကားဆစ်ဖ်ကောလ် နှစ်ခုအတွက်ပြထားတာပါ။ ရှေ့က ဥပမာအတွက် မြှားအပြာ လေးခု ရှိရမှာပါ (ရီကားဆစ်ဖ်ကောလ် လေးကြိမ်အတွက်)။ မြှားအပြာ နောက်ထပ် နှစ်ခုရှိတယ် မှတ်ယူပါ (ပုံမှာထပ်ထည့်ရင် ကြပ်ညပ်ပြီး ကြည့်ရရှုပ်လို့ မဆွဲပြတာ)။ လေးကြိမ်မြောက်မှာ ရီကားဆစ်ဖ်ကောလ် ထပ်မဖြစ်တော့ဘူး (`if` အပိုင်းကို မလုပ်တော့တဲ့အတွက်)။ ဘိပါချုပ်ပြီး ဖန်ရှင်ကောလ် စခဲ့တဲ့နေရာကို ပြန်ရောက် သွားမယ် (မြှားအနီ)။ ဘယ်လို ပြန်ရောက်သွားတာလဲ ဆက်ကြည့်ရအောင်။

နောက်ဆုံး ရီကားဆစ်ဖ်ကောလ်မှ ပြန်လာခြင်း

နောက်ဆုံး ရီကားဆစ်ဖ်ကောလ်ကနေ မူလ ဖန်ရှင်ခေါ်ခဲ့တဲ့ နေရာကို ဘယ်လိုပြန်ရောက်သွားတာလဲ။ ဒီကိစ္စနားလည်ဖို့ ဖန်ရှင် *return* ပြန်ခြင်းအကြောင်း အရင်ကြည့်ရပါမယ်။ ဖန်ရှင်ကောလ် လုပ်ဆောင်တဲ့အခါ 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 ကောလ် လုပ်ဆောင်တဲ့အခါ ကောလ်လုပ်တဲ့ နေရာကနေ turn_right ဖန်ရှင်ထဲကို jump လုပ်ပြီး ရောက်သွားတယ်။ မြှားအနက်နဲ့ ပြထားတယ်။ ဖန်ရှင်ဘလောက် လုပ်ဆောင်ပြီးတဲ့အခါ ခေါ်ခဲ့တဲ့နေရာ main ဖန်ရှင်ထဲ ပြန်ရောက်သွားတယ် (မြှားအနီ)။ 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 ဖန်ရှင်ကောလ်တွေမှာလည်း ဒီသဘောတရား အတိုင်းပါပဲ။ အောက်ဖော်ပြပါ ပရိုဂရမ်ကုဒ်ကို ကြည့်ပါ။ main ဖန်ရှင်ထဲကနေ do_tricks ဆီကို ရောက်သွားမယ်။ do_tricks ထဲကနေ put_two ထဲကို ရောက်သွားမယ်။

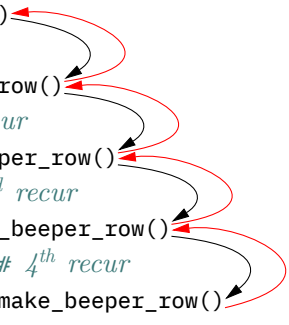
```
def main():
    do_tricks()
    move()

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 ဆက်လုပ်ပြီး 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 ဆီကို အရင် return ပြန်တယ်။ ပြီးတော့မှ do_tricks ကနေ main ကို ပြန်ရောက်မှာပါ။ ဒီသဘောအရ put_two return မပြန်မချင်း do_tricks ဖန်ရှင်မပြီးသေးဘူး။ put_two ကနေ ပြန်လာပြီးမှ ကျန်တဲ့ turn_left, move ဆက်လုပ်တယ်။ ပြီးတော့မှ do_tricks ဖန်ရှင်က return ပြန်ပါတယ်။

ရီကားဆစ်ဖ် ဖန်ရှင်ကောလ်တွေ ဘယ်လို return ပြန်လဲ။ ရှေ့ကလို မြှားတွေနဲ့ ဆွဲပြလို့ ရပေမဲ့ ကြည့်ရတာ ရှုပ်ရှက်ခတ်နေမှာပါ။ အခုလို မြင်ကြည့်ရင် ပိုရှင်းပါတယ်။

```
# initial call
make_beeper_row()
    # 1st recur
    make_beeper_row()
        # 2nd recur
        make_beeper_row()
            # 3rd recur
            make_beeper_row()
                # 4th recur
                make_beeper_row()
```



မြှားအနက်တွေက ဖန်ရှင်ကောလ် တစ်ဆင့်ပြီးတစ်ဆင့် ဖြစ်တာကို ပြတာပါ။ အထက်မှအောက် အစီအစဉ်အတိုင်း ဖြစ်ပါတယ်။ လေးကြိမ်မြောက်မှာ နောက်ထပ် ရီကားဆစ်ဖ်ကောလ် ထပ်မဖြစ်တော့ဘဲ နောက်ဆုံး ရီကားဆစ်ဖ်ကောလ်က အရင်ဆုံး return စပြန်ပါတယ် (အောက်ဆုံး မြှားအနီနဲ့ ပြထား)။ ဒီအခါ တတိယ ရီကားဆစ်ဖ်ကောလ်ကို ပြန်ရောက်သွားမှာပါ။ ဒီအတိုင်း တစ်ဆင့်ပြီးတစ်ဆင့် အထက်ကို return ပြန်ပြီး နောက်ဆုံးမှာ ပထမဆုံး ခေါ်ခဲ့တဲ့နေရာ ပြန်ရောက်သွားမှာပါ။ (အခုပြထားတာကို တကယ့် Python ကုဒ် အနေနဲ့ မယူဆရပါ။ ဖြစ်စဉ် နားလည်အောင် ပြခြင်းသာဖြစ်ပါတယ်)။ နဲနဲပြင်ထားတဲ့ make_beeper_row ဗားရှင်းမှာ return ပြန်တဲ့ ဖြစ်စဉ်ကို ကြည့်ရအောင်။

```
make_beeper_row()
```

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

if အပိုင်း ရီကားဆစ်ဖ်ကောလ်ပြီး turn_left နဲ့ move ထပ်ဖြည့်ထားတာ။ ရီကားဆစ်ဖ်ကောလ် return ပြန်လာပြီးမှ ဒီနှစ်ခု ဆက်လုပ်မှာပါ။ နောက်ဆုံး ရီကားဆစ်ဖ်ကောလ်က return စဖြစ်တယ်။ ဒီ

တော့မှ တတိယအောက် turn_left နဲ့ move ကို လုပ်ဆောင်မှာပါ။ ပြီးမှ တတိယကောလ် return ပြန်တယ်။ ဒီအတိုင်း အထက်ကို တက်သွားပြီး ကျန်နေသေးတဲ့ စတိတ်မန့်တွေကို လုပ်ဆောင်ပါတယ်။ ပထမဆုံးကောလ်အောက် ကျန်နေတာတွေ နောက်ဆုံးကျမှ ပြီးမှာပါ။

```
# initial call
make_beeper_row()
    # 1st recur
    make_beeper_row()
    turn_left()
    move()
        # 2nd recur
        make_beeper_row()
        turn_left()
        move()
            # 3rd recur
            make_beeper_row()
            turn_left()
            move()
                # 4th recur
                make_beeper_row()
```

ရီကားဆစ်ဖ်ကောလ် နှစ်ခါပဲ ဖြစ်မယ်ဆိုရင် အောက်ပါအတိုင်း မြင်ကြည့်လို့ ရပါတယ်။ မြှားအပြာနှစ်ခုက ရီကားဆစ်ဖ်ကောလ်ဖြစ်တာ။ ဒုတိယကောလ်က ဘိပါချပြီး (else အပိုင်း) အရင် return မယ်။ အထက်ကို ညွှန်တဲ့ မြှားအနီကို ကြည့်ပါ။ ဘယ်လှည့်၊ ရှေ့တိုးပြီး ပထမ ကောလ်က နောက်မှ initial call လုပ်ခဲ့တဲ့ဆီ ပြန်ရောက်တာ။ အောက်ကိုညွှန်တဲ့ မြှားအနီကို ကြည့်ပါ။

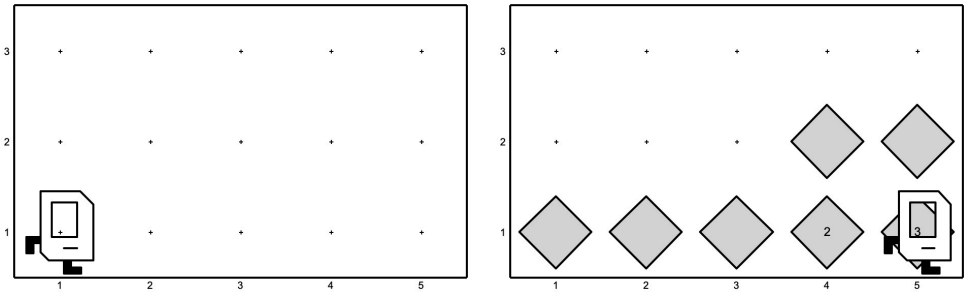
```
def make_beeper_row():
    if front_is_clear():
        put_beeper()
        move()
        make_beeper_row()
        turn_left()
        move()
    else:
        put_beeper()

# initial call
make_beeper_row()
```

ရီကားဆစ်ဖ် ဖန်ရှင်အကြောင်း လေ့လာတဲ့အခါ ဘီဂင်နာအများစု ကြေကြေညက်ညက် နားလည်ဖို့ အတွက် အခက်အခဲဆုံးတစ်ခုက return ပြန်တဲ့ သဘောတရားပါပဲ။ များများစဉ်းစား၊ 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 မတိုင်မီနဲ့ ပြီးနောက် အခြေအနေပါ။ နောက်ဆုံးမှာ ဘိပါလေးခုကို ပါတ်လည် ဘယ်လိုချသွားလဲ စဉ်းစားကြည့်ပါ။ ရှေ့မှာဖော်ပြခဲ့သလို ဖန်ရှင်ကောလ်တွေ တစ်ဆင့်ပြီးတစ်ဆင့် ဖြစ်ပုံနဲ့ return ဖြစ်ပုံကို မြှားဆွဲကြည့်ပါ။



(က) (ခ)

ပုံ ၄.၃

၄.၂ ရီကားဆစ်ဖ်နည်းဖြင့် ပုစ္ဆာဖြေရှင်းခြင်း

ရှေ့ စက်ရှင်မှာ လေ့လာခဲ့တာက ရီကားဆစ်ဖ်ဖန်ရှင် ဘယ်လို အလုပ်လုပ်လဲပဲ ရှိပါသေးတယ်။ တစ်နည်း အားဖြင့် မက္ကနစ်ဇမ် (mechanism) ကို လေ့လာတာပါ။ အခုတစ်ခါ ရီကားဆစ်ဖ်ဖန်ရှင်တွေနဲ့ ပုစ္ဆာတွေ ဘယ်လိုဖြေရှင်းမလဲ ဆက်လက်လေ့လာပါမယ်။ 'ရီကားဆစ်ဖ် စဉ်းစားခြင်း' (thinking recursively) သို့မဟုတ် 'ရီကားဆစ်ဖ်နည်းဖြင့် ပုစ္ဆာဖြေရှင်းခြင်း' (solving problems recursively) ကို လေ့လာမှာပါ။

ရီကားဆစ်ဖ် စဉ်းစားခြင်း ဥပမာ (၁)

ကွန်နာမှာရှိတဲ့ ဘိပါတွေအားလုံး ကောက်မယ်ဆိုပါစို့။ ဘိပါတစ်ခုနဲ့ အထက်ရှိနိုင်တယ်။ ဘိပါမရှိတာလည်း ဖြစ်နိုင်တယ် ယူဆပါ။ ဒီအတွက် ရီကားဆစ်ဖ်ဖန်ရှင် သတ်မှတ်ပါမယ်။

```
def pick_all_beepers():
    ... # to do soon
```

ဖြေရှင်းမဲ့ကိစ္စတစ်ခုကို ၎င်းကိုယ်တိုင်နဲ့ ပုံပန်းသဏ္ဌာန်တူပြီး အရွယ်အစားအားဖြင့် တစ်ဆင့်ထက် တစ်ဆင့် သေးငယ်တဲ့ ကိစ္စတွေအဖြစ် ခွဲခြမ်းကြည့်ပါတယ်။ ဥပမာ ဘိပါငါးခုရှိတဲ့ ကိစ္စကို ဘိပါလေးခု၊ သုံးခု၊ နှစ်ခု နဲ့ တစ်ခု ရှိတဲ့ ကိစ္စတွေအဖြစ် ခွဲပြီး မြင်ကြည့်ရမှာပါ။ ဒီကိစ္စမှာ ဘိပါအရေအတွက်ဟာ အရွယ်အစားပဲ။ လေးခုရှိတဲ့ ကိစ္စဟာ ငါးခုရှိတဲ့ကိစ္စထက် အရွယ်အားဖြင့် တစ်ဆင့်ငယ်တာပေါ့။ ဘိပါမရှိတာလည်း ဖြစ်နိုင်တော့ သုညဘိပါဟာ အငယ်ဆုံးဖြစ်တယ်လို့ ယူဆနိုင်တယ်။

`pick_all_beepers` ဖန်ရှင်ဟာ လက်ရှိဖြေရှင်းမဲ့ အရွယ်အစားထက် တစ်ဆင့်ငယ်တဲ့ ကိစ္စကို ဖြေရှင်းနိုင်ပြီးသားလို့ မှတ်ယူရပါမယ်။ လက်ရှိဖြေရှင်းမဲ့ ကိစ္စက ဘိပါငါးခု ကောက်ရမယ်ဆိုရင် ဘိပါလေးခု ကောက်နိုင်ပြီးသားလို့ ယူဆရမှာပါ။ n ဘိပါရှိတယ် ဆိုရင် $n-1$ ဘိပါကို ကောက်နိုင်ပြီးသား ယူဆရမယ်။ Top-down နည်းမှာလည်း ဒီလိုပဲ မရှိသေးဘဲ၊ မလုပ်နိုင်သေးဘဲ ရှိတယ်၊ လုပ်နိုင်တယ် မှတ်ယူပြီး စဉ်းစားဖြေရှင်းတာကို ပြန်အမှတ်ရမှာပါ။ ရီကားဆစ်ဖ် စဉ်းစားတဲ့အခါမှာ လက်ရှိသတ်မှတ်နေတဲ့ ဖန်ရှင်ကိုယ်တိုင်ကို ရှိပြီးဖြစ်တယ်လို့ သဘောထားရတာ။

ပြီးတဲ့အခါ လက်ရှိကိစ္စကနေ သူ့ထက်တစ်ဆင့်ငယ်တဲ့ ကိစ္စဖြစ်သွားအောင် ဘာလုပ်ရမလဲ စဉ်းစားရတယ်။ ဘိပါငါးခုကနေ လေးခုဖြစ်အောင် ဘိပါတစ်ခု ကောက်ရမှာပေါ့။ ယျေဘုယျပြောရင် n ဘိပါရှိရင် ဆိုရင် $n-1$ ဘိပါဖြစ်သွားအောင် ဘာလုပ်ရမလဲ စဉ်းစားတာ။

လက်ရှိဖန်ရှင်ဟာ $n-1$ ဘိပါကို ကောက်နိုင်ပြီးသားလို့ ယူဆထားတယ်။ n ဘိပါရှိရင် ဘိပါတစ်ခု ကောက်လိုက်ရင် $n-1$ ဘိပါဖြစ်သွားမယ်။ ကျန်နေတဲ့ $n-1$ ဘိပါကောက်ဖို့ လက်ရှိဖန်ရှင်ကိုပဲ ပြန်ခေါ်လိုက်မှာပေါ့။

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

ရီကားဆစ်ဖ် စဉ်းစားတတ်ဖို့ ဒီအဆင့်က အခရာအကျဆုံးပဲ။ တစ်ဆင့်ငယ်တဲ့ ကိစ္စ $n-1$ ကို ဖြေရှင်းနိုင်ပြီးသားလို့ မှတ်ယူပြီး လက်ရှိကစ္စ n ကို ဘယ်လို ဖြေရှင်းမလဲ စဉ်းစားသွားတာ။

ဖန်ရှင်သတ်မှတ်ချက်က မပြီးသေးပါဘူး။ အသေးငယ်ဆုံး ကိစ္စကို ချင်းချက်အနေနဲ့ စဉ်းစားရမယ်။ အသေးငယ်ဆုံးကိစ္စက ဘိပါမရှိတာ (သုညပ) ဖြစ်တယ်။ ဘိပါမရှိရင် ဘာမှလုပ်စရာမလိုဘူး။ n နဲ့ $n-1$ ဘိပါအတွက် အထက်ပါအတိုင်း စဉ်းစားတဲ့အခါ $n \neq 0$ လို့ ယူဆရမှာပါ။ ဒါကြောင့် ဘိပါရှိမှပဲ လုပ်အောင် အခုလိုဖြစ်ရပါမယ်

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

ရီကားဆစ်ဖန်ရှင် မှန်မမှန် စစ်ဆေးခြင်း

ရီကားဆစ်ဖန်ရှင် တက်စ် (test) လုပ်ရင် အသေးဆုံးကိစ္စကနေ စရတယ်။ ပြီးခဲတဲ့ ဖန်ရှင်အတွက် အသေးဆုံးက သုညပါ။ ဘိပါမရှိရင် ဖန်ရှင်က မှန်ရဲ့လား အရင်ဆုံး စစ်ကြည့်ပါမယ်။

```
# assume no beeper
pick_all_beepers()
```

`if` ဘလောက် မလုပ်ဆောင်ဘဲ return ဖြစ်သွားမှာပါ (ရီကားဆစ်ဖ်ကောလ် မဖြစ်လိုက်ဘူး)။ သုညဘိပါအတွက် ဖြစ်သင့်တဲ့အတိုင်း ဖြစ်ပါတယ်။ ဘိပါတစ်ခုပဲ ရှိရင်ရော ဘယ်လိုဖြစ်မလဲ။ ဘိပါကောက်တယ် သုညဘိပါဖြစ်သွားပြီး ရီကားဆစ်ဖ်ကောလ် ဖြစ်မယ်။ တစ်ကြိမ်ပဲ ဖြစ်မယ်။ `if` ဘလောက် အလုပ်မလုပ်ဘဲ return ပြန်မယ်။

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

သုံးခုရှိရင် ရီကားဆစ်ဖ်ကောလ် နှစ်ခါဖြစ်ပြီးမှ return ပြန်မှာပါ။

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

မျှော်လင့်ထားသလို အလုပ်လုပ်နေပါတယ်။ အထက်ပါအတိုင်း စစ်ကြည့်သွားရင် ဘိပါ သုံး၊ လေး၊ ငါး၊ ... ခု တွေအတွက်လည်း တောက်လျှောက်မှန်နေမှာပါ။ ဘာကြောင့် ပြောနိုင်ရတာလဲ။ အခုလို စဉ်းစားကြည့်နိုင်ပါတယ်။ ရှေ့မှာ စစ်ကြည့်တာ $n = 0, n = 1$ အတွက် မှန်တာ သေချာပြီ။ $n = 2$ အတွက် စစ်မယ်ဆိုပါစို့

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

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

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

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

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

ဒီတိုင်းဆက်သွားရင် သုညအပါအဝင် မည်သည့် အပေါင်းကိန်းပြည့် n အတွက်မဆို (non-negative integer) မှန်တယ်ဆိုတာ မြင်နိုင်ပါတယ်။

ရီကားဆစ်ဖ် စဉ်းစားခြင်း ဥပမာ (၂)

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

> ဖြေရှင်းမဲ့ကိစ္စကို ၎င်းကိုယ်တိုင်နဲ့ သဏ္ဌာန်တူပြီး အရွယ်အစားအားဖြင့် တစ်ဆင့်ထက်တစ်ဆင့် သေးငယ်တဲ့ ကိစ္စတွေအဖြစ် ခွဲခြမ်းကြည့်ရမှာပါ။

လမ်းတစ်ခုရဲ့ အရှည်ကို အရွယ်အစားလို့ ယူဆနိုင်တယ်။ တစ်နည်းအားဖြင့် လမ်းတစ်လျှောက် ကွန်နာအရေအတွက်ဟာ အခုဖြေရှင်းမဲ့ ကိစ္စရဲ့ အရွယ်အစားပဲ။ ကွန်နာတစ်ခုတော့ အနည်းဆုံး ရှိရမယ်။ ဒါကြောင့် အသေးဆုံးက $n = 1$ ဖြစ်တယ်။

> `clean_street` ဖန်ရှင်ဟာ လက်ရှိဖြေရှင်းမဲ့ အရွယ်အစားထက် တစ်ဆင့်ငယ်တဲ့ ကိစ္စကို ဖြေရှင်းနိုင်ပြီးသားလို့ မှတ်ယူရပါမယ်။

အခုကိစ္စအတွက် ကွန်နာငါးခု ရှင်းမယ်ဆိုရင် `clean_street` ဖန်ရှင်ဟာ ကွန်နာလေးခုနဲ့ လမ်းကို ရှင်းနိုင်တယ်လို့ ယူဆရမယ်။ n ကွန်နာရှိတဲ့လမ်းအတွက် $n - 1$ ကွန်နာကို ရှင်းနိုင်ပြီးသားလို့ ယူဆရမှာ ဖြစ်ပါတယ်။

> လက်ရှိအရွယ်အစားကို သူ့ထက်တစ်ဆင့်ငယ်တဲ့ ကိစ္စဖြစ်သွားအောင် ဘာလုပ်ရမလဲ စဉ်းစားရတယ်။ ကွန်နာငါးခု လမ်းကိုရှင်းတဲ့အခါ ကွန်နာ လေးခုပဲ ရှင်းစရာကျန်အောင် ဘာလုပ်မလဲ။ ယျေဘုယျပြောရင် n ကွန်နာဆိုရင် $n - 1$ ကွန်နာ ရှင်းဖို့လိုတော့အောင် ဘာလုပ်မလဲ။

လမ်းတစ်လမ်းရှင်းတဲ့အခါ လမ်းအစ သို့မဟုတ် လမ်းအဆုံးမှာ အခြားဘက်စွန်းကို မျက်နှာမူထားတယ်။ ရှေ့တစ်ကွန်နာ ရွှေ့လိုက်ရင် ရှင်းစရာ ကွန်နာတစ်ခု လျော့သွားမှာပါ။

> တစ်ဆင့်ငယ်တဲ့ ကိစ္စ $n - 1$ ကို ဖြေရှင်းနိုင်ပြီးသားလို့ မှတ်ယူပြီး လက်ရှိကစ္စ n ကို ဘယ်လို ဖြေရှင်းမလဲ စဉ်းစားရပါမယ်။

ကွန်နာငါးခုရှင်းမယ်ဆိုရင် လေးခုကို ရှင်းနိုင်ပြီးသား မှတ်ယူထားရမယ်။ ဘိပါရှိရင်ကောက်၊ ရှေ့တစ်ကွန်နာ ရွှေ့ထားလိုက်ရင် ရှင်းစရာ လေးခုပဲကျန်မယ်။ ဒီလေးခုကို လက်ရှိသတ်မှတ်နေတဲ့ ဖန်ရှင်နဲ့ ရှင်းလိုက်ရုံပဲပေါ့။

```
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 ဘယ်လိုဖြေရှင်းမလဲ။

ကွန်နာတစ်ခုပဲ ရှိတယ်ဆိုရင် ဘိပါရှိရင် ကောက်လိုက်ရုံပါပဲ။ n နဲ့ $n-1$ 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။ ကွန်နာနှစ်ခု ရှိတယ်ဆိုရင် စစချင်း ရှေ့မှာ နံရံမရှိဘူး။ if အပိုင်း အလုပ်လုပ်မယ်။ ဘိပါရှိရင် ကောက်တယ်၊ ရှေ့တိုးတယ် (နံရံပိတ်သွားပြီ)။ ရီကားဆစ်ဖ်ကောလ် ဖြစ်တယ်။ else အပိုင်းလုပ်ပြီးတာနဲ့ return စဖြစ်တယ်။

ဒီအတိုင်းဆက်စစ်သွားရင် တစ်ခုထက်ပိုတဲ့ ကွန်နာတွေအတွက်လည်း မှန်အောင် အလုပ်လုပ်နေမယ်ဆိုတာ သက်သေပြနိုင်ပါတယ်။ စာရေးသူ အတွေ့အကြုံအရ အသေးဆုံးနဲ့ သူ့ထက်ကြီးတာ နှစ်ခုသုံးခု လောက်ထိ မှန်တယ်ဆိုရင် နောက်ဟာတွေအတွက် မှားစရာ အကြောင်းမရှိတော့ဘူး။

ရီကားဆစ်ဖ်နည်းနဲ့ လမ်းတစ်လမ်း ရှင်းလို့ရပါပြီ။ ကားရဲလ်ကမ္ဘာတစ်ခုလုံး ရှင်းဖို့ ရီကားဆစ်ဖ် ဆက်ပြီး စဉ်းစားပါမယ်။

- လမ်းအရေအတွက်ဟာ အရွယ်အစားလို့ ယူဆနိုင်တယ်။ အသေးဆုံးက လမ်းတစ်လမ်းပါ။
- ငါးလမ်းရှိရင် ကျန်တဲ့လေးလမ်းကို ရှင်းနိုင်ပြီးသား မှတ်ယူရမယ်။
- (၁) လမ်းရှင်းပြီး အဆုံးမှာ အပေါ်လမ်း ကူးလိုက်ရင် လေးလမ်းပဲကျန်မယ် (လမ်းအရေအတွက် n ရှိရာကနေ $n-1$ ဖြစ်အောင် ဘာလုပ်မလဲ စဉ်းစားတာ)။
- တစ်ဆင့်ငယ်တဲ့ ကိစ္စ $n-1$ ကို ဖြေရှင်းနိုင်ပြီးသားလို့ မှတ်ယူပြီး လက်ရှိကစ္စ n ကို ဘယ်လို ဖြေရှင်းမလဲ စဉ်းစားရပါမယ်။

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

  တစ်လမ်းရှင်းပြီး နောက်တစ်လမ်းကို ကူးလိုက်ရင် ဆက်ရှင်းဖို့ လမ်း အရေအတွက် $n-1$ ကျန်မယ် (ငါးလမ်းရှိတာကို တစ်လမ်းရှင်းပြီး အပေါ်လမ်းကူးလိုက်ရင် ရှင်းစရာ လမ်း လေးခုပဲ ကျန်မယ်) ။ ကျန်တဲ့ $n-1$ လမ်းကို ရှင်းဖို့ လက်ရှိ clean_world ဖန်ရှင်ကိုပဲ ပြန်ခေါ်တယ်။
- အသေးငယ်ဆုံး ကိစ္စကို ချင်းချက်အနေနဲ့ စဉ်းစားရမယ်။ လမ်းတစ်လမ်းပဲရှိတာက အသေးငယ်ဆုံး။ လမ်းတစ်လမ်းရှင်းပြီး မြောက်ဘက်လှည့်အပြီး ပိတ်နေပြီဆိုရင်တော့ နောက်ထပ် ဆက်ပြီး ရှင်းစရာ လမ်းမရှိတော့ဘူး။ အပေါ်အဆင့်က လမ်းကူးတာနဲ့ ကျန်တဲ့လမ်းတွေကို ရှင်းတဲ့ကိစ္စကို မြောက်ဘက်လှည့်ပြီးတဲ့အခါ ရှေ့မှာရှင်းနေမှ လုပ်ရမှာပါ။ ဒီတော့ အခုလို

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

ဖြစ်ရမယ်။

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

ရှေ့ပိုင်း ကားရဲလ်အခန်း လေးခုမှာ လေ့လာခဲ့ကြတဲ့ အကြောင်းအရာတွေဟာ ပရိုဂရမ်မင်း ဘာသာရပ်ရဲ့ အခြေခံအကျဆုံး ပင်မထောက်တိုင် သဘောတရားတွေလို့ ဆိုရမှာပါ။ ဒီသဘောတရားတွေ မကျေညက်ဘဲ ပရိုဂရမ်ရေးလို့ မရပါဘူး။ ကွန်ဒီရှင်နယ်တွေဖြစ်တဲ့ `if`, `if...else`၊ ပြန်ကျော့ခြင်းအတွက် `for` နဲ့ `while` loop၊ ဖန်ရှင်တွေ၊ top-down, bottom-up ပရိုဂရမ်မင်း၊ ရီကားရှင်းနဲ့ ရီကားဆစ်ဖ် စဉ်းစားခြင်း စတာတွေနဲ့ ပရိုဂရမ်မင်း ပုစ္ဆာတွေ ဖြေရှင်းနိုင်ရင် ပရိုဂရမ်မာလေ့ကား ပထမ တစ်ထစ် တက်လှမ်းအောင်မြင်ပြီ ပြောနိုင်ပါတယ်။ ဒီသဘောတရားတွေကို ဘီဂင်နာတွေ အရိုးရှင်းဆုံးနည်းနဲ့ နားလည်အောင်၊ လေ့ကျင့်လို့ရအောင် ကားရဲလ်က ထောက်ကူပေးတာပါ။ စက်ရုပ်လေး ကားရဲလ်ကို နှုတ်ဆက်ခဲ့ပြီး အခုဆက်လက်လေ့လာကြမှာကတော့ ဒေတာ၊ အိပ်စ်ပရက်ရှင်းနဲ့ ဗေရီရေဘဲလ်တွေ အကြောင်းပါ။

ကွန်ပျူတာတွေဟာ အချက်အလက်(ဒေတာ) အမျိုးမျိုးကို ကိုင်တွယ်ဆောင်ရွက်ပေးနိုင်တယ်။ ကိန်းဂဏန်းတွေအပြင် စာသား၊ ရုပ်သံ စတာတွေကိုပါ လက်ခံ အလုပ်လုပ်ပေးနိုင်တယ်။ ဒီလိုလုပ်ဆောင်နိုင်စွမ်းဟာ ကွန်ပျူတာတွေကို နယ်ပယ်ပေါင်းစုံမှာ တွင်တွင်ကျယ်ကျယ် အသုံးချလာရခြင်းရဲ့ အဓိက အကြောင်းအရင်း ဆိုရင်လည်း မမှားဘူး။

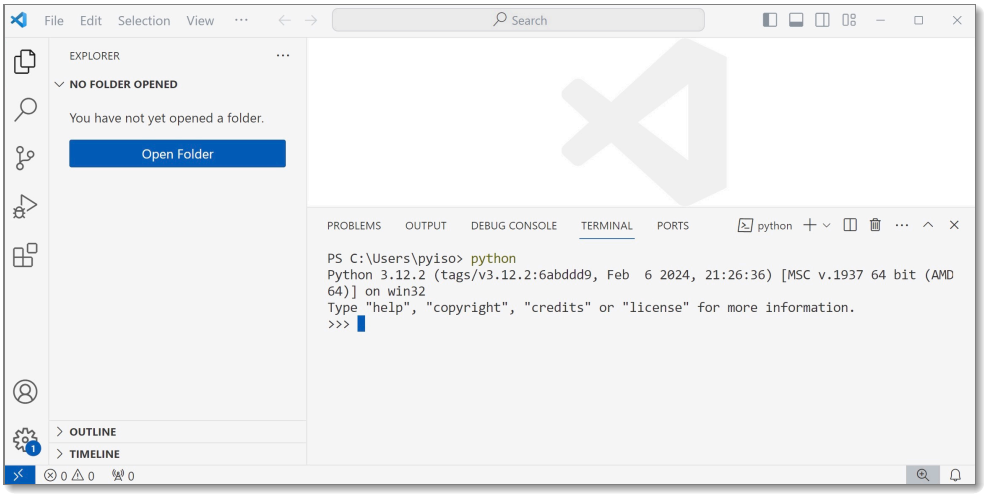
ဒေတာ အမျိုးအစား အများအပြားရှိပေမဲ့ အခြေခံအကျဆုံးက ကိန်းဂဏန်းတွေပါ။ ကွန်ပျူတာတွေကို စတင်တီထွင်ဖို့ ကြိုးစားလာကြတဲ့ အဓိက အကြောင်းအရင်းကလည်း ဂဏန်းသင်္ချာ အတွက်အချက်တွေကို လုပ်ဆောင်ရာမှာ လူတွေကို အကူအညီ ပေးဖို့အတွက်ပဲလို့ ဆိုနိုင်ပါတယ်။ ဒါ့အပြင် စာသား၊ ရုပ်သံ စတဲ့ အခြားဒေတာ အမျိုးအစားတွေကို ကွန်ပျူတာထဲမှာ ဖော်ပြသိမ်းဆည်းထားဖို့အတွက် ကိန်းဂဏန်းတွေကိုပဲ အသုံးပြုထားတယ်ဆိုတာ နောက်ပိုင်းမှာ နားလည်သိမြင် လာပါလိမ့်မယ်။ ဒါကြောင့်လည်း ယနေ့ခေတ် ကွန်ပျူတာတွေကို ဒစ်ဂျစ်တယ် ကွန်ပျူတာတွေလို့ ခေါ်တာဖြစ်တယ်။ ကိန်းဂဏန်းကို အခြေခံပြီး အလုပ်လုပ်တဲ့ ကွန်ပျူတာတွေပေါ့။

၅.၁ ကိန်းဂဏန်းများ

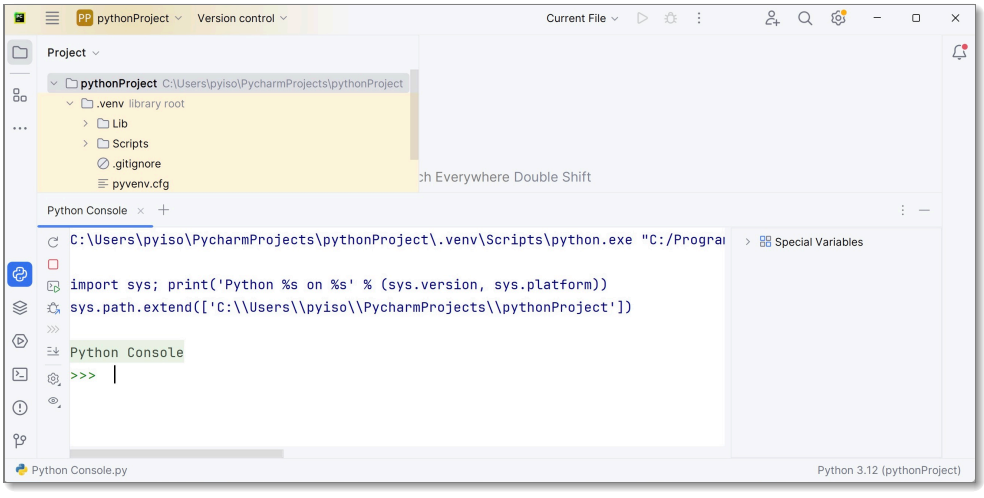
Python အင်တာပရက်တာနဲ့ Python ကုဒ်တွေကို run လို့ရတဲ့နည်း နှစ်ခုရှိပါတယ်။ တစ်နည်းက ကုဒ်တွေကို .py ဖိုင်နဲ့သိမ်းပြီး python ကွန်မန်းနဲ့ run တာပါ။ ဒါက ကားရဲလ် ပရိုဂရမ်တွေမှာ သုံးခဲ့တဲ့နည်း။ နောက်တစ်နည်းကတော့ ကုဒ်တစ်ကြောင်းချင်း ရိုက်ထည့်ပြီး run တဲ့နည်းပါ။ ဒီနည်းလမ်းက interactive mode နဲ့ အင်တာပရက်တာကို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

ကြည့်ရတာ လွယ်တဲ့အတွက်ကြောင့်ပါ။

ဝင်းဒိုး Command Prompt သို့မဟုတ် မက်ခ်အိုအက်စ် Terminal မှာ python ကွန်မန်း run ပြီး interactive mode ကို ဝင်နိုင်ပါတယ်။ VS Code မှာပဲ သုံးချင်လည်း ရတယ်။ View မီနူးမှ Terminal ကိုဖွင့် (Ctrl + ` ရှော့ကတ်ကီးနဲ့ ဖွင့်လို့လည်းရတယ်) ပြီး python ကွန်းမန်း run ရုံပဲ။ PyCharm မှာဆိုရင် Python Console အိုင်ကွန်နှိပ်ပြီး ဝင်ရပါမယ်။ ပုံ (၅.၁)၊ (၅.၂) တွင်ကြည့်ပါ။



ပုံ ၅.၁ VS Code Python Console



ပုံ ၅.၂ PyCharm Python Console

2 + 5 ထည့်ပြီး Enter ကီးနှိပ်ပါ။ အင်တာပရက်တာက ရလဒ် 7 နဲ့ တုံ့ပြန် လုပ်ဆောင်ပေးပါလိမ့်မယ်။ ဒီလို အင်တာပရက်တာက လုပ်ဆောင်ပေးတာကို *evaluate* လုပ်တယ်လို့ ပြောတယ်။

```
>>> 2 + 5
7
```

အောက်ပါအတိုင်း တစ်ခုပြီးတစ်ခု ဆက်လက်စမ်းသပ်ကြည့်ပါ။

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

3×3 ကို `3 * 3` လို့ ရိုက်ထည့်ပေးရပြီး $5 \div 2$ အတွက် `5 / 2` လို့ ရေးရတာကို သတိပြုမိမှာပါ။ Programming language အများစုမှာ `*` (asterisk) အမြှောက်သင်္ကေတအဖြစ် အသုံးပြုပြီး `/` (forward slash) ကို အစားသင်္ကေတအနေနဲ့ အသုံးပြုလေ့ရှိတယ်။

Values and Types

တန်ဖိုးတိုင်းဟာ တိုက်ပ် (*type*) တစ်မျိုးမျိုးမှာ ပါဝင်ပါတယ်။ `-3`, `0`, `2` စတဲ့ တန်ဖိုးတွေဟာ `int` (integer ရဲ့ အတိုကောက်) တိုက်ပ်ဖြစ်ပြီး `-3.0`, `0.1`, `3.3333` စတာတွေက `float` တိုက်ပ် ဖြစ်ပါတယ်။ ဒဿမကိန်းတွေကို ကွန်ပျူတာနဲ့ ဖော်ပြဖို့ *floating point* လို့ခေါ်တဲ့ နည်းစနစ်ကို အသုံးပြုတယ်။ ဒဿမကိန်း အတွက်အချက်တွေကိုလည်း ဒီနည်းစနစ်ကို အခြေခံပြီး ကွန်ပျူတာက လုပ်ဆောင်တာပါ။ ဒါကြောင့် floating point ဟာ ဒဿမကိန်းတွေကို ဖော်ပြဖို့နဲ့ ဒဿမကိန်း အတွက်အချက်တွေ လုပ်ဆောင်ဖို့ တီထွင်ထားတဲ့ နည်းစနစ်တစ်ခုလို့ ဆိုနိုင်ပါတယ်။ ဒီစနစ်ကို အခြေခံထားတဲ့ ဒဿမကိန်းတွေကို programming language တွေမှာ `float` တိုက်ပ်လို့ ခေါ်တာပါ။

`float` တိုက်ပ်ဟာ လိုသလောက် တိကျလို့မရတဲ့ သဘောရှိတယ်။ အောက်ပါအတိုင်း စမ်းကြည့်ရင် `0.3` နဲ့ `1.0` ရသင့်တာ ဖြစ်ပေမဲ့ အတိအကျ အဖြေမထွက်ပါဘူး။

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

ကွာခြားချက်က မဆိုစလောက် သေးငယ်တယ် ဆိုပေမဲ့ ဒီအချက်ကို ပရိုဂရမ်မာ အနေနဲ့ ဂရုပြုမိဖို့ လိုပါတယ်။ Floating point စနစ်ဟာ လုံးဝကြီးတိကျဖို့ မလိုအပ်တဲ့ (တနည်းအားဖြင့် မဆိုစလောက် သေးငယ်တဲ့ ကွာဟချက်ကို လက်ခံနိုင်တဲ့) ကိန်းဂဏန်းအတွက်အချက် ကိစ္စတွေအတွက် ရည်ရွယ်တာပါ။ သိပ္ပံနဲ့ နည်းပညာဆိုင်ရာ တိုင်းတာ တွက်ချက်မှုတွေအတွက် အသုံးပြုလေ့ရှိတယ်။ ဒဿမကိန်းတွေ လုံးဝအတိအကျ ဖြစ်ဖို့ လိုအပ်တဲ့ ကိစ္စမျိုးတွေ (ဥပမာအားဖြင့် ငွေကြေးကိစ္စ အတွက်အချက်) မှာ အသုံးမပြုသင့်ပါဘူး။ ဆယ်ပြားကို `0.1` နဲ့ဖော်ပြရင် ဆယ်ပြားစေ့ ဆယ်စေ့ဟာ တစ်ကျပ် ဖြစ်ကို ဖြစ်သင့်ပြီး `0.9999999999999999` မဖြစ်သင့်ဘူး။ ဒီလိုကိစ္စမျိုးတွေအတွက် Python မှာ `Decimal` ကို အသုံးပြုနိုင်ပါတယ်။ လောလောဆယ် ကိန်းဂဏန်းတွေနဲ့ ပါတ်သက်ပြီး စကတည်းက သိထားသင့်တာတချို့ကို ကြိုပြောထားတာပါ။ `Decimal` တိုက်ပ် အကြောင်း မကြာခင် 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။ သီအိုရီအရ ကြိုက်သလောက် ကြီးလို့/ငယ်လို့ ရပါတယ်။ လက်တွေ့

မှာတော့ အသုံးပြုတဲ့ ကွန်ပျူတာစနစ်ပေါ် မူတည်ပြီး ကန့်သတ်ချက်ရှိမှာပါ။ ဒီစာရေးနေတဲ့ ကွန်ပျူတာမှာ အောက်ပါကိန်းပြည့်တွေကို အသာလေး တွက်ချက်ပေးနိုင်ပါတယ်။ ဒီ့ထက်အများကြီး ကြီးတဲ့ဟာတွေကိုလည်း ကိုင်တွယ်တွက်ချက်နိုင်အုံးမှာပါ။ တော်ရုံကိစ္စတွေအတွက် ကွန်ပျူတာစနစ်တစ်ခုရဲ့ လက်တွေ့ကန့်သတ်ချက်ကို ကျော်လွန်သွားဖို့ မလွယ်ပါဘူး။

```
>>> 9000000000000000000000000000 + 5000000000
9000000000000000005000000000
>>> -9000000000000000000000000000 + 1
-8999999999999999999999999999
```

`float` ကတော့ `int` နဲ့ မတူဘဲ အကြီးဆုံး/အငယ်ဆုံး ကန့်သတ်ချက်ရှိပါတယ်။ ဒီကန့်သတ်ချက်ကလည်း ကွန်ပျူတာစနစ်ပေါ် မူတည်ပါတယ်။ တစ်ကိုယ်ရေသုံး ကွန်ပျူတာ အများစုအတွက် 1×10^{400} (၁ နောက် သုည အလုံး ၄၀၀) သို့မဟုတ် -1×10^{400} (အနှုတ် ၁ နောက် သုည အလုံး ၄၀၀) ဟာ ကန့်သတ်ချက်ကျော်လွန်ပါတယ်။

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

ဒဿမကိန်းကို e အမှတ်အသားနဲ့ရေးထားတာပါ။ အကွာရာ `e` နောက် လိုက်တဲ့ ကိန်းပြည့်ဂဏန်းကို 10 ထပ်ကိန်းလို့ မှတ်ယူရပါတယ်။ `e3` က 1×10^3, `e-3` က 1×10^{-3} ပါ။ အကွာရာ E အကြီးနဲ့လည်း ရေးနိုင်တယ်။ ထပ်ကိန်း ကြီးလွန်းတဲ့အတွက် ကန့်သတ်ချက် ကျော်လွန်ရင် `inf` (အနှုတ်ကိန်းဆိုရင် `-inf`) ရပါမယ်။ Infinity ကို ဆိုလိုတာပါ။ (၁) မပြည့်တဲ့ ပမာဏ သေးငယ်လွန်းတဲ့ ဂဏန်းတွေကိုလည်း အနီးစပ်ဆုံး သုညအနေနဲ့ ယူပါတယ်။ အနီးစပ်ဆုံးယူတဲ့အခါ အပေါင်း/အနှုတ်ကိုတော့ ခွဲခြားပေးပါတယ်။ e အမှတ်အသားနဲ့ရေးထားတဲ့ နောက်ထပ် ဥပမာလေး နှစ်ခုပါ

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

ဂဏန်းအလုံးအရေအတွက် များရင် $100,500$ (တစ်သိန်းငါးရာ)၊ $1,500,000$ (တစ်သန်းငါးသိန်း) စသဖြင့် သုံးလုံးတစ်ဖြတ် ကော်မာခံရေးလေ့ရှိတယ်။ Python မှာတော့ ကော်မာအစား _(underscore) နဲ့ သုံးလုံးတစ်ဖြတ် ခြားရေးနိုင်ပါတယ်။

```
>>> 1_500_000 + 100_500
1600500
>>> 200_000.33 + 3_800_000.22
4000000.5500000003
```

တိုက်ပ်မတူသည့် ကိန်းဂဏန်း အိပ်စ်ပရက်ရှင်များ

အိပ်စ်ပရက်ရှင်တွေမှာ ပါဝင်တဲ့ တိုက်ပ် တစ်မျိုးတည်း ဖြစ်ရမယ် မရှိပါ။ တိုက်ပ်မတူတဲ့ ကိန်းဂဏန်းတွေ အိပ်စ်ပရက်ရှင်တစ်ခုမှာ ရောပြီး ပါဝင်နိုင်ပါတယ်။

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

int နဲ့ float ရောနေရင် အိပ်စ်ပရက်ရှင် ရလဒ်သည် float တိုက်ပ် 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 ဒီဗီးရှင်း၊ မော်ဒျူလို နှင့် ထပ်ကိန်းတင်ခြင်း

အကယ်၍ ဒသမကိန်းမထွက်ဘဲ ကိန်းပြည့် လိုချင်ရင် // ကို သုံးရပါမယ်။ ဒီအခါ အကြွင်းကိုဖယ်ပြီး စားလဒ်ကိုပဲ ကိန်းပြည့်အနေနဲ့ ရမှာပါ။

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

သင်္ချာမှာ ဒီလိုမျိုး အစားကို အင်တီဂျာ ဒီဗီးရှင်း (integer division) လို့ခေါ်ပါတယ်။ အကြွင်းရှာမယ် ဆိုရင် % အော်ပရိတ်တာ ရှိပါတယ်။ % ကို မော်ဒျူလို (modulo) အော်ပရိတ်တာလို့ ခေါ်တယ်။ remainder အော်ပရိတ်တာလို့လည်း ခေါ်တယ်။

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

အင်တီဂျာ ဒီဗီးရှင်းနဲ့ မော်ဒျူလိုကို အနှုတ်ကိန်းတွေနဲ့ သုံးမယ်ဆိုရင် သတိပြုပါ။ စားလဒ် အနှုတ်ကိန်း ဖြစ်ရင် // က ပိုငယ်တဲ့ အနှုတ်ကိန်းကို 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။ အဖြေအတိအကျက -1.2 ကို အနီးစပ်ဆုံး သူ့ထက်ပိုငယ်တဲ့ -2 ကို အနီးစပ်ဆုံး ယူတယ်။ -3.1, -3.5, -3.8 တို့ကိုလည်း အနီးစပ်ဆုံး -4 ယူတာပါ။

မော်ဒျူလို အော်ပရိတ်တာ % သုံးတဲ့အခါ ရလဒ်ဟာ စားကိန်းရဲ့ sign အပေါ် မူတည်တယ်။ (အပေါင်း အနှုတ်ကို ဆိုလိုတာပါ)။

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

မော်ဒျူလိုနဲ့ အင်တီဂျာ ဒီဗီးရှင်း 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 ဒဿမကိန်းတွေနဲ့လည်း 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 အခုလို တွက်ပါတယ်။

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

သင်္ချာဖန်ရှင်များ

ကိန်းဂဏန်းတွေအကြောင်း လေ့လာလက်စနဲ့ math လိုက်ဘရီ သင်္ချာဖန်ရှင်တချို့ကိုလည်း တစ်ခါတည်း ဆက်ကြည့်လိုက်ရအောင်။ အဓိကက သင်္ချာဖန်ရှင်ဆိုတာထက် ဖန်ရှင် အခြေခံအသုံးပြုပုံကို စပြီးလေ့လာမှာပါ။ math လိုက်ဘရီက Python မှာ တစ်ခါတည်း ထည့်ထားပေးပြီးသား (built-in) လိုက်ဘရီပါ။ အင်စတောလ်လုပ်စရာ မလိုဘဲ အင်ပို့လုပ်ပြီး သုံးလို့ရတယ်။

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

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

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

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

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

အကြီးဆုံးဘုံဆခွဲကိန်းကို gcd ဖန်ရှင်နဲ့ ရှာတာပါ။ အင်တီဂျာ input တစ်ခုနဲ့အထက် လက်ခံတဲ့ ဖန်ရှင်ဖြစ်တယ်။ input ဂဏန်းအားလုံးကို စားလို့ပြတ်တဲ့ အကြီးဆုံးကိန်းကို ရှာပေးတယ်။ ကိန်းပြည့်မဟုတ်တာ ထည့်ရင် အယ်ရာဖြစ်ပါတယ်။

```
>>> 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
```

လော့ဂရစ်သမ်၊ ထရီဂိုနိုမေထရီ ဖန်ရှင်တွေလည်းပါတယ်။ $\log_{10}(x)$, $\sin(x)$, $\cos(x)$ တို့ကို ဥပမာ ပြထားတာ 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
```

၅.၂ 'တိုက်ပ်' ဆိုတာ ဘာလဲ

Programming language အားလုံးမှာ တိုက်ပ် သို့မဟုတ် ဒေတာတိုက်ပ် သဘောတရား ပါရှိပါတယ်။ int နဲ့ float တိုက်ပ်မှာ ပါဝင်တဲ့ ကိန်းပြည့်တွေနဲ့ ဒဿမကိန်းတွေကို မိတ်ဆက်ပြီးတဲ့အခါ 'တိုက်ပ်' ဆိုတာဘာလဲ တိတိကျကျ ရှင်းပြလို့ရပါပြီ။ တိုက်ပ်တစ်ခုဟာ

- တန်ဖိုးတွေပါဝင်တဲ့ အစု (set) တစ်ခု နဲ့
- ၎င်းတန်ဖိုးများအပေါ်တွင် အသုံးချနိုင်တဲ့ အော်ပရေးရှင်းတွေ ပါဝင်တဲ့ အစုတစ်ခု

ဖြစ်ပါတယ်။ ဥပမာ int တိုက်ပ်ကို ကြည့်ရင် ကိန်းပြည့်တွေ ပါဝင်တဲ့ အစုနဲ့ ကိန်းပြည့်တွေအပေါ်မှာ လုပ်ဆောင်လို့ရတဲ့ အော်ပရေးရှင်းတွေ ပါဝင်တဲ့ အစု

$$\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$$
$$\{+, -, *, /, //, \%, **, \ldots\}$$

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

အပေါ်က 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 အသုံးချပုံပါ။ နှစ်ထပ်ကိန်းရင်းနဲ့ ပကတိတန်ဖိုး ရှာပေးပါတယ်။ လိုအပ်ရင် ကိုယ်ပိုင်ဖန်ရှင်တွေ သတ်မှတ်ပြီး တိုက်ပ်တစ်ခုရဲ့ အော်ပရေးရှင်းတွေကို ဖြည့်စွက်တိုးချဲ့နိုင်ပါတယ်။

> အော်ပရေးရှင်းနဲ့ အော်ပရိတ်တာ ရောထွေးစရာ ရှိပါတယ်။ $+, -, *, /, //, \%, **$ စတဲ့ သင်္ကေတတွေကို အော်ပရိတ်တာလို့ ခေါ်ပါတယ်။ အော်ပရေးရှင်း လုပ်ဆောင်ဖို့အတွက် အသုံးပြုတဲ့ သင်္ကေတတွေကို အော်ပရိတ်တာလို့ ခေါ်တာပါ။ ဥပမာ "$*$ သင်္ကေတဟာ အမြှောက်အော်ပရေးရှင်း လုပ်ဆောင်ဖို့ သတ်မှတ်ထားတဲ့ အော်ပရိတ်တာ" လို့ ပြောတယ်။ အမြှောက် 'အော်ပရေးရှင်း' ကျတော့ မြှောက်တဲ့အလုပ် ဆောက်ရွက်တာကို ဆိုလိုတာ။

၅.၃ ဗေရီရေဘဲလ်များ

ဗေရီရေဘဲလ်ဆိုတာ တန်ဖိုးတစ်ခုကို ကိုယ်စားပြုတဲ့ နံမည်ပါပဲ။ နံမည်နဲ့ ၎င်းကိုယ်စားပြုတဲ့ တန်ဖိုး တွဲဖက်ပေးဖို့ အဆိုင်းမန့် (*assignment*) စတိတ်မန့်ကို သုံးရပါတယ်။

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

age နဲ့ weight ဟာ ဗေရီရေဘဲလ်တွေ ဖြစ်ပါတယ်။ ညီမျှခြင်းသင်္ကေတ (=) ကတော့ အဆိုင်းမန့် အော်ပရိတ်တာပါ။ ဗေရီရေဘဲလ်နဲ့ တန်ဖိုး တွဲဖက်ပေးတဲ့ အော်ပရိတ်တာ ဖြစ်တယ်။ ဗေရီရေဘဲလ်နံမည်ကို variable *identifier* လို့လည်း ခေါ်ပါတယ်။ Identifier က နည်းပညာ အခေါ်အဝေါ်ပေါ့။ Variable name က သာမန်လူ နားလည်တဲ့ နည်းနဲ့ ပြောတာပါ။ ဗေရီရေဘဲလ် တစ်ခုချင်း ထည့်ကြည့်ရင် ၎င်းကိုယ်စားပြုတဲ့ တန်ဖိုးကို ပြန်ထုတ်ပေးတာ တွေ့ရမှာပါ။

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

အိပ်စ်ပရက်ရှင်တွေက ဗေရီရေဘဲလ်တွေနဲ့ ဖြစ်နိုင်ပါတယ်။ အိပ်စ်ပရက်ရှင် တွက်ချက်ရင် ဗေရီရေဘဲလ် တန်ဖိုးနဲ့ အစားထိုး တွက်ချက်တယ်လို့ ယူဆရမှာပါ။ ဥပမာ

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

ဗေရီရေဘဲလ်တစ်ခုကို အိပ်စ်ပရက်ရှင်ရလဒ်နဲ့ အဆိုင်းမန့် လုပ်လို့ရပါတယ်။ rect_area ကို အောက်တွင် ကြည့်ပါ။ အလျား အနံ မြှောက်လဒ်ကို အဆိုင်းမန့် လုပ်ထားတာ တွေ့ရပါမယ်။

```
>>> rect_width = 22.5
>>> rect_length = 10
```

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

အဆိုင်းမန့် စတိတ်မန့်

ဗေရီရေဘဲလ်တစ်ခုဟာ အချိန်တစ်ချိန်မှာ တန်ဖိုးတစ်ခုကိုပဲ ကိုယ်စားပြုနိုင်တယ်။ ဒါပေမဲ့ အချိန်ကာလပေါ် မူတည်ပြီး တန်ဖိုးပြောင်းနိုင်တယ်။ (တစ်ချိန်တည်းမှာ တန်ဖိုးနှစ်ခု မဖြစ်နိုင်ဘူး)။ ဥပမာ x တန်ဖိုးဟာ ပထမ 10 ပါ။ ဒုတိယ အဆိုင်းမန့်လုပ်ပြီးတဲ့ အချိန်မှာ အဲ့ဒီ x ကပဲ 1000 ဖြစ်နေမှာပါ။

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

၅.၄ စာသားများ

စာသား (text) ဟာ အသုံးအများဆုံး ဆက်သွယ်ဆောင်ရွက်ရေး ကြားခံနယ်တစ်ခုပါ။ ဝက်ဘ်ဆိုက် စာမျက်နှာ၊ အီးမေးလ်၊ အီးဘွတ်ခ်နဲ့ အီလက်ထရွန်းနစ် စာရွက်စာတမ်း (e-documents) စတာတွေမှာ ရုပ်သံတွေ အသုံးပြုလာကြပေမဲ့ စာသား အဓိကဖြစ်နေဆဲပါပဲ။ ဆိုရှယ်မီဒီယာ၊ ဂိမ်းနဲ့ အခြားအပ်ပ်တွေ ဟာလည်း စာသားနဲ့ မကင်းနိုင်ကြပါဘူး။ ဒါကြောင့် ပရိုဂရမ်းမင်းအတွက် စာသားဟာ ဘယ်လောက်ထိ အရေးပါကြောင်း အများကြီးပြောစရာ လိုမယ်မထင်ပါဘူး။

ပရိုဂရမ်းမင်းမှာ စာသားကို *string* လို့ခေါ်ပြီး ကာရက်တာ (*character*) တွေနဲ့ စီတန်းဖွဲ့စည်းထားတယ်။ ကာရက်တာဆိုတာ အခြေခံ သတင်းအချက်အလက် ယူနစ်တစ်ခုပါပဲ။ အက္ခရာ၊ ဂဏန်း (digit)၊ သင်္ကေတ သို့မဟုတ် ကွန်ထရိုးလ်ကုဒ် တစ်ခုခု ဖြစ်နိုင်ပါတယ်။ ဥပမာ A, B, C, $, @, #, 1, 3, _ စသည်ဖြင့်။ 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 တိုက်ပ်ရဲ့ အခြေခံကျတဲ့ အော်ပရေးရှင်းတစ်ခုက စာသားတစ်ခုနဲ့ တစ်ခု ဆက်တာပါ။ + အော်ပရိတ်တာနဲ့ စာသားတွေကို ဆက်နိုင်ပါတယ်။

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

စာသားအချင်းချင်းပဲ ဆက်လို့ရပါတယ်။ စာသားနဲ့ ကိန်းဂဏန်း ဆက်လို့မရပါဘူး။ အောက်ပါအတိုင်း စမ်းကြည့်တဲ့အခါ str နဲ့ float ဆက်လို့မရဘူးလို့ အယ်ရာမက်ဆေ့ချ် 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။ မူရင်းကိန်းဂဏန်းကို စာသားဖြစ်အောင် ပြောင်းလိုက်တာ မဟုတ်ပါဘူး။ ကိန်းဂဏန်း တန်ဖိုးကနေ ၎င်းကိုဖော်ပြတဲ့ စာသားကို ဖန်ရှင်ကပြန်ထုတ်ပေးတာပါ။

```
>>> str(pi)
'3.141592653589793'
```

ထွက်လာတဲ့ တန်ဖိုးဟာ စာသားဖြစ်တဲ့အတွက် single quote ပါနေတာ သတိပြုပါ။ pi တန်ဖိုးနဲ့ စာသားအခုလို 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
7.199999999999999
```

ဂဏန်းပြောင်းလို့မရတဲ့ စာသားဖြစ်နေရင် အယ်ရာဖြစ်မှာပါ။

```
>>> 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 ဒဿမ ပါနေတာကြောင့် int ပြောင်းလို့ မရတဲ့အတွက် အယ်ရာတက်တာပါ။

စာသားကို * အော်ပရိတ်တာနဲ့ ပွားယူလို့ရတယ်။

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

'hello' သုံးခါ ဆက်လိုက်တာပါ။ string နဲ့ အကြိမ်အရေအတွက် ကိန်းပြည့် ဖြစ်ရပါမယ်။ သုညသို့မဟုတ် အနှုတ်ကိန်း ဖြစ်နေရင် empty string ရမှာပါ။

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

စာသားနဲ့ ဂဏန်း ဖလှယ်လို့ရပါတယ်

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

string ဗေရီရေဘဲလ်များ

ဗေရီရေဘဲလ်တွေကို string တန်ဖိုးတွေအတွက်လည်း အသုံးပြုနိုင်တယ်။ အောက်ပါ အိပ်စ်ပရက်ရှင်တွေကို နားလည်နိုင်မလား ကြိုးစားကြည့်ပါ။ ဗေရီရေဘဲလ် တစ်ခုချင်းကို သူ့ရဲ့တန်ဖိုးနဲ့ အစားထိုးပြီး +, * အော်ပရိတ်တာတွေ အလုပ်လုပ်ပုံနဲ့ ဆက်စပ်စဉ်းစားရင် ဘာကြောင့် အခုလို အဖြေထွက်လဲ ခန့်မှန်းနိုင်မှာပါ။ သိပ်ခက်ခက်ခဲခဲ 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 တစ်ခု ရေးတဲ့အခါ ပုံမှန်အားဖြင့် လိုချင်တဲ့ စာသားအတိုင်း ကီးဘုဒ်ကနေ ကာရက်တာ တစ်လုံးချင်း ရိုက်ရုံပါပဲ။ စပယ်ရှယ် ကာရက်တာ တချို့ကိုတော့ ကီးဘုဒ်ကနေ တိုက်ရိုက် ရိုက်ထည့်လို့ မရဘဲ သီ

ခြားနည်းလမ်းတစ်ခုနဲ့ ရေးပေးရပါတယ်။ ဥပမာ စာသားထဲမှာ tab ကာရက်တာအတွက် `\t` နဲ့ newline အတွက် `\n` ရေးရမှာပါ။ ကီးဘုဒ်ကနေ tab ကီး၊ enter/return ကီး နှိပ်ပြီး တိုက်ရိုက်ထည့်လို့မရပါဘူး။ သီးခြားအဓိပ္ပါယ် တစ်ခုအတွက် `\` နဲ့စတဲ့ ကာရက်တာအတွဲလိုက်ကို *escape sequence* လို့ခေါ်ပြီး `\` 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 တွေကို **bold** ဖောင့်နဲ့ ပြထားပါတယ်။ Python ကွန်ဆိုးလ်မှာ `\t` နဲ့ `\n` ကို အရှိအတိုင်း ပြနေပါတယ်။ ဒါပေမဲ့ အခုလို 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 `\"` လို့ရေးရပါမယ်။ Single quote တစ်စုံနဲ့ စာသားထဲမှာ `'` ပါနေရင်လည်း `\'` လို့ရေးရပါမယ်။

```
>>> '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 သို့မဟုတ် 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။ ကျန်တဲ့တစ်မျိုးကတော့ ပါရင်လည်းရ၊ မပါလည်း ပြဿနာမရှိဘူး။ အောက်ပါတို့ကို ဂရုစိုက် 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	**အဓိပ္ပါယ်**
`\'`	single quote
`\"`	double quote
`\\`	backslash
`\t`	tab
`\n`	newline
`\r`	carriage return

တေဘဲလ် ၅.၁ Python Escape Sequences

၅.၅ အိပ်စ်ပရက်ရှင်များ

တိုက်ပ် ဆိုတာဘာလဲ အကြမ်းဖျဉ်း ရှင်းပြခဲ့ ပြီးပါပြီ။ ကိန်းဂဏန်းနဲ့ စာသား တိုက်ပ် တချို့ကိုလည်း လေ့လာခဲ့ပြီးပြီ။ တကယ်တော့ အိပ်စ်ပရက်ရှင် (*expression*) ဆိုတာလည်း အသစ်အဆန်း မဟုတ်ပါဘူး။ တန်ဖိုးတစ်ခုဟာ အရိုးရှင်းဆုံး အိပ်စ်ပရက်ရှင်လို့ ဆိုနိုင်ပါတယ်။ `"Hello"`, `2.3` စတဲ့ တန်ဖိုးတွေဟာ အိပ်စ်ပရက်ရှင်တွေပါပဲ။ ဗေရီရေဘဲလ်ဟာလည်း တန်ဖိုးကို ကိုယ်စားပြုတဲ့အတွက် အိပ်စ်ပရက်ရှင်လို့ ယူဆရမှာပါ။

ရိုးရှင်းတဲ့ အိပ်စ်ပရက်ရှင်တွေကနေတစ်ဆင့် ပေါင်းစပ် အိပ်စ်ပရက်ရှင် (compound expression) တွေ ဖွဲ့စည်းတည်ဆောက် ယူနိုင်ပါတယ်။

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

အိပ်စ်ပရက်ရှင်ကို ရှုထောင့်အမျိုးမျိုးကနေ အဓိပ္ပါယ်ဖွင့်ဆိုကြတာ တွေ့ရပါတယ်။ "အိပ်စ်ပရက်ရှင်ဆိုတာ တန်ဖိုးတစ်ခု ပြန်ပေးတဲ့ အော်ပရေးရှင်း အတွဲအဆက်ဖြစ်တယ်" လို့ဆိုရင် အတိုင်းအတာတစ်ခုအထိ တိတိကျကျရှိပြီး နားလည်ရလည်း လွယ်ပါတယ်။ တချို့စာအုပ်တွေမှာတော့ အိပ်စ်ပရက်ရှင်ကို တန်ဖိုးပြန်ပေးတဲ့ စတိတ်မန့်လို့ သတ်မှတ်ပါတယ်။

ပထမအဓိပ္ပါယ်အရ အိပ်စ်ပရက်ရှင်နဲ့ စတိတ်မန့် မတူဘူးလို့ ယူဆပါတယ်။ ဒီရှုထောင့်က ကြည့်ရင် စတိတ်မန့်ဟာလည်း အော်ပရေးရှင်း အတွဲအဆက်ဖြစ်ပေမဲ့ တန်ဖိုးပြန်မပေးဘူး။ အိပ်စ်ပရက်ရှင်ကတော့ တန်ဖိုးပြန်ပေးရမှာပါ။ `3 * 2` ကို လုပ်ဆောင်တဲ့အခါ 6 ရပါတယ်။ ဒါကြောင့် `3 * 2` ဟာ အိပ်စ်ပရက်ရှင်ဖြစ်တယ်။ `result = 3 * 2` က စတိတ်မန့်ဖြစ်တယ်။ အဆိုင်းမန့်ဟာ ဗေရီရေဘဲလ်ကို တန်ဖိုးတစ်ခုနဲ့ တွဲဖက်ပေးတာ။ တန်ဖိုးပြန်မပေးဘူး။

ဒုတိယအဓိပ္ပါယ်အရ အိပ်စ်ပရက်ရှင်သည်လည်း စတိတ်မန့်ပဲ။ စတိတ်မန့်တွေကို တန်ဖိုးပြန်ပေးတဲ့ စတိတ်မန့်နဲ့ ပြန်မပေးတဲ့ စတိတ်မန့် အုပ်စုနှစ်စု ခွဲခြားတယ်။ တန်ဖိုးပြန်ပေးတဲ့ စတိတ်မန့်တွေကို အိပ်စ်ပရက်ရှင် သို့မဟုတ် အိပ်စ်ပရက်ရှင်စတိတ်မန့်လို့ ဒုတိယအဓိပ္ပါယ် သတ်မှတ်ချက်အရ ခေါ်တာပါ။

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

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

`print(3)` ကတော့ အိပ်စ်ပရက်ရှင် မဟုတ်ပါဘူး။ အခုလို စမ်းကြည့်ရင် 3 ထုတ်ပေးတဲ့အတွက် အိပ်စ်ပရက်ရှင်လို့ ထင်စရာ အကြောင်းရှိပါတယ်။

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

ဒါပေမဲ့ ဒါဟာ `print` ဖန်ရှင်က output ထုတ်ပေးတာပါ။ တန်ဖိုးပြန်ပေးတာ မဟုတ်ပါဘူး။ တန်ဖိုးပြန်ရတယ်ဆိုရင် အခြားအိပ်စ်ပရက်ရှင်တစ်ခုမှာ 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 တိုက်ပ်နှင့် ဘူလီယန် အိပ်စ်ပရက်ရှင်

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

```
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 သင်္ကေတတွေဟာ comparison operator တွေပါ။ Relational operator တွေလို့လည်း ခေါ်ကြတယ်။ အလယ်တန်းသင်္ချာမှာ သင်ခဲ့ရတဲ့ $<$, $>$, $\leq$, $\geq$, $=$, $\neq$ စတာတွေနဲ့ အဓိပ္ပါယ်တူပါတယ်။ ညီ/မညီ စစ်ချင်ရင် ညီမျှခြင်းသင်္ကေတနှစ်ခု `==` နဲ့ စစ်ရပါမယ်။ Comparison operator တွေကို ဒေတာတိုက်ပ် အမျိုးမျိုးနဲ့ တွဲဖက် အသုံးပြုနိုင်တာကို တွေ့ရမှာပါ။ တန်ဖိုးတစ်ခုနဲ့ တစ်ခု နှိုင်းယှဉ်နိုင်ခြင်းဟာ 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 သုံးခုကတော့ `and`, `or`, `not` တို့ပဲ ဖြစ်ပါတယ်။ ၎င်းတို့ကို 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
```

ကိန်းဂဏန်းတစ်ခုက တစ်နဲ့ တစ်ဆယ်ကြား ရှိ/မရှိ အခုလိုစစ်နိုင်ပါတယ်

```
>>> 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 စစ်လို့လည်း ရတယ်)။ ယဉ်မောင်းလိုင်စင် ဖြေဖို့ အသက် ဆယ့်ရှစ်နှစ်နဲ့ အထက် ဖြစ်ရမယ်၊ မှတ်ပုံတင်လည်း ရှိရမယ်ဆိုပါစို့။ ဘူလီယန် အိပ်စ်ပရက်ရှင်နဲ့ 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 မူတည်ပြီး တွက်ချက်လုပ်ဆောင်တာတွေ တွေ့ရမှာပါ)။

ဘူလီယန်တန်ဖိုး/အိပ်စ်ပရက်ရှင် နှစ်ခုအနက် အနည်းဆုံး တစ်ခု မှန်/မမှန် `or` နဲ့ စစ်နိုင်တယ်။ နှစ်ခုလုံး 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 စစ်နိုင်တယ်

```
>>> 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) တစ်ဆယ်ထက်ကြီးရင် တစ်နဲ့တစ်ဆယ်ကြားမှာ မဟုတ်လို့ပါ။ (Python မှာ တန်ဖိုးနှစ်ခုကြားမှာရှိလား သို့မဟုတ် တန်ဖိုးနှစ်ခု ပြင်ပမှာရှိလား စစ်လို့ရတဲ့နည်း တစ်ခုမကရှိပါတယ်။ အခုနည်းလမ်းက and နဲ့ or ကို အသုံးချတဲ့ ဥပမာတချို့ကို ပြခြင်းသာဖြစ်တယ်)။

ကုမ္ပဏီ တစ်ခုက အလုပ်ခေါ်တဲ့အခါ ဘွဲ့ရ (သို့) ဒီပလိုမာနဲ့ လုပ်သက် (၂) နှစ်အထက် အနည်းဆုံး ရှိသူ ဖြစ်ရမယ် ဆိုပါစို့။ 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 အဲ့ဒီတန်ဖိုးနှစ်ခု အပါအဝင်လား (inclusive)၊ ဒါမှမဟုတ် မပါဝင်ဘူးလား (exclusive) ဂရုစိုက်ဖို့ လိုပါတယ်။ တစ်နဲ့ တစ်ဆယ်ကြား (ၢင်းတို့ 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
```

တစ်နဲ့ တစ်ဆယ် ပြင်ပ (ၢင်းတို့မပါ) မှာ ရှိ/မရှိကို အခုလိုလည်း စစ်လို့ရတယ်

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

'ငါးရာအောက်' လို့ ပြောတာဟာ 'ငါးရာနှင့်အထက် မဟုတ်' လို့ ပြောတာနဲ့ အဓိပ္ပါယ်တူတူပါပဲ။

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

not အော်ပရိတ်တာ သုံးခြင်းအားဖြင့် အကြောင်းအရာတစ်ခုကိုပဲ အဓိပ္ပါယ်အားဖြင့် ညီမျှတဲ့ အိပ်စ်ပရက်ရှင် အမျိုးမျိုးနဲ့ ဖော်ပြလို့ရလာတာကို တွေ့နိုင်တယ်။

အခန်း ၆

အော့ဘ်ဂျက်များ

"အော့ဘ်ဂျက် (*object*) ဆိုတာဘာလဲ" ရှုထောင့် အမျိုးမျိုးကနေ ရှင်းပြနိုင်ပါတယ်။ အပြည့်စုံဆုံး၊ အမှန်ကန်ဆုံး ဥပမာ သို့မဟုတ် အဓိပ္ပါယ်ဖွင့်ဆိုချက် ဆိုတာ မရှိပါဘူး။ သူ့နည်း သူ့ဟန်နဲ့ မှန်ကန်ကြတာပါပဲ။ ဒီအခန်းမှာတော့ အော့ဘ်ဂျက်ဆိုတာ ဘာလဲ၊ ဘယ်လိုမျိုးလဲ ခံစားလို့ရအောင်နဲ့ အခြေခံ အသုံးချတတ်ရုံ လောက်ပဲ အဓိကထား လေ့လာကြမှာပါ။ လက်ရှိအခြေအနေနဲ့ သင့်တော်မဲ့ အဓိပ္ပါယ်ဖွင့်ဆိုချက် တချို့ကိုလည်း ဖော်ပြပေးသွားမှာပါ။ သိထားသင့်တဲ့ ဘာသာရပ်ဆိုင်ရာ စကားလုံး အသုံးအနှုန်းတွေကိုလည်း မိတ်ဆက်ပါမယ်။

ဆော့ဖ်ဝဲ အော့ဘ်ဂျက်တွေဟာ အပြင်မှာ တကယ်ရှိတဲ့ အရာတွေရော တကယ်မရှိဘဲ စိတ်ကူးသက်သက်ဖြစ်တဲ့ အိုင်ဒီယာတွေကိုပါ ပရိုဂရမ်ထဲမှာ ထင်ဟပ် ဖော်ပြတယ်။ ဥပမာ ကေသီ့ဘဏ်အကောင့်၊ $\frac{7}{13}$ (အပိုင်းဂဏန်း တစ်ခု)၊ 1948-01-04 (မြန်မာပြည် လွတ်လပ်ရေးရတဲ့နေ့)၊ စန္ဒီပိုင်တဲ့ အနီရောင် တိုယိုတာကား စတာတွေကို အော့ဘ်ဂျက်တွေနဲ့ ဖော်ပြနိုင်တယ်။

အော့ဘ်ဂျက်မှာလည်း တိုက်ပ်သဘောတရား ရှိတယ်။ တိုက်ပ်တူတဲ့ အော့ဘ်ဂျက်အားလုံး ဒေတာဖွဲ့စည်းထားပုံ တူတယ်။ လုပ်ဆောင်လို့ရတဲ့ အော်ပရေးရှင်းတွေလည်း တူပါတယ်။ ဘဏ်အကောင့် အော့ဘ်ဂျက်အားလုံးဟာ လက်ကျန်ငွေနဲ့ အကောင့်နံပါတ် ပါရှိပြီး ငွေသွင်း၊ ငွေထုတ်၊ ငွေလွှဲ အော်ပရေးရှင်းတွေ လုပ်ဆောင်လို့ ရပါမယ်။

အော့ဘ်ဂျက်တွေရဲ့ တိုက်ပ်နဲ့ နီးနီးစပ်စပ် ဆက်နွယ်နေတာကတော့ ကလပ်စ် (*class*) သဘောတရားပါ။ ကလပ်စ်ကို တိုက်ပ်တူအော့ဘ်ဂျက်တွေ ဖန်တီးဖို့အတွက် သတ်မှတ်ထားတဲ့ ပရိုဂရမ်ကုဒ် အစုအဝေးလို့ ယေဘုယျ ပြောနိုင်ပါတယ်။ `Account` ကလပ်စ်၊ `date` ကလပ်စ်၊ `Fraction` ကလပ်စ် စသည်ဖြင့် အော့ဘ်ဂျက် တိုက်ပ် တစ်မျိုးအတွက် ကလပ်စ်တစ်ခု ရှိမှာပါ။ တိုက်ပ်တူ အော့ဘ်ဂျက်တွေ အားလုံးမှာ ပါဝင်မဲ့ အချက်အလက်တွေ၊ အော်ပရေးရှင်းတွေနဲ့ အော့ဘ်ဂျက် ဖန်တီးယူတဲ့ ဖန်ရှင်တွေကို ကလပ်စ်တစ်ခုနဲ့ သတ်မှတ်ရတာပါ။ ကလပ်စ်ကနေ အော့ဘ်ဂျက်တွေ (တိုက်ပ် တူပါမယ်) ထုတ်ယူရတာ ဖြစ်တဲ့အတွက် ကလပ်စ်ကို အော့ဘ်ဂျက် စက်ရုံလို့လည်း ဆိုနိုင်ပါတယ်။ ပုံစံတူ အော့ဘ်ဂျက်တွေ ထုတ်ပေးတာမို့လို့ ကလပ်စ်ဆိုတာ အော့ဘ်ဂျက်တည်ဆောက်တဲ့ blueprint သို့မဟုတ် template ပဲလို့ ယူဆတာ ဟာလည်း သဘာဝကျတယ် ဆိုရမှာပါ။

အော့ဘ်ဂျက်တွေကို အက်ဘ်စရက်ရှင်း (*abstraction*) အနေနဲ့လည်း ရှုမြင်နိုင်တယ်။ ဘယ်လို ဖန်တီး တည်ဆောက်ထားလဲ မသိဘဲ အသုံးပြုလို့ရတဲ့ အရာအားလုံးကို အက်ဘ်စရက်ရှင်းလို့ ဆိုနိုင်တယ်။ အပြင်မှာသုံးကြတဲ့ ကား၊ တီဗီ၊ ကွန်ပျူတာ စတာတွေဟာ အက်ဘ်စရက်ရှင်းတွေ ဖြစ်တယ်။ ဖန်ရှင်တွေဟာလည်း အက်ဘ်စရက်ရှင်းတွေပါပဲ။ အော့ဘ်ဂျက်တွေကတော့ ဒေတာနဲ့ အော်ပရေးရှင်း တွဲဖက်ပေါင်းစပ်ထားတဲ့ အက်ဘ်စရက်ရှင်းတွေပါ။ အော့ဘ်ဂျက်တစ်ခုနဲ့ တွဲဆက်ထားတဲ့ အော်ပရေးရှင်းတွေဟာ

အဲ့ဒီအော့ဘ်ဂျက်ရဲ့ ဒေတာတွေကို အသုံးပြုတယ်။ အဲ့ဒီအော့ဘ်ဂျက်ရဲ့ ဒေတာအပေါ် သက်ရောက်မှု ရှိနိုင်တယ်။ အခြားအော့ဘ်ဂျက်ရဲ့ ဒေတာကို မသုံးဘူး။ သက်ရောက်မှုလည်း မရှိစေဘူး။ အော့ဘ်ဂျက် အတွင်းပိုင်း ဒေတာတွေ ဖွဲ့စည်းထားပုံနဲ့ တိုက်ပ်ကို မသိဘဲ အော့ဘ်ဂျက်ကို အသုံးပြုလို့ရတယ်။ အော်ပရေးရှင်းတွေဟာ တကယ်ကတော့ အော့ဘ်ဂျက်ဒေတာ အသုံးပြုတဲ့ ဖန်ရှင်တွေပါပဲ။ ဒီဖန်ရှင်တွေ ဘယ်လိုရေးထားလဲ၊ ဒေတာကို ဘယ်ပုံဘယ်နည်း အသုံးပြုတာလဲ သိစရာမလိုဘဲ အသုံးပြုလို့ ရပါတယ်။

ဖန်ရှင် အသင့်ရှိပြီးသား ဆိုရင် အသုံးပြုလို့ ရသလို ကလပ်စ် အသင့်ရှိပြီးသား ဆိုရင် အော့ဘ်ဂျက်တွေ ဖန်တီးအသုံးပြုနိုင်ပါတယ်။ ဖန်ရှင်ရေးရတာ ခက်ခဲနိုင်ပါတယ်။ ရှိပြီးသား ဖန်ရှင်သုံးတာကတော့ မခက်ပါဘူး။ ဒီသဘောပါပဲ။ ကလပ်စ်သတ်မှတ်ရတာ၊ ဒီဇိုင်းလုပ်ရတာ ရှုပ်ထွေး ခက်ခဲနိုင်ပါတယ်။ ရှိပြီးသား ကလပ်စ်ကနေ အော့ဘ်ဂျက် ဖန်တီးအသုံးပြုရတာ မခက်ပါဘူး။ အသုံးပြုသူ လွယ်ကူအဆင်ပြေစေဖို့ တည်ဆောက်သူက အဓိကစဉ်းစား ဖြေရှင်းရတာပါ။ သုံးစွဲသူအဆင့်ကနေ စတင်ပြီး တည်ဆောက်သူ ပရိုဂရမ်မာ ဖြစ်လာအောင် တစ်ဆင့်ချင်း တက်လှမ်းဖို့ဟာ အဓိကပန်းတိုင်ပါ။ အော့ဘ်ဂျက်မိတ်ဆက် ခဏရပ်၊ အခုပဲ လက်တွေ့စမ်းသပ် ကြည့်လိုက်ရအောင် . . .

၆.၁ `date`, `time` and `datetime`

ဆော့ဖ်ဝဲ အပ်ပလီကေးရှင်းတွေမှာ အချိန်နာရီ၊ နေ့ရက်တွေနဲ့ တွက်ချက်ဆုံးဖြတ်ရတာတွေ အမြဲလိုလိုပါတယ်။ ဒါကြောင့် အချိန်နဲ့သက်ဆိုင်တဲ့ အချက်အလက်တွေကို စနစ်တကျ ကိုင်တွယ်ဖြေရှင်းတတ်ဖို့ လေ့လာထားရပါမယ်။ အချိန်နဲ့ နေ့ရက်အတွက် `date`, `time`, `datetime` ကလပ်စ်တွေ ထောက်ပံ့ပေးထားတဲ့ `datetime` လိုက်ဘရီ အသုံးပြုပါမယ်။ `date` ကလပ်စ်ကနေ ဖန်တီးယူတဲ့ အော့ဘ်ဂျက်တစ်ခုဟာ အနောက်တိုင်းပြက္ခဒိန် နေ့ရက်တစ်ရက်ကို ဖော်ပြတယ်။ ခုနှစ်၊ လ၊ ရက် အချက်အလက် သုံးခုပါဝင်တယ်။ မြန်မာပြည် လွတ်လပ်ရေးရခဲ့တဲ့ နေ့ရက်ကို ဖော်ပြရင် အခုလိုပါ

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

ဒုတိယလိုင်းက အော့ဘ်ဂျက် ဖန်တီးတာပါ။ ဖန်ရှင်ခေါ်တာနဲ့ ပုံစံတူတာ တွေ့ရတယ်။ အော့ဘ်ဂျက်ဖန်တီးဖို့ စပယ်ရှယ် ဖန်ရှင်တစ်ခု ခေါ်ထားတယ်လို့ ယူဆနိုင်ပါတယ် (နောက်ပိုင်း ကလပ်စ်အခန်းမှာ အသေးစိတ်လေ့လာရမှာပါ)။ ကလပ်စ်ကနေ အော့ဘ်ဂျက် ဖန်တီးယူတာကို *instantiation* လို့ခေါ်ပြီး ရရှိလာတဲ့ အော့ဘ်ဂျက်ကို အဲဒီကလပ်စ်ရဲ့ *instance* လို့လည်း ခေါ်ပါတယ်။

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

အော့ဘ်ဂျက်ကို ဗေရီရေဘဲလ်နဲ့ အဆိုင်းမန့်လုပ်တာပါ။ အော့ဘ်ဂျက်ကို `mmid` ဗေရီရေဘဲလ်နဲ့ ရည်ညွှန်းအသုံးပြုလို့ ရမှာဖြစ်တယ်။

```
>>> mmid.year
1948
```

Dot notation (`.` အမှတ်အသား) နဲ့ အော့ဘ်ဂျက်ရဲ့ ခုနှစ်ကို ရယူထားတာပါ။ လနဲ့ ရက်ကိုလည်း အလားတူနည်းလမ်းနဲ့ ယူကြည့်နိုင်တယ်။

```
>>> mmid.month
1
>>> mmid.day
4
```

အော့ဘ်ဂျက်တစ်ခုမှာ ပါဝင်တဲ့ ဒေတာကို *attribute* လို့ ခေါ်တယ်။ Attribute တွေဟာ အော့ဘ်ဂျက်ရဲ့ လက်ရှိအခြေအနေ (*state*) ကိုဖော်ပြတယ်။ စတိတ် ပြောင်းလဲနိုင်တဲ့ အော့ဘ်ဂျက်တွေကို *mutable object* လို့ ခေါ်တယ်။ စတိတ် မပြောင်းလဲနိုင်တဲ့ အော့ဘ်ဂျက်တွေကို *immutable object* လို့ ခေါ်တယ်။ `date` အော့ဘ်ဂျက်တွေဟာ immutable object တွေပါ။ ဆိုလိုတာက attribute တွေဖြစ်တဲ့ `year`, `month`, `day` တန်ဖိုးတွေ မပြောင်းလဲနိုင်ပါဘူး။

ခုနှစ်၊ လ၊ ရက် အတွက် attribute သုံးခုဟာ `date` အော့ဘ်ဂျက် တစ်ခုစီတိုင်းအတွက် ကိုယ်ပိုင်ပါရှိမှာပါ။ ဆိုလိုတာက အောက်ပါ `usid` အော့ဘ်ဂျက်ရဲ့ attribute တွေနဲ့ ခုနက `mmid` ရဲ့ attribute တွေဟာ သီးခြားစီပဲ။ နံမည်တူပေမဲ့ တစ်ခုနဲ့တစ်ခု မရောယှက်ဘူး။

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

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

အခုလို တန်ဖိုးပြန်ယူကြည့်ရင်လည်း ဖြစ်သင့်တဲ့အတိုင်း သက်ဆိုင်ရာ အော့ဘ်ဂျက်ရဲ့ attribute တန်ဖိုးတွေပဲ ပြန်ရတာပေါ့။ လွတ်လပ်ရေး ရခဲ့တဲ့နေ့က ဘာနေ့ဖြစ်မလဲ

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

ဖန်ရှင်တစ်ခုတည်းကို မတူညီတဲ့ အော့ဘ်ဂျက်နှစ်ခုအပေါ်မှာ အသုံးချတာ ဖြစ်တယ်။ ဒေါ့ထ်ကိုပဲ သုံးတယ်။ ပထမတစ်ခုက `usid` အော့ဘ်ဂျက်၊ နောက်တစ်ခုက `mmid` အော့ဘ်ဂျက်အပေါ်မှာ သုံးထားတာပါ။ အမေရိကန် လွတ်လပ်ရေးရခဲ့တာ ကြာသာပတေးနေ့၊ မြန်မာကတော့ တနင်္ဂနွေနေ့ပါ (တနလင်္ဘာက တစ်၊ တနင်္ဂနွေက ခုနှစ်ပါ)။ `isoweekday` ဖန်ရှင်ကို အော့ဘ်ဂျက်တစ်ခုအပေါ် အသုံးပြုတဲ့အခါ ၎င်းအော့ဘ်ဂျက်နဲ့ သက်ဆိုင်တဲ့ ဒေတာနဲ့ ဖန်ရှင်က အလုပ်လုပ်သွားတာပါ။ ဒါကြောင့်လည်း attribute မတူတဲ့ အော့ဘ်ဂျက်တွေအပေါ်မှာ အသုံးချတဲ့အခါ မတူညီတဲ့ ရလဒ်တွေ ထွက်လာရတာပေါ့။ 'ဒေတာနဲ့ အော်ပရေးရှင်း တွဲဖက်ထားတယ်' ဆိုတာ ဒီသဘောတရားကို ဆိုလိုတာပါ။ အော့ဘ်ဂျက်ဒေတာနဲ့ တွဲဖက်အလုပ်လုပ်တဲ့ ဖန်ရှင်တွေကို မက်သဒ် (*method*) လို့ ခေါ်တယ်။ နောက် မက်သဒ်တစ်ခုက `isoformat` ပါ။ နေ့ရက်ကို စာသားအဖြစ် `'yyyy-mm-dd'` ဖော့မတ်နဲ့ ပြန်ပေးတယ်။

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

`date` တစ်ခု ဖန်တီးတဲ့အခါ ခုနှစ်၊ လ၊ ရက် နေရာ မှန်ဖို့ အရေးကြီးပါတယ်။ `date(1948,4,1)` လို့ ရေးမိရင် လေးလပိုင်း တစ်ရက်နေ့ ဖြစ်သွားမှာပါ။ ဒါပေမဲ့ Python မှာ အော့ဂုမန့်တွေကို နံမည်နဲ့ တွဲပြီး ထည့်ပေးလို့ရတယ်။

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

ဒီနည်းနဲ့ ဆိုရင်တော့ year, month, day ကြိုက်သလို အစီအစဉ်နဲ့ ထည့်လို့ရမှာပါ။

replace မက်သဒ် ဘယ်လို အလုပ်လုပ်လဲ ကြည့်ရအောင်

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

နဂို usid နေ့ရက်ရဲ့ ခုနှစ်ကို 1876 နဲ့ အစားထိုးထားတဲ့ အော့ဘ်ဂျက် အသစ်တစ်ခု ပြန်ရပါတယ်။ နဂို ရက်စွဲက မပြောင်းသွားဘူး (date အော့ဘ်ဂျက် 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)
```

အော့ဂုမန့် နံမည် 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 တခုခု ချန်ထားကြည့်ပါ

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

ချန်ထားခဲ့တာတွေ နဂိုအတိုင်းရှိပါမယ်

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

ရက်တစ်ခုတည်း အစားထိုးမယ်ဆိုရင် နံမည်နဲ့တွဲတဲ့ နည်းကပဲ အဆင်ပြေပါမယ်

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

နံမည်မပါရင် ခုနှစ်၊ လ၊ ရက် အစဉ်အတိုင်းဖြစ်တာကြောင့် ခုနှစ်ကို အစားထိုးမှာပါ

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

time and datetime

နေ့ရက်နဲ့ အချိန် တွဲရက်ကို datetime, ရက်စွဲမလိုဘဲ အချိန်ပဲဆိုရင် 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 ကြာချိန် (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 အော်တို ထည့်ပေးသွားတာ။ ကိုယ်တိုင် ရိုက်ထည့်စရာမလိုဘူး။ တစ်လိုင်းချင်း Enter ခေါက်သွားရုံပဲ။ အပိတ်ဝိုက်ကွင်းမှာ စတိတ်မန့် ဆုံးတယ်ဆိုတာ အင်တာပရက်တာက နားလည်တယ်။)

အော့ဘ်ဂျက် အသုံးပြုသူအနေနဲ့ အချိန်ကာလ ကြာမြင့်ချိန်ကို days, weeks, hours ... စတာတွေနဲ့ သတ်မှတ်လို့ရတယ်။ ၎င်းတို့ကို ရက်၊ စက္ကန့်၊ မိုက်ခရိုစက္ကန့် ဖွဲ့ပြီး အော့ဘ်ဂျက် အတွင်းပိုင်း days, seconds, microseconds attributes ဒေတာအနေနဲ့ သိမ်းမှာပါ။

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

ဖော်ပြခဲ့ပြီးတဲ့ အော့ဘ်ဂျက်တွေနဲ့ အပေါင်း၊ အနှုတ် အော်ပရေးရှင်းတွေ လုပ်လို့ရပါတယ်။ ဥပမာ တချို့ 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
```

၆.၂ list

List ဆိုတာ အိုက်တမ် item တွေ အတွဲလိုက် စုစည်းထားဖို့ အသုံးပြုတဲ့ စထရက်ချာတစ်မျိုး ဖြစ်တယ်။ Python မှာ list အော့ဘ်ဂျက်တွေကို item တွေ အတွဲလိုက် စုစည်းထားဖို့ သုံးတယ်။ ဘာအိုက်တမ်မှ မပါတဲ့ list အသစ်တစ်ခု လိုချင်ရင် ဒီလို

```
>>> odds = list()
```

ဒီ lsit ထဲမှာ ပါတွေပါလဲ

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

လေးထောင့်ကွင်းနဲ့ list ကိုပြပေးတယ်။ လက်ရှိ list ထဲမှာ အိုက်တမ် မရှိသေးဘူး။ အိုက်တမ် တစ်ခုချင်း ထည့်ချင်ရင် append မက်သဒ်ရှိတယ်

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

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

ပါဝင်တဲ့ အိုက်တမ်တစ်ခုစီကို ကော်မာခြားပြီး ပြတယ်။ နောက်ထပ် နည်းလမ်းတစ်ခုနဲ့လည်း list ဖန်တီးလို့ ရတယ်။ လေးထောင့်ကွင်း သုံးတဲ့နည်းပါ

```
>>> 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]
```

နဂို အော့ဘ်ဂျက်မှာ အိုက်တမ်တစ်ခု ထပ်တိုးသွားတာ။ အော့ဘ်ဂျက်ရဲ့ စတိတ် ပြောင်းသွားတယ်။ အိုက်တမ်တစ်ခုကို ဖယ်ထုတ်ချင်ရင်

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

list အိုက်တမ်တွေရဲ့ တည်နေရာကို index လို့ခေါ်တယ်။ သုညနဲ့ စတယ်။ ဒုတိယက တစ်၊ တတိယက နှစ် စသည်ဖြင့် ဖြစ်မယ်။ အခု odds မှာ အိုက်တမ် လေးခုရှိနေတယ်။ 7 ဖြုတ်ချင်ရင် index နံပါတ် 2 ကို pop လုပ်ရမှာ

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

ဖယ်လိုက်တဲ့ အိုက်တမ်တွေ နဂို့နေရာမှာ 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 အိုက်တမ်ကို ဖယ်ထုတ်လိုက်တယ်။ စတိတ်ပြောင်းလဲစေတယ်။ အခုနည်းက မဖယ်ထုတ်ဘူး။ အိုက်တမ်ကိုပဲ 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။

ဗေရီရေဘဲလ် နှစ်ခုက အော့ဘ်ဂျက်တစ်ခုတည်းကို ရည်ညွှန်းနေရင် သတိပြုသင့်တဲ့ ထူးခြားချက်တွေ ရှိလာပါတယ်။

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

ဒီလိုဆိုရင် အော့ဘ်ဂျက် တစ်ခုတည်းကို fruits ရော myfav နဲ့ပါ သုံးလို့ရပါမယ်။ fruits နဲ့ အိုက်တမ်တစ်ခု ထပ်ထည့်ကြည့်မယ်

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

myfav ပါ လိုက်ပြောင်းတာကို တွေ့ရမှာပါ

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

Mutable အော့ဘ်ဂျက်တွေမှာ ဒီအချက်ကို သတိပြုဖို့ လိုပါတယ်။ ဗေရီရေဘဲလ် တစ်ခုကနေ အော့ဘ်ဂျက် စတိတ်ကို ပြောင်းလဲတဲ့အခါ အဲ့ဒီအော့ဘ်ဂျက်ကို ရည်းညွှန်းတဲ့ အခြား ဗေရီရေဘဲလ် အားလုံးက လည်း အပြောင်းအလဲကို မြင်ရမှာ ဖြစ်တယ်။ Immutable ဆိုရင်တော့ စတိတ်မပြောင်းနိုင်တာကြောင့် ဒီလိုကိစ္စမျိုး စဉ်းစားစရာ မလိုဘူး။

အခန်း ၇

ကွန်ထရိုးလ် စတိတ်မန့်များ

ကွန်ထရိုးလ် စတိတ်မန့်တွေက ကားရဲလ်မှာ တွေ့တော့ တွေ့ခဲ့ပြီးသားပါ။ ဒါပေမဲ့ ကားရဲလ်ပရိုဂရမ်မင်းအတွက် လိုသလောက် အခြေခံကိုပဲ ကန့်သတ်ဖော်ပြခဲ့တာပါ။ ဒီအခန်းမှာ ပြည့်စုံအောင် အသေးစိတ်ဆက်လက် လေ့လာကြပါမယ်။ လက်တွေ့အသုံးချ ဥပမာတွေ၊ လေ့ကျင့်ခန်းတွေ ဂရုတစိုက် စီစဉ်ပေးထားတယ်။ စတိတ်မန့် အသစ်တချို့လည်း တွေ့ရမယ်။ စာအုပ်တွေမှာ ဖော်ပြတာ သိပ်မတွေ့ရပေမဲ့ ဘီဂင်နာ အများစု အခက်အခဲတွေ့ကြတဲ့ နေရာတွေ၊ တိတိကျကျ နားလည်ဖို့ လိုတဲ့ ပို့ချင့်တွေကိုလည်း အလေးပေးရှင်းပြထားတယ်။ အထူးခြားဆုံးကတော့ ရုပ်ပုံတွေဆွဲတာနဲ့ အန်နီမေးရှင်း အခြေခံကို Arcade ဂိမ်းလိုက်ဘရီ အသုံးပြုပြီး စတင်မိတ်ဆက်ထားတာပဲ ဖြစ်တယ်။ စာသားတွေချည်းပဲထက် စိတ်လှုပ်ရှားဖို့ ပိုကောင်းမယ်ထင်ပါတယ်။

၇.၁ `if` စတိတ်မန့်

စားသောက်ဆိုင် တစ်ဆိုင်က ကျပ်ငွေ $50,000$ နဲ့ အထက် သုံးတဲ့ ကတ်စတမ်မာတွေကို 10% လျှော့ပေးပြီး ပရိုမိုးရှင်း လုပ်တယ် ဆိုပါစို့။ ကီးဘုဒ်ကနေ ကျသင့်ငွေ ရိုက်ထည့်ပေးရမယ်။ ဒစ်စကောင့် ရမဲ့ ကတ်စတမ်မာအတွက်ပဲ 'Get 10 % discount.' ပြပေးပြီး လာရောက် စားသုံးတဲ့အတွက် ကျေးဇူးတင်ကြောင်း 'Thanks for coming!' ကိုတော့ ကတ်စတမ်မာတိုင်းကို ပြပေးချင်ပါတယ်။

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

`amt > 50_000` ဘူလီယန် အိပ်စ်ပရက်ရှင် `true` ဖြစ်မှပဲ `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 ဗားရှင်း 3.6 ကစပြီး 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 ဗေရီရေဘဲလ် (သို့) အိပ်စ်ပရက်ရှင် တွေကို တွန့်ကွင်းထဲမှာ ထည့်ရေးလို့ရတယ်။ ၎င်းတို့ကို f-string က တန်ဖိုးရှာပြီး အစားထိုးပေးမှာပါ။ ဒါကြောင့် {2 + 3} က 5 ဖြစ်သွားတာပါ။ ရိုးရိုး string နဲ့ဆို အခုလို

```
>>> '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 ရမှာဖြစ်ကြောင်းပြောပြီး ဆွဲဆောင်လေ့ရှိတယ်။ ဒစ်စကောင့်ရအောင် ဘယ်လောက်ထပ်သုံးရမလဲ ပရိုဂရမ်က ပြပေးချင်တယ် 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` ယေဘုယျပုံစံကို 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 ကွန်ဒီရှင်တွေဟာ ဘူလီယန်တန်ဖိုး ပြန်ပေးတဲ့ predicate မက်သဒ်တွေပါ။ predicate မက်သဒ်တွေကို ဘူလီယန် အိပ်စ်ပရက်ရှင်းလို့ ယူဆနိုင်တယ်။)

အခုဆက်ကြည့်ကြမဲ့ `if...elif...else` ပုံစံကတော့ ရှေ့ပိုင်းမှာ မတွေ့ဖူးသေးဘူး။ "Cascading `if` statement" လို့ခေါ်တယ်။ အောက်ပါ ဇယားအရ စာမေးပွဲရမှတ် ကနေ grading ထုတ်ပေးမယ် ဆိုပါစို့။

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

တေဘဲလ် ၇.၁ Score and Grading

ကျောင်းသူ/သား နံမည်နဲ့ ရမှတ်ကို ထည့်ပေးရင် ပရိုဂရမ်က အခုလို ပြပေးရပါမယ်။

```
Student name: Amy
Score: 95
Amy get grade A
```

ဒီပရိုဂရမ် အတွက် cascading `if` သုံးထားတာ 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 <= 69:
    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 အတွဲလိုက် တွေ့ရပါမယ်။ နောက်ဆုံးမှာ `else` အပိုင်းကို တွေ့ရတယ် (ဒီအပိုင်းက optional ပဲ၊ မပါလို့လဲရတယ်။ 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 မပါလို့လည်းရတယ်။ အပေါ်မှာ တစ်ခုမှ `True` မဖြစ်တော့မှ `else` ဘလောက်ကို လုပ်ဆောင်တယ်။ နောက်ဆုံးမှာ `else` အပိုင်းမပါဘူး၊ အပေါ်မှာလည်း ဘယ်တစ်ခုကမှ `True` မဖြစ်ဘူး ဆိုရင်တော့ လုပ်ဆောင်ပေးစရာ ဘလောက်လည်း မရှိဘူးပေါ့။ ဒီတော့ အားလုံး `False` ဖြစ်ခဲ့ရင် လုပ်ချင်တဲ့ကိစ္စ ရှိ/မရှိ အပေါ်မူတည်ပြီး `else` အပိုင်း လို/မလို ဆုံးဖြတ်ရတယ်။

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

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

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

အခုပရိုဂရမ်မှာ သိပ်စိတ်တိုင်းကျစရာ မကောင်းတဲ့ ပြဿနာတစ်ခုတွေ့ရပါတယ်။ 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 <= 79:
        grade = 'C'
    elif 60 <= score <= 69:
        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.')
```

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

Cascading `if` မှာ ဘူလီယန် အိပ်စ်ပရက်ရှင် တစ်ခုချင်းကို အထက်အောက် အစဉ်အတိုင်း တန်ဖိုးရှာတယ်၊ 'ပထမဆုံး `True` ဖြစ်တဲ့ ဘလောက် တစ်ခုကိုပဲ လုပ်ဆောင်ပေးတယ်' ဆိုတဲ့အချက်ကို နားလည်ဖို့ အရေးကြီးတယ်။ အောက်ကို ရောက်လာတာဟာ အပေါ်မှာ မှားခဲ့လို့ပဲ။ တစ်ခုမှန်ပြီဆိုတာနဲ့ သက်ဆိုင်တဲ့ ဘလောက်ကို လုပ်ဆောင်ပြီး cascading `if` တစ်တွဲလုံး ပြီးဆုံးသွားမှာ ဖြစ်တယ်။ အောက်ပိုင်းက `elif` (သို့) `else` တွေကို မရောက်လာတော့ဘူး။ ဒီအကြောင်းကြောင့် grading အတွက် အခုလိုလည်း ရေးလို့ရတယ်

```
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 ကိုးဆယ်အောက် ဖြစ်မှာတော့ သေချာတယ် (score >= 90 မဟုတ်လို့ ဒီကို ရောက်လာတာ)၊ ဒါကြောင့် ရှစ်ဆယ်နဲ့အထက် (score >= 80) ဖြစ်လား စစ်ရင်ရပြီ။ နောက်တစ်ဆင့်ကို ရောက်လာရင် ရှစ်ဆယ်အောက် မို့လို့ သေချာတယ်၊ score >= 70 ဖြစ်လားစစ်ရုံပဲ။ စသည်ဖြင့် အောက်အဆင့်တွေ အတွက်လည်း ထိုနည်းတူစွာ စဉ်းစားနိုင်တယ်။

Cascading if မသုံးဘဲ if...else တွေနဲ့လည်း ရေးလို့တော့ ရပါတယ်။ Nesting လုပ်တာတွေ အရမ်းများပြီး ဖတ်ရမလွယ်ကူတာ တွေ့ရမှာပါ။ 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။ စတိတ်မန့်တစ်စုကို သတ်မှတ်ထားတဲ့ အကြိမ်အရေအတွက် ပြည့်အောင် ထပ်ခါထပ်ခါ လုပ်ဆောင်ဖို့ လိုတဲ့အခါ for loop ကို အသုံးပြုတယ်။ Loop ကို စတင် လုပ်ဆောင်တဲ့အချိန်မှာ ဘယ်နှစ်ကြိမ် ပြန်ကျော့မလဲ အတိအကျ ကြိုသိရင် *definite loop* လို့ သတ်မှတ်တယ်။ for loop ဟာ definite loop ဖြစ်ပါတယ်။ 'အဆင့်မြင့် အက်ဘ်စရက်ရှင်း' လို့ ပြောရတာက list, dictionary, set, range စတဲ့ စထရက်ချာ အမျိုးမျိုးနဲ့ အသုံးပြုလို့ရတဲ့ အတွက်ကြောင့်ပါ။

for loop နဲ့ list ထဲက အိုက်တမ်တစ်ခုချင်း ထုတ်ယူအသုံးပြုနိုင်ပါတယ် . . .

```
fruits = ['Orange', 'Kiwi', 'Banana', 'Papaya', 'Apple', 'Plum', 'Mango']
for itm in fruits:
    print(itm)
```

Output:

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

Loop တစ်ခေါက်ပြန်ကျော့တိုင်း itm ဗေရီရေဘဲလ်ထဲမှာ list အိုက်တမ်တစ်ခုချင်း အစဉ်အတိုင်း ထည့်ပေးမှာပါ။ အိုက်တမ်တွေကို နံပါတ်စဉ်နဲ့ တွဲချင်ရင် enumerate လုပ်ပြီး အခုလို ထုတ်လို့ရတယ်

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

Output:

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

နံပါတ်စဉ်ကို idx၊ အိုက်တမ်ကို itm နဲ့ ယူသုံးထားတာပါ။ str တစ်ခုထဲက ကာရက်တာတစ်လုံးချင်း လိုချင်ရင်လည်း ရတာပဲ

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

Output:

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

List နှစ်ခုရဲ့ cartesian product ပါ (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 စတဲ့ စထရက်ချာအမျိုးမျိုးနဲ့ အလုပ်လုပ်နိုင်တာ တွေ့ရမှာပါ။ တစ်ခါကျော့တိုင်း အိုက်တမ်တစ်ခုကို loop ဗေရီရေဘဲလ်ထဲမှာ ထည့်ပေးထားတယ်။ အိုက်တမ်တွေအားလုံး ပြီးတဲ့အခါ `for` loop ရပ်သွားမှာ ဖြစ်တယ်။ ပြန်ကျော့တဲ့ အကြိမ်အရေအတွက်ဟာ အိုက်တမ်အရေအတွက်ပဲ ဖြစ်တယ်။

`range` ဖန်ရှင်နှင့် `for` loop

သုညကနေ တစ်ဆယ်ထိ အစဉ်အတိုင်း ရေတွက်ချင်ရင် နည်းလမ်းတစ်ခုက

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

ဒီလိုသာ ဂဏန်းတစ်လုံးချင်း ရိုက်ထည့်ရရင် အဆင်မပြေပါဘူး။ ပိုကောင်းတဲ့ နည်းလမ်း ရှိရမှာပါ။ `range` ဖန်ရှင်ဟာ ဒီလိုနေရာမျိုးအတွက် အသင့်တော်ဆုံးပါပဲ။ `range(0,11)` က သုညကနေ တစ်ဆယ်ထိ အစဉ်အတိုင်း ထုတ်ပေးမဲ့ `range` အော့ဘ်ဂျက်ကို ပြန်ပေးတယ်။ ဒီဂဏန်းတွေကို တစ်ခါတည်း ကြိုထုတ်ထားတာ မဟုတ်ပါဘူး။ လိုအပ်မှ တစ်ခုချင်း ထုတ်ပေးတာပါ။ (ဒီအတွက်ကြောင့် `range(0, 11)` နဲ့ `range(0, 1_000_000)` နှစ်ခုလုံး မမ်မိုရီသုံးစွဲတာအရ သိပ်မကွာခြားပါဘူး)။

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

$2, 4, 6, \ldots, 12$ စုံကိန်းတွေ လိုချင်ရင် `range(2, 13, 2)`၊ $5, 8, 11, \ldots, 98$ လိုချင်ရင် `range(5, 99, 3)` စသည်ဖြင့် သုံးခုထည့်ပြီး ဖန်ရှင်ခေါ်ရုပါမယ်။ အစ၊ အဆုံးနဲ့ နောက်ဆုံးတစ်ခုကတော့ ကိန်းတန်းမှာပါတဲ့ ကပ်လျက်ကိန်းနှစ်ခုရဲ့ ကွာခြားချက်ပါ။

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

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

အစဂဏန်း မသတ်မှတ်ပေးရင် သုညလို့ ယူဆတယ်။ ဒါကြောင့် သုညကနေ တစ်ဆယ်ထိကို `range(11)` နဲ့ ယူလို့ရတယ်။ ကွာခြားချက်က အနှုတ်ကိန်း ဖြစ်လို့ရတယ်

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

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

မှတ်ချက်။ ။ ကွာခြားချက် တစ်မဟုတ်ရင် အစ၊ အဆုံး၊ ကွာခြားချက် သုံးခုလုံး လိုပါမယ်။

for loop အသုံးချ ဥပမာများ

စတိတ်မန့်တစ်စုကို သတ်မှတ်ထားတဲ့ အကြိမ်အရေအတွက် ပြည့်အောင် ပြန်ကျော့ဖို့ for loop ကို အသုံးပြုနိုင်ပါတယ်။ ကီးဘုဒ်ကနေ ထည့်ပေးတဲ့ ဂဏန်း ဆယ်လုံးကို ပေါင်းမယ်ဆိုပါစို့။

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

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

ပရိုဂရမ် run တဲ့ အချိန်ကျတော့မှ အကြိမ်အရေအတွက် သတ်မှတ်လို့လည်း 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 ဗေရီရေဘဲလ် တန်ဖိုးတွေ ဘယ်လောက်ဖြစ်နေလဲ အခုလို လိုက်ကြည့်နိုင်ပါတယ်။

```
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)

3rd iter:
i = 2
```

```
val = float(input("? "))     # 4 ထည့်တယ် ယူဆပါ
tot += val                   # 9 (လက်ရှိ tot တန်ဖိုး)
```

4th iter:

```
i = 3
val = float(input("? "))     # 11 ထည့်တယ် ယူဆပါ
tot += val                   # 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 ဗေရီရေဘဲလ် နှစ်ခု သုံးထားတာ သတိထားကြည့်ပါ။ ဒီနေရာမှာ ဗေရီရေဘဲလ် စကုပ် (*scope*) သဘောတရားကို နားလည်ဖို့ လိုအပ်လာပါတယ်။ ဗေရီရေဘဲလ်တစ်ခုကို ဘယ်နေရာကနေ သုံးလို့ရလဲဆိုတာဟာ ၎င်းဗေရီရေဘဲလ်ရဲ့ စကုပ်နဲ့ သက်ဆိုင်ပါတယ်။ `grand_tot` ဟာ top level ဗေရီရေဘဲလ် ဖြစ်တယ်။ ၎င်းကို ကြေငြာတဲ့ နေရာကစပြီး အောက်ပိုင်းတလျှောက်လုံး သုံးလို့ရတယ်။ `grand_tot` ရဲ့ စကုပ်ဟာ ၎င်းကို ကြေငြာထားတဲ့ ဖိုင်အဆုံးထိ ဖြစ်တယ်။ `rot_tot` ကတော့ block level ဗေရီရေဘဲလ်ပါ။ Block level ဗေရီရေဘဲလ်ကိုတော့ ၎င်းကို ကြေငြာထားတဲ့ ဘလောက်အတွင်းမှာပဲ သုံးလို့ရပါမယ်။ Block level ဗေရီရေဘဲလ်တစ်ခုရဲ့ စကုပ်ဟာ ၎င်းကို ကြေငြာထားတဲ့ ဘလောက် အဆုံးထိ ဖြစ်တယ်။

`row` တစ်ခုချင်း ပေါင်းလဒ်ကို `grand_tot` မှာ 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 ၎င်းကို အပြင်ဘလောက်မှာ ကြေငြာရမယ်။ အတွင်းဘလောက်မှာဆိုရင် အပြင်ကနေ သုံးလို့ရမှာ မဟုတ်တော့ဘူး။ (အတွင်း `for` loop ဟာ အပြင် `for` loop ရဲ့ ဘလောက် အတွင်းမှာ ပါဝင်တဲ့အတွက် `rot_tot` ကို သုံးလို့ရပါတယ်)။

Loop တစ်ကျော့ပြီး တစ်ကျော့ ဗေရီရေဘဲလ် တန်ဖိုးတွေ ပြောင်းလဲသွားတာကို ပြထားတယ်။ တစ်ဆင့်ချင်း ဂရုစိုက်ပြီး လိုက်ကြည့်ပါ။ (အကြိမ်အရေအတွက် သိပ်မများအောင် အိုက်တမ် နည်းတဲ့

list နဲ့ ဥပမာ ပြထားတာပါ)။

```
# ဒီ list ထဲက ဂဏန်းတွေ ပေါင်းမှာပါ
rows = [[1, 3, 5],
        [2, 4, 6]]
```

```
grand_tot = 0

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

        # အတွင်း 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
    # အပြင်ဘလောက် တစ်ကျော့ပြီး row_tot စကုပ် အဆုံး

    # အပြင် for loop
    2nd iter:
    row = [2, 4, 6]
    row_tot = 0 # ဗေရီရေဘဲလ် တစ်ခါထပ်ကြေငြာတယ်

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

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

        3rd iter:
        val = 6
```

```
        row_tot += val          # 12

    print('Row total: ' + str(row_tot))
    grand_tot += row_total  # 21
    # အပြင်ဘလောက် နောက်တစ်ကျော့ပြီး row_tot စကုပ် အဆုံး

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

ခရစ်စမတ် သစ်ပင်လေး ဆွဲကြည့်ရအောင်။ တစ်တန်းမှာ စပေ့စ်ဘယ်နှစ်ခုပါလဲ ကြည့်ရလွယ်အောင် □ လေးတွေနဲ့ 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 တြိဂံမှာ အတန်း ရှစ်ခု (LEAF_ROWS)၊ ပင်စည်မှာ သုံးခု (TRUNK_ROWS) သတ်မှတ်

ထားတယ်။ ပင်စည် အကျယ် (TRUNK_SZ) ကိုလည်း * သုံးခု ထားတယ်။ အပေါ်ဆုံးကို row နံပါတ် သုည၊ ဒုတိယကို တစ် စသည်ဖြင့် ယူဆပါမယ်။ row နံပါတ်စဉ်ကို r ဗေရီရေဘဲလ်က ဖော်ပြတယ်။ တြိဂံပုံမှာ စပေ့စ်အရေအတွက်နဲ့ row နံပါတ်စဉ်ဟာ (LEAF_ROWS - 1 - r) ဖော်မြူလာနဲ့ ဆက်စပ်နေတယ်။ * အရေအတွက်ကတော့ နှစ်ခုစီတိုးသွားပြီး (r * 2 + 1) ဖြစ်တယ်။ ပင်စည်ပိုင်း စပေ့စ် အရေအတွက်ကတော့ တစ်တန်း ခြောက်ခုပါ (LEAF_ROWS - 2) ။

၇.၃ Python Arcade Library

Python Arcade (ဝက်ဘ်ဆိုက် https://api.arcade.academy) ဟာ အရည်အသွေးကောင်းတဲ့ ထိပ်ဆုံး Python ဂိမ်းလိုက်ဘရီတွေထဲက တစ်ခုဖြစ်တယ်။ Arcade အပြင် အသုံးများတဲ့ အခြားတစ်ခုက pygame (ဝက်ဘ်ဆိုက် https://www.pygame.org) ပါ။ ဒီစာအုပ်မှာ Arcade ကို သုံးပါမယ်။ Arcade ပိုကောင်းတယ်လို့ မဆိုလိုပါဘူး။ ဘီဂင်နာတွေအတွက် ပိုသင့်တော်မယ် ယူဆတဲ့အတွက် အသုံးပြုတာပါ။ အောက်ပါအတိုင်း အင်စတောလ်လုပ်နိုင်ပါတယ်

```
pip install arcade
```

Arcade နဲ့ ပုံဆွဲဖို့အတွက် ဂရပ်ဖစ် ဝင်းဒိုးတစ်ခုကို အောက်ပါအတိုင်း ယူရပါတယ်။ Run လိုက်ရင် ပုံ (၇.၁) မှာ တွေ့ရတဲ့ နောက်ခံရောင်နဲ့ ဝင်းဒိုးအလွတ် တစ်ခု တက်လာမှာပါ။

```python
# File: arcade_starter.py
import arcade

arcade.open_window(300, 200, "Arcade Starter")
arcade.set_viewport(left=0, right=300, top=0, bottom=200)

# 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။ ရှေ့ပိုင်းမှာသုံးတဲ့ from *lib* import * ပုံစံ နဲ့ ကွာခြားတာက အခုနည်းနဲ့ အင်ပို့လုပ်ထားရင် လိုက်ဘရီမှာ ပါတဲ့ အစိတ်အပိုင်းတွေကို ဒေါ့ထ် အမှတ်အသားနဲ့ အသုံးပြုရပါမယ်။ ဥပမာ open_window ဖန်ရှင်ကို

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

လိုက်ဘရီ နံမည်နောက်မှာ (.) အမှတ်အသား ခံပြီး ခေါ်ရမှာပါ။ ဒီဖန်ရှင်မှာ လိုချင်တဲ့ဝင်းဒိုး အကျယ်၊ အမြင့်၊ တိုက်တယ်လ်စာသား ထည့်ပေးထားတယ်။

ပုံ ၇.၁

`set_viewport` ဖန်ရှင်က ဝင်းဒိုးနေရာယူထားတဲ့ စခရင်ဧရိယာမှာ ကိုညြဒိနိတ်စစ်စတမ် သတ်မှတ်ပေးတာပါ။ Origin အမှတ်နဲ့ x, y ဒါရိုက်ရှင် သတ်မှတ်ပေးတာပါ။

```
arcade.set_viewport(left=0, right=300, top=0, bottom=200)
```

ဝင်းဒိုး ဘယ်ဘက်စွန်းကို $x = 0$, ညာဘက်စွန်းကို $x = 300$ သတ်မှတ်ထားတယ်။ အပေါ်ဘက်စွန်း (တိုက်တယ်လ်ဘားမပါ) ကို $y = 0$, အောက်ဘက်စွန်းကို $y = 200$ သတ်မှတ်ထားပါတယ်။ ဝင်းဒိုးရဲ့ ဘယ်ဘက်အပေါ်ထောင့်စွန်းက origin အမှတ် $(0, 0)$ ဖြစ်တယ်။ ဘယ်ဘက်ကို သွားရင် x တန်ဖိုး တိုးသွားပြီး အောက်ဘက်ကို ဆင်းရင် y တန်ဖိုး တိုးသွားမှာဖြစ်တယ်။ အခုလို ကိုယ်တိုင် မသတ်မှတ်ပေးဘဲ သူ့နဂိုအရှိအတိုင်းဆိုရင် 'အောက်ခြေ' ဘယ်ဘက်စွန်းက origin ဖြစ်နေမှာပါ။ အောက်ခြေကနေ အပေါ်ကိုတက်သွားရင် y တန်ဖိုး များလာမှာပါ။ သူ့နဂိုအတိုင်းက `set_viewport` ကို အခုလိုခေါ်ထားတာဖြစ်မယ်။ `top=200, bottom=0` ပါ။

```
arcade.set_viewport(left=0, right=300, top=200, bottom=0)
```

အသုံးများတဲ့ ဂိမ်းလိုက်ဘရီတွေမှာ ဒီစနစ်ကို သုံးကြတာ မတွေ့ရဘူး။ ဒီစနစ်နဲ့ အကျင့်ဖြစ်သွားရင် အခြားလိုက်ဘရီတွေ လေ့လာတဲ့အခါ အခက်အခဲရှိနိုင်တယ်။ အခုလေ့လာကြမဲ့ ဥပမာတွေ အတွက်လည်း အပေါ်မှာပြောတဲ့နည်းက ပိုအဆင်ပြေတယ်။ (Arcade ကိုပဲ တစိုက်မတ်မတ်သုံးမယ်၊ လေ့လာမယ်ဆိုရင်တော့ သူ့နဂိုအတိုင်းသုံးတာ အကောင်းဆုံးဖြစ်မှာပါ)။

အောက်ပါ ဖန်ရှင်ကတော့ ဝင်းဒိုးရဲ့ နောက်ခံရောင် သတ်မှတ်တာပါ။ လိုက်ဘရီရဲ့ `color` မော်ဂျူး (*module*) မှာ အရောင်တန်ဖိုးတွေ အဆင်သင့် သတ်မှတ်ပေးထားတယ်။ (မော်ဂျူးဆိုတာ လိုက်ဘရီရဲ့ အစိတ်အပိုင်းတစ်ခုလို့ အကြမ်းဖျဉ်း ယူဆနိုင်တယ်)။

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

အခုလို အင်ပို့လုပ်ထားရင် အရောင်တွေ သုံးရတာ ပိုအဆင်ပြေတယ်

```
import arcade
from arcade.color import *
...
arcade.set_background_color(PINK_PEARL)
```

`PINK_PEARL`, `RED` စသည်ဖြင့် အရောင်နံမည် တမ်းရေးလို့ရတယ်။ ရှေ့မှာ `arcade.color.` ထည့်ဖို့ မလိုတော့ဘူး။

ပုံဆွဲဖို့ အဆင်သင့်ဖြစ်အောင် start_render ခေါ်ပေးရမယ်။ ဆွဲပြီးရင်လည်း finish_render ခေါ်ဖို့လိုတယ်။ ပုံဆွဲတဲ့ကိစ္စကို ၎င်းတို့နှစ်ခုကြားမှာ လုပ်ရမှာပါ။

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

```
arcade.run()
```

ဝင်းဒိုးကို မပိတ်မချင်း ပေါ်နေအောင် run ဖန်ရှင် ခေါ်ပေးရတာပါ။ မခေါ်ထားဘဲ ပရိုဂရမ်ကို run ရင် ဝင်းဒိုးပွင့်လာပြီး ဖျတ်ခနဲ ပြန်ပိတ်သွားမှာပါ။ မကျန်ခဲ့ဖို့ သတိပြုရပါမယ်။

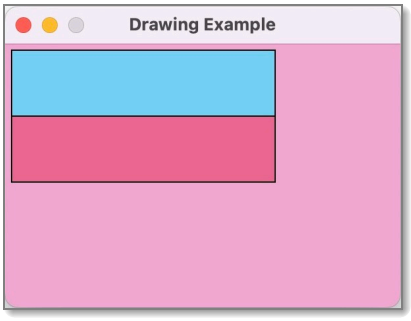
Arcade မှာ ပါတဲ့ အခြေခံ ပုံဆွဲဖန်ရှင် တချို့ကို ဆက်ကြည့်ရအောင်။ ထောင့်မှန်စတုဂံဆွဲတဲ့ ဖန်ရှင်တွေထဲက နှစ်ခု သုံးပြထားတယ်။ နှစ်ခုလုံးက ထောင့်မှန်စတုဂံရဲ့ ဘယ်ဘက်အပေါ်ထောင့်စွန်းနဲ့ တည်နေရာကို သတ်မှတ်ပြီး အရွယ်အစားကို အကျယ်၊ အမြင့်နဲ့ သတ်မှတ်ပေးရတာပါ။ draw_xywh_rectangle_filled က အတွင်းပိုင်း အရောင်နဲ့ ဆွဲပေးတယ်။ အနားတွေကိုပဲ ဆွဲချင်ရင် draw_xywh_rectangle_outline ဖန်ရှင်သုံးရပါမယ်။

```
import arcade
from arcade.color import *

arcade.open_window(300, 200, "Drawing Example")
arcade.set_viewport(0,300, 200, 0)

arcade.set_background_color(PINK_PEARL)
arcade.start_render()
# start drawing
arcade.draw_xywh_rectangle_filled(5,5,200, 50,BABY_BLUE)
arcade.draw_xywh_rectangle_outline(5,5,200, 50,BLACK)

arcade.draw_xywh_rectangle_filled(5,55,200, 50,PALE_VIOLET_RED)
arcade.draw_xywh_rectangle_outline(5,55,200, 50,BLACK)
#finish drawing
arcade.finish_render()
arcade.run()
```

အပေါ် ထောင့်မှန်စတုဂံပုံကို ဒီနှစ်ခုနဲ့

```
arcade.draw_xywh_rectangle_filled(5,5,200, 50,BABY_BLUE)
arcade.draw_xywh_rectangle_outline(5,5,200, 50,BLACK)
```

ဆွဲထားတာပါ [[ပုံ (၇.၂)]]။ ပါရာမီတာတွေက $x, y, width, height, color$ အစဉ်အတိုင်းပဲ။ ဘယ်ဘက်ဘောင်ကနေ $x = 5$ ယူနစ် ခွာထားတယ်။ အပေါ်ဘောင်ကနေလည်း $y = 5$ ယူနစ်ခွာထားတယ်။ သုညထားပြီး စမ်းကြည့်ပါ။ အခြားတန်ဖိုးတွေ ထည့်ပြီး စမ်းကြည့်ပါ။ ပိုပြီးသဘောပေါက် လာပါလိမ့်မယ်။ ဒုတိယ ထောင့်မှန်စတုဂံက ဒီနှစ်ခုနဲ့



ပုံ ၃.၂

```
arcade.draw_xywh_rectangle_filled(5,55,200, 50,PALE_VIOLET_RED)
arcade.draw_xywh_rectangle_outline(5,55,200, 50,BLACK)
```

ဆွဲထားတာပါ။ ဘယ်ဘက်ကို ခွာထားတာက အပေါ်ပုံနဲ့ တူတူပဲ $(x = 5)$။ အပေါ်ပုံရဲ့ အောက်ခြေအနား နဲ့ ကပ်နေအောင် $y = 5 + 50(height) = 55$ ထားရပါမယ်။

သူ့နဂိုအတိုင်းဆိုရင် Arcade ကိုသြဒိနိတ်စနစ်မှာ origin က အောက်ခြေဘယ်ဘက်စွန်းလို့ ပြောခဲ့ပါတယ်။ `set_viewport` နဲ့ အပေါ်ဘယ်ဘက်စွန်းကို origin အဖြစ်ပြောင်းလဲ သတ်မှတ်ထားတယ်။ ဒီအတွက်ကြောင့် အထက်အောက် ပြောင်းပြန်ဖြစ်သွားပါတယ်။ ဖန်ရှင် documentation မှာကြည့်ရင် အခုလိုတွေ့ရမှာပါ (VS Code/PyCharm မှာ မောက်စ်ပွိုင်တာကို ဖန်ရှင်နံမည်ပေါ် မှာထားရင် documentation ပြပေးပါလိမ့်မယ်)

```
def draw_xywh_rectangle_filled(bottom_left_x: float,
                               bottom_left_y: float,
                               width: float,
                               height: float,
                               color: tuple[int, int, int]
                                   | list[int]
                                   | tuple[int, int, int, int]) -> None
```

Documentation မှာ bottom ဆိုရင် ကိုယ့်အတွက် top လို့ ပြောင်းပြန် စဉ်းစားရမယ်။ သူ့နဂိုစနစ်ကို ပြောင်းလိုက်လို့ ဒီအခက်အခဲ ကြုံရတာပါ။ ဒါကြောင့် Arcade ကိုပဲ အဓိကသုံးမယ်ဆိုရင် သူ့အရှိအတိုင်းသုံးတာ အကောင်းဆုံးပဲ။ အထက်အောက် ပြောင်းပြန် စဉ်းစားစရာ မလိုတော့ဘူး။

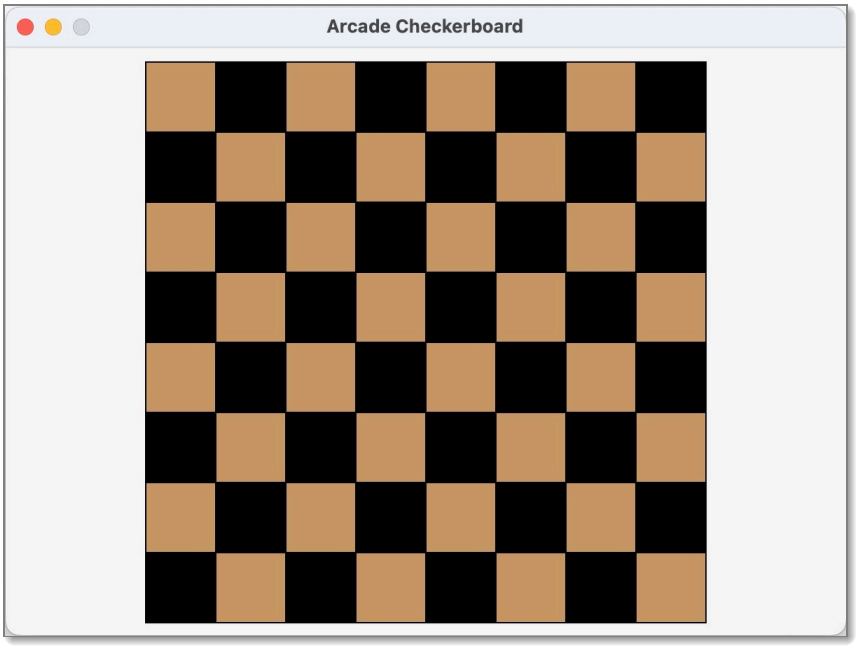
```
import arcade
from arcade.color import *

WIN_WIDTH = 600
BOARD_SIZE = 400
WIN_HEIGHT = 420
arcade.open_window(WIN_WIDTH, WIN_HEIGHT, "Arcade Checkerboard")
arcade.set_viewport(left=0,
                    right=WIN_WIDTH,
                    top=0,
```

```
                    bottom=WIN_HEIGHT)
arcade.set_background_color(WHITE_SMOKE)
arcade.start_render()

COLS = 8
ROWS = 8
SQ_SIZE = BOARD_SIZE / ROWS
X_LFT = (WIN_WIDTH - BOARD_SIZE) / 2
Y_TOP = (WIN_HEIGHT - BOARD_SIZE) / 2 + 1

for i in range(ROWS):
    for j in range(COLS):
        x = X_LFT + SQ_SIZE * i
        y = Y_TOP + SQ_SIZE * j
        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)
        arcade.draw_xywh_rectangle_outline(x,
                                           y,
                                           SQ_SIZE,
                                           SQ_SIZE,
                                           BLACK)

arcade.finish_render()
arcade.run()
```

၇.၄ `while` loop

`while` loop ဟာ *indefinite loop* ဖြစ်ပါတယ်။ Loop ကို စတင် လုပ်ဆောင်တဲ့အချိန်မှာ ဘယ်နှစ်ကြိမ် ပြန်ကျော့မလဲ အတိအကျ ကြို ‘မသိ’ ရင် indefinite loop လို့ သတ်မှတ်တယ်။ တစ်နဲ့ တစ်ဆယ်ကြား (အပါအဝင်) ကိန်းပြည့်တစ်လုံးကို ကျပန်း (random) ထုတ်ထားမယ်။ မမှန်မချင်း အဲဒီဂဏန်းကို ခန့်မှန်းပေးရမယ်။ ခန့်မှန်းတာ မှန်သွားရင် ဘယ်နှစ်ခါ ခန့်မှန်းရလဲနဲ့ မှန်တဲ့ဂဏန်းကို ပြပေးတဲ့ ပရိုဂရမ်မှာ `while` loop အသုံးပြု ရေးထားတာ တွေ့ရပါမယ်။

```
from random import *
```



ပုံ ၉.၃

```
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 အိပ်စ်ပရက်ရှင်) မမှားမချင်း loop က ပြန်ကျော့ပေးနေမှာပါ။ မှားတဲ့အခါ ထပ်မကျော့တော့ဘဲ ရပ်သွားမှာဖြစ်တယ်။ ကျပန်းဂဏန်းက ခုနှစ်ကျတယ် ဆိုပါစို့။ တစ်၊ ကိုး၊ သုံး၊ တစ်ဆယ်နဲ့ ခုနှစ် တို့ကို အစဉ်အတိုင်း ခန့်မှန်း ထည့်ပေးမယ်ဆိုရင် တစ်ကျော့ချင်း လိုက်ကြည့်တဲ့အခါ အခုလို တွေ့ရမှာပါ။

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

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

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

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

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

guess != num:                    # 7 != 7 က False ဖြစ်သွားပြီ
    # ထပ်မကျော့တော့ဘူး

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

Loop က ထပ်မကျော့တော့ဘဲ ရပ်သွားတာကို *loop exits* ဖြစ်သွားတယ်လို့ ပြောတယ်။ မြန်မာလိုတော့ loop ကနေ ထွက်သွားတယ်ပေါ့။ အကယ်၍ စောစောက ဥပမာမှာ ပထမဆုံးတစ်ခါမှာပဲ ခုနှစ် ထည့်လိုက်ရင်တော့ ဘလောက်ကို တစ်ခါမှ မလုပ်ဆောင်ဘဲ loop က ချက်ချင်းထွက်သွားမှာပါ။

Sentinel Controlled Loop

ဂဏန်းတွေ ဘယ်နှစ်လုံး ပေါင်းမှာလဲ ကြိုသိရင် `for` loop နဲ့ အလုပ်ဖြစ်တယ်။ ဂဏန်း ဘယ်နှစ်လုံးရှိလဲ ကြိုမသိထားဘဲ ရှိသလောက် တစ်လုံးချင်းရိုက်ထည့်ပြီး အားလုံးပေါင်းချင်တာဆိုရင်တော့ `for` loop နဲ့ အဆင်မပြေဘူး (မရနိုင်ဘူးလို့ မဆိုလိုပါ။ မရမက လုပ်တဲ့နည်းတွေလည်း ရှိပါတယ်)။ ဒီလိုအခြေအနေမျိုးမှာ *sentinel controlled loop* ကို သုံးပါတယ်။

Sentinel controlled loop မှာ loop ကနေ ထွက်ဖို့အတွက် အသုံးပြုတဲ့ သီးသန့်တန်ဖိုးတစ်ခု ရှိရပါမယ်။ ဒီတန်ဖိုးကို *sentinel value* လို့ ခေါ်တယ်။ ဂဏန်းတွေပေါင်းတဲ့ ကိစ္စအတွက် *sentinel value* ရွေးချယ်သတ်မှတ်တဲ့အခါ ပေါင်းရမဲ့ဂဏန်း မဖြစ်နိုင်တဲ့ တစ်ခုကို ရွေးရပါမယ်။ သုညဟာ အပေါင်းထပ်တူရ ဂုဏ်သတ္တိရှိတဲ့အတွက် ၎င်းကို *sentinel value* အဖြစ် ထားနိုင်တယ်။

```
# 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 ဂဏန်းဟာ သုညမဟုတ်မချင်း `total` မှာ ပေါင်းထည့်ပြီး နောက်ဂဏန်းတစ်ခု ထည့်ပေးဖို့ input တောင်းမှာပါ။ Sentinel တန်ဖိုး သုည ထည့်လိုက်ရင် loop က ထွက်သွားမယ်။ ပြီးတဲ့အခါ

total ကို ပြပေးမှာပါ။ စစချင်းပဲ သုည ထည့်လိုက်ရင် loop က တစ်ခါမှ မကျော့တဲ့ ထွက်သွားမယ်။ total လည်း သုညပဲ ထွက်ပါမယ်။ အခုပရိုဂရမ်ကို 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 ပြန်ကျော့တာ ဘယ်တော့မှ ရပ်မှာ မဟုတ်ဘူး။ ဒီလိုဖြစ်နေတာကို ဖြတ်ချပစ်ဖို့အတွက် `break` စတိတ်မန့်ကို သုံးထားပါတယ်။ `if val == SENTINEL:` (ထည့်ပေးတဲ့ တန်ဖိုးက သုညဆိုရင်) လက်ရှိ အလုပ်လုပ်နေတဲ့ loop ကို `break` လိုက်မှာဖြစ်တယ်။

အထက်ပါ sentinel loop ပုံစံနှစ်ခုကို ယှဉ်ကြည့်ရင် ပထမတစ်ခုမှာ input တန်ဖိုးထည့်ခိုင်းတာကို loop မစမီ တစ်ခါ ကြိုလုပ်ထားရတယ်။ နောက်တစ်ခုမှာတော့ အဲ့ဒီလို ကြိုလုပ်ထားစရာမလိုဘူး။ Loop ဘလောက်ထဲက စတိတ်မန့်တစ်ခုကို အပြင်မှာထုတ်ထားရတဲ့အတွက် တချို့က ပထမပုံစံထက် ဒုတိယပုံစံက ကုဒ်စတိုင်လ်အားဖြင့် ပိုပြီး သပ်ရပ်ကြော့ရှင်းတယ်လို့ ယူဆကြပါတယ်။

Result Controlled Loop

တစ်နှစ်ငါးရာခိုင်နှုန်း အတိုးနဲ့ ဘဏ်မှာ ဒေါ်လာတစ်ထောင် စုထားတယ်။ တစ်နှစ်ပြည့်တိုင်း အတိုးအရင်းပေါင်းပြီး နောက်နှစ်အတွက် အတိုးတွက်တယ်။ နှစ်ထပ်တိုး (yearly compound interest) လို့ခေါ်ပါတယ်။ ဒီနည်းလမ်းနဲ့ အနည်းဆုံး ဒေါ်လာတစ်သန်း စုမိဖို့ (မီလျံနာတစ်ယောက်ဖြစ်ဖို့) ဘယ်နှစ်နှစ်စောင့်ရမလဲ။ သုံးနှစ်ပြည့်ရင် စုမိမဲ့ 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	$1102.5 \times 0.05 = 55.125$	$1102.5 + 55.125 = 1157.625$

တေဘဲလ် ၇.၂ သုံးနှစ်စာ နှစ်ထပ်တိုး

ဒေါ်လာတစ်သန်း အနည်းဆုံးရဖို့ 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 ပြဿနာမရှိပါ။ တစ်နှစ်စာ တွက်ထားတာ စစ်ကြည့်လို့ရအောင် ထုတ်ကြည့်တာပါ)။ တစ်သန်းမပြည့်မချင်း ပြန်ကျော့ပေးအောင် loop ကွန်ဒီရှင်ကို `balance < TARGET` နဲ့ စစ်ထားတယ်။ တစ်သန်းနဲ့ညီရင် (သို့) တစ်သန်းကျော်သွားတာနဲ့ ကွန်ဒီရှင် `False` ဖြစ်သွားပြီး ထပ်မကျော့တော့ဘူး။ Loop တစ်ခါကျော့တိုင်း တစ်နှစ်စာ အတိုးအရင်းပေါင်း `balance` ကို တွက်ချက်ပြီး loop ကနေ ဘယ်အချိန် ထွက်မလဲကလည်း အဲဒီရလဒ်အပေါ် မူတည်နေတာ တွေ့ရပါမယ်။ ဒီလို သဘောတရားရှိတဲ့ loop မျိုးကို result controlled loop လို့ ခေါ်ပါတယ်။ ကျော့မဲ့ အကြိမ်အရေအတွက်က loop ထဲမှာ တွက်ချက်ထားတဲ့ ရလဒ်ပေါ် မူတည်တယ်။

ပရိုဂရမ် output ကို အခုလို 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 အကျယ်တဝင့် မဖော်ပြတော့ဘဲ အခုလို ဖော့မတ်ရအောင် ဘယ်လိုလုပ်ထားလဲကိုပဲ ရှင်းပြပါမယ်။

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

ကော်လံ ခေါင်းစည်း အတွက် f-string ပါ။ ဖော့မတ်လုပ်ရမဲ့ တန်ဖိုး/ဗေရီရေဘဲလ်/အိပ်စ်ပရက်ရှင်က `:` (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။ ကျန်တဲ့ 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 ကိန်းပြည်ဂဏန်းအနေနဲ့ ပြပေးအောင် `d` သုံးတယ်။ ကျန်တဲ့နှစ်ခုကို ဒဿမနဲ့ ပြဖို့ `f` သုံးတယ်။ `.2` ကတော့ ဒဿမနောက် ဂဏန်းနှစ်လုံး ပြပေးခိုင်းတာ။ ကိန်းဂဏန်းဆိုရင် သူ့နဂိုအတိုင်း ညာဘက်ကပ်ပေးတဲ့အတွက် `>` ထည့်ဖို့ မလိုဘူး။ ဘယ်ဘက်ကပ်ချင်ရင် `<` သုံးလို့ရတယ်။

၇.၅ လေ့ကျင့်ရန် ဥပမာများ

ကွန်ထရိုးလ် စထရက်ချာတွေ အသုံးချတတ်လာအောင် များများလေ့ကျင့်၊ များများစဉ်းစား၊ များများရေးကြည့် ရမှာပါ။ ဘယ်အတတ်ပညာမဆို များများလေ့ကျင့်မှ ကျွမ်းကျင်လာနိုင်မှာပါ။ ဒီသဘောအရ ပရိုဂရမ်းမင်း ပညာရပ်ဟာလည်း ချင်းချက်မဖြစ်နိုင်ပါဘူး။

> အောက်ပါ ဥပမာတစ်ခုချင်းကို ပုစ္ဆာနားလည်အောင်ဖတ်ပြီး ကိုယ်တိုင်စဉ်းစား ရေးကြည့်ဖို့ လေးလေးနက်နက် အကြံပြုပါတယ်။ ရေးပြီးသွားရင် ပုစ္ဆာမှာ ဖော်ပြထားချက်နဲ့ အညီ အလုပ်လုပ်/မလုပ် ဟာကွက်မရှိအောင် ဘယ်လိုစစ်ဆေး စိစစ်မလဲ မိမိဘာသာ စဉ်းစားကြည့်ပါ။

Internet Delicatessen

Online ကနေ အစားအသောက်တွေ delivery ပို့ပေးဖို့ အမှာလက်ခံတဲ့ ဆိုင်လေးတစ်ဆိုင်အတွက် ပရိုဂရမ်ရေးပေးရပါမယ်။ အော်ဒါမှာတဲ့အခါ မှာမဲ့ အစားအသောက် နံမည်၊ ဈေးနှုန်းနဲ့ အိပ်စ်ပရက်စ် delivery ယူမယူ ပရိုဂရမ်မှာ ထည့်ပေးရပါမယ်။ ပရိုဂရမ်က အော်ဒါနဲ့ စုစုပေါင်းကျသင့်ငွေကို ထုတ်ပေးရပါမယ်။ တစ်သောင်းနဲ့အထက်မှာရင် delivery ဖိုး ပေးစရာမလိုပါ။ တစ်သောင်းအောက်ဆိုရင်တော့ နှစ်ရာပေးရပါမယ်။ အိပ်စ်ပရက်စ် delivery ယူမယ်ဆိုရင် သုံးရာအပို 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_exp_deli = int(input('Express delivery (0==no, 1==yes): '))

tot_deli_fee = 0
if price >= 10_000 and is_exp_deli == 1:
    tot_deli_fee = 300
elif price >= 10_000 and is_exp_deli == 0:
    tot_deli_fee = 0
elif price < 10_000 and is_exp_deli == 1:
    tot_deli_fee = 200 + 300
elif price < 10_000 and is_exp_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}")
```

အိပ်စ်ပရက်စ် delivery ယူ/မယူ တစ် (သို့) သုညထည့်ပေးရမှာပါ။ တစ်/သုညမဟုတ်တဲ့ ဂဏန်းတစ်ခု မှားထည့်မိရင် `if...elif` ကွန်ဒီရှင်တွေ တစ်ခုမှ `True` ဖြစ်မှာ မဟုတ်ပါဘူး။ မှားထည့်ထားတယ်လို့ သတိပေးဖို့ `else` အပိုင်းမှာ လုပ်ထားတယ်။ မှန်/မမှန် စိစစ်တဲ့အခါ အောက်ပါဇယားမှ ဖြစ်နိုင်ခြေ အားလုံး ခြုံငုံမိအောင် စစ်သင့်တယ်။ အိပ်စ်ပရက်စ် delivery အတွက် input ဂဏန်း မှားထည့်ရင် သတိပေးတာကိုလည်း စစ်သင့်တယ်။ တစ်သောင်းဖိုး ဝယ်တာကို သုံးမျိုးစစ်ထားတာ လေ့လာကြည့်ပါ။ တစ်သောင်း မပြည့်တာနဲ့ တစ်သောင်းကျော်ဖိုး အတွက်လည်း အလားတူ စစ်ကြည့်လို့ အားလုံးမှန်တယ်ဆိုရင် ဒီပရိုဂရမ်မှာ bug ပါနိုင်ခြေ လုံးဝမရှိသလောက် ဖြစ်သွားပါပြီ။

10,000 MMK and above?	Express Delivery?
Yes	Yes
Yes	No
No	Yes
No	No

တေဘဲလ် ၇.၃ 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 သုံးရာ့ငါးဆယ် ပြောင်းလဲသတ်မှတ်လိုက်တဲ့အပြင် အိပ်စ်ပရက်စ် delivery ကလည်း တစ်ရာဈေးထပ်တက်သွားတယ် ဆိုပါစို့။ တကယ့်လက်တွေ့မှာလည်း ဒါမျိုးဖြစ်လေ့ရှိပါတယ်။ ဒီအတွက်ကို ပရိုဂရမ်ကို ပြင်ပေးရပါမယ်။ ဆိုင်ပိုင်ရှင်ကလည်း ဈေးနှုန်းမကြာခဏ ပြောင်းဖို့လိုနိုင်ကြောင်း ပြောလာပါတယ်။ နောင်လည်း ထပ်ပြင်ပေးဖို့လိုမဲ့ သဘောပါ။ လွယ်လွယ်ကူကူ ပြင်ပေးနိုင်ရင် အကောင်းဆုံးပါ။ ပြင်ဆင်တဲ့အခါ မှားနိုင်ခြေနည်းဖို့လည်း အရေးကြီးပါတယ်။ ခုရေးထားတဲ့အတိုင်းဆိုရင် ပြီးခဲ့တဲ့ပရိုဂရမ်မှာ ပြဿနာရှိနေပါတယ်။

ပြီးခဲ့တဲ့ ပရိုဂရမ်မှာ ပြင်မယ်ဆိုရင် 10_000 ကို လေးနေရာ၊ ပုံမှန်ပို့ခ 200 နဲ့ အိပ်စ်ပရက်စ်အပိုကြေး 300 စတာတွေကို နှစ်နေရာစီ လိုက်ပြင်ရပါမယ်။ အလုပ်ရှုပ်တဲ့အပြင် ပြင်ဖို့ကျန်ခဲ့တာလို့ မှားတာလည်းဖြစ်နိုင်တယ်။ “Find and Replace” လုပ်မှာပေါ့လို့ စောဒကတက်စရာ ရှိပါတယ်။ အခုလို ကုဒ်လိုင်းနည်းနည်းပဲရှိတဲ့ ပရိုဂရမ်အသေးလေးမှာ အဆင်ပြေနိုင်ပေမဲ့ ပိုရှုပ်ထွေးပြီး ကုဒ်လိုင်းတွေများတဲ့ ပရိုဂရမ်မျိုးတွေမှာ “Find and Replace” လုပ်ရင် မပြင်သင့်တာတွေကိုပါ မရည်ရွယ်ပဲ ပြင်မိသွားတာဖြစ်တတ်ပါတယ်။ ပရိုဂရမ်ရေးတဲ့အခါ ကျင့်သုံးရမဲ့ အလေ့အထကောင်းတစ်ခုက ကုဒ်ထဲမှာ ဒီတိုင်းချရေးထားတဲ့ တန်ဖိုးတွေ (literal constants) တွေကို နာမည်ပေးထားတာပါ။ အပေါ်က ပရိုဂရမ်မှာ literal constants တွေချည်း သုံးထားတယ်။ နံမည်ပေးထားတဲ့ constants (named constants) တွေအဖြစ် ပြောင်းရေးသင့်တယ်။

```python
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 %8.2f' % (itm_name, price))
```

```
print(f'%-10s %8.2f' % ('delivery', tot_deli_fee))
print(f'%-10s %8.2f' % ('total', tot_cost))
```

ဒီပရိုဂရမ်ကို cascading if မသုံးဘဲ ရိုးရိုး 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 ခ ပေးဖို့ လိုတယ်ဆိုရင် tot_deli_fee မှာ DELI_FEE ပေါင်းထည့်ပေးတယ်။ ဒုတိယ if က အိပ်စ်ပရက်စ် delivery ယူမယ်ဆိုရင် tot_deli_fee မှာ EXP_DELI_FEE ထပ်ပေါင်းထည့်တယ်။ အောက်ဆုံး if ကတော့ အိပ်စ်ပရက်စ် delivery အတွက် တစ်နဲ့ သုည မဟုတ်တာ ထည့်မိရင် သတိပေးစာသား ပြပေးတယ်။

ဒီပရိုဂရမ်နဲ့ ဒီ့မတိုင်ခင် သူ့ရှေ့က ပရိုဂရမ်က ဖြစ်နိုင်ခြေအားလုံးအတွက်တော့ ရလဒ် တူတူမထွက်ပါဘူး။ အခြေအနေ တစ်ခုကလွဲလို့ ကျန်တာတွေအတွက်တော့ ရလဒ်တူပါတယ်။ တစ်သောင်းငါးထောင် ထက်ငယ်တဲ့ တန်ဖိုးနဲ့ အိပ်စ်ပရက်စ် delivery အတွက် ဂဏန်း လွဲထည့်ကြည့်ပါ။ 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      12350.00
```

နောက်ဆုံး ပရိုဂရမ်က အိပ်စ်ပရက်စ် delivery အတွက် မပေါင်းထည့်ပေမဲ့ ပုံမှန် delivery ခ သုံးရာ့ငါးဆယ်ကိုတော့ ထည့်ပေါင်းသွားတာ တွေ့ရမယ်။ မှားထည့်တာကိုပဲ သတိပေးစာသားပြပေးတာ၊ ထည့်မတွက်သွားတာက ပိုပြီး သဘာဝကျတယ်လို့ ယူဆရမှာပါ။

> ပြင်ပကနေ ထည့်ပေးတဲ့အခါ မဖြစ်သင့်တဲ့ input တန်ဖိုးတွေ ဝင်မလာအောင် စိစစ်တာကို input validation လို့ ခေါ်တယ်။ တကယ့် လက်တွေ့ အသုံးချ ပရိုဂရမ်တွေမှာ input validation လုပ်ထားဖို့ အရေးကြီးပေမဲ့ ဘီဂင်နာအဆင့် လေ့လာတဲ့ ဥပမာတွေမှာတော့ လေ့လာရင်းကိစ္စကနေ လမ်းကြောင်းမချော်သွားအောင် ဆင်ခြင်ရမှာဖြစ်ပြီး သင့်တော်ရုံ ဆက်စပ်ရှင်းပြပါမယ်။

တာရာ ပရက်ရှာ

ကားဘီးလေထိုးတာဟာ ကားရဲ့ စွမ်းဆောင်ရည်ရော အန္တရာယ်ကင်းဖို့အတွက်ပါ ပဓာနကျပါတယ်။ ကားတစ်စီးအတွက် အကြံပြုထားတဲ့ တာယာပရက်ရှာ (recommended pressure) အပေါ် မူတည်ပြီး လေဘယ်လောက်တင်းလို့ရလဲ၊ လျော့လို့ရလဲ ရှိပါတယ်။ ဥပမာ recommended pressure က 35 psi (pounds per square inch) ဖြစ်ရင် အလွန်ဆုံး 31.5 psi ထိ လေလျော့လို့ရပါတယ်။ လေပိုတင်းမယ်ဆိုရင်လည်း အလွန်အလွန်ဆုံး 44 psi အထိ ရပါနိုင်ပါတယ်။

ရှေ့တာယာနှစ်လုံး ပရက်ရှာအနီးစပ်ဆုံးတူသင့်ပြီး 3 psi အထိ ကွာဟလို့ရတယ်။ ကွာဟချက်က 3 psi ထက်တော့ မများသင့်ဘူး။ နောက်တာယာနှစ်လုံးလည်း ထိုနည်းတူစွာပဲ ဖြစ်တယ်။ ကားမော်ဒယ် အလိုက် recommended pressure ကွာခြားပေမဲ့ အိမ်စီးကားအများစုအတွက် 35 psi ဖြစ်တယ်လို့ ယူဆပြီး တာယာပရက်ရှာ အိုကေမကေ 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၊ ရှေ့တာယာနှစ်လုံး 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` ဘူလီယန် သုံးထားတာ နည်းနည်း ရှင်းပြဖို့ လိုပါမယ်။ စစချင်း `False` တန်ဖိုး ထည့်ထားတာ တွေ့ရမှာပါ။ `if` စတိတ်မန့်တွေက တာယာတစ်ခုချင်းကို out of range ဖြစ်နေလား စစ်ထားတာတွေ့ရတယ်။ တာယာတစ်ခု out of range ဖြစ်တာနဲ့ `is_out_of_range` က `True` ဖြစ်သွားမှာပါ။

အောက်ပိုင်းမှာ `front_diff` နဲ့ `rear_diff` ရှာတဲ့အခါ `abs` ဖန်ရှင်နဲ့ ပကတိတန်ဖိုး ယူထားတာ သတိပြုပါ။ ပရက်ရှာ ခြားနားချက်ရှာတဲ့အခါ အနှုတ်တန်ဖိုး ထွက်နိုင်တဲ့အတွက်ကြောင့်ပါ။ ဘယ်ဘက် တာယာက ပရက်ရှာနည်းနေတဲ့အခါ (ဥပမာ $32 - 37 = -5$) အနှုတ်တန်ဖိုး ဖြစ်နေမယ်။ ဒါကြောင့် ပကတိတန်ဖိုးယူမှပဲ ကွာဟချက် 3 psi ကျော်မကျော်စစ်ပေးတဲ့အခါ အဖြေမှန်ရပါမယ်။

```
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 ပရိုဂရမ် အဆောက်အဦးတစ်ခုရဲ့ အခြေခံ အုပ်ချုပ်တွေပါ။ ဒီအခြေခံ အုပ်ချုပ်တွေကိုမှ ကောင်းကောင်းမွန်မွန် နားလည် အသုံးမချတတ်ရင် ကျွမ်းကျင်တဲ့ ပရောဖက်ရှင်နယ် ပရိုဂရမ်မာ တစ်ယောက် မဖြစ်နိုင်ဘူးဆိုတာ အထူးပြောစရာ လိုမယ် မထင်ပါဘူး။ ကားရဲလ်မှာလည်း ဖန်ရှင် အခြေခံ သဘောတရားနဲ့ အသုံးချပုံကို လေ့လာခဲ့ကြရပါတယ်။ ဒီအခန်းမှာတော့ တန်ဖိုးပြန်ပေးခြင်း၊ ပါရာမီတာတွေ အသုံးပြုပုံနဲ့ *exception-handling* တို့ကို ဆက်လက် လေ့လာကြမှာပါ။ လက်တွေ့နဲ့ နီးစပ်ပြီး အမှန်တကယ် အသုံးဝင်မဲ့ ပရိုဂရမ်လေးတွေကို အစကနေ အဆုံးထိ တစ်ဆင့်ပြီးတစ်ဆင့် ဘယ်လိုဒီဇိုင်းပြုလုပ်တည်ဆောက်လဲ ကိုလည်း ဖော်ပြပေးသွားမှာပါ။

၈.၁ တန်ဖိုးပြန်ပေးတဲ့ ဖန်ရှင်များ

အခန်း (၅) မှာ ဖော်ပြခဲ့တဲ့ နှစ်ထပ်ကိန်းရှာတဲ့ square ဖန်ရှင်ကိုပဲ အသေးစိတ် တစ်ခါထပ်ကြည့်ရအောင်။ ဒီလောက် ရှင်းရှင်းလေးကို အကျယ်ချဲ့နေတယ်လို့ ထင်ကောင်း ထင်ပါလိမ့်မယ်။ နည်းနည်းတော့ စိတ်ရှည်သည်းခံ ပေးရပါမယ်။ အခြေခံကျတဲ့ သဘောတရားတွေ ကျေညက်ထားမှ ရှေ့ဆက်တဲ့ အခါ လွယ်ကူမှာ မို့လို့ပါ။

```
>>> def square(x):
...     return x ** 2
...
>>>
```

ဝိုက်ကွင်းထဲက ဗေရီရေဘဲလ် x က ဖန်ရှင် ပါရာမီတာ (*parameter*) ဖြစ်ပြီး ဖန်ရှင်ခေါ်တဲ့အခါ ထည့်ပေးမဲ့ အာဂုမန့် (*argument*) တန်ဖိုးကို ကိုယ်စားပြုတယ်။ return စတိတ်မန့်က ဖန်ရှင်ခေါ်တဲ့နေရာကို တန်ဖိုးပြန်ပေးတဲ့ စတိတ်မန့်ပါ။

ဖန်ရှင်အသုံးပြုတာကို *function call* လုပ်တယ်လို့ သိထားပြီးပါပြီ။ မြန်မာလိုတော့ 'ဖန်ရှင်ခေါ်တယ်' သို့မဟုတ် 'ဖန်ရှင်ကောလ်တယ်' လို့ အပြောများတယ်။ ဖန်ရှင်ခေါ်တဲ့ ပုံစံက ဒီလိုပါ

```
>>> square(2.5)
6.25
```

အခု ဖန်ရှင်ကောလ် အတွက် ပါရာမီတာ x ရဲ့ တန်ဖိုးက 2.5 ဖြစ်မှာပါ။ (ဖန်ရှင်ခေါ်တဲ့အခါ ပါရာမီတာ ဗေရီရေဘဲလ် x ကို အာဂုမန့်နဲ့ အဆိုင်းမန့်လုပ်ပေးတယ်လို့ ယူဆနိုင်တယ်။ ဒီကိစ္စအတွက် အာဂုမန့်က

2.5 ဖြစ်တယ်)။ အားလုံးသိပြီး ဖြစ်တဲ့အတိုင်း ဖန်ရှင်ခေါ်ရင် ဖန်ရှင်ဘလောက်ကို လုပ်ဆောင်ပေးမှာပါ။ ဖန်ရှင်ဘလောက်ထဲက return စတိတ်မန့် လုပ်ဆောင်တဲ့အခါ အိပ်စ်ပရက်ရှင် x ** 2 ကို တန်ဖိုးအရင်ရှာတယ်။ 6.25 ရတယ်။ ဒီတန်ဖိုးကို ဖန်ရှင်ခေါ်ထားတဲ့ နေရာကို return က ပြန်ပို့ပေးလိုက်တာပါ။ အောက်ပါ ဖန်ရှင်ကောလ်မှာလည်း ဒီဖြစ်စဉ် သဘောအတိုင်း တစ်ခါထပ်ဖြစ်မှာ ဖြစ်တယ်။

```
>>> a = 1024
>>> result = square(a)
>>> result
1048576
```

အခုတစ်ခါ ပါရာမီတာ x ဟာ အာ့ဂုမန့် a ရဲ့ တန်ဖိုး ဖြစ်တယ် (x = a အဆိုင်းမန့် လုပ်တဲ့သဘောပဲ)။ x ** 2 က ရလာတဲ့ 1048576 ကို ဖန်ရှင်ခေါ်တဲ့ နေရာက ပြန်ရတယ်။ နောက်ဆုံးတော့ ဒီတန်ဖိုးကို result မှာ အဆိုင်းမန့်လုပ်တယ်။ ဖြစ်စဉ်အရ ရိုးရှင်းပါတယ်။

```
>>> x = 10
>>> square(x)
```

ဒီလိုဆိုရင်ရော ဘယ်လို ဖြစ်မလဲ။ နည်းနည်းထူးခြားတာက အာ့ဂုမန့်နဲ့ ပါရာမီတာ နံမည်တူနေတာ။ ပါရာမီတာရဲ့ စကုပ်ဟာ ဖန်ရှင်သတ်မှတ်ချက် အတွင်းမှာပဲ ရှိတယ်လို ယူဆရမှာပါ။ ဒါကြောင့် အာ့ဂုမန့် x နဲ့ ပါရာမီတာ x နဲ့က သီးခြား ဗေရီရေဘဲလ်တွေ။

```
>>> u = 15
>>> t = 5
>>> square(u + 2*t)
```

အာ့ဂုမန့်က အိပ်စ်ပရက်ရှင် ဖြစ်နေရင် တန်ဖိုးအရင်ရှာပြီး ရလဒ်ကို ပါရာမီတာနဲ့ အဆိုင်းမန့် 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 အဆိုင်းမန့် မလုပ်တော့ဘဲ တစ်ခါတည်း အာ့ဂုမန့် အနေနဲ့ ထည့်လိုက်တာပါ။ သဘောတရား တူတူပါပဲ။

ဖန်ရှင် return လုပ်တဲ့ သဘောကို နားလည်ထားဖို့လည်း အရေးကြီးတယ်။ return စတိတ်မန့်ဟာ ဖန်ရှင်ကနေ တန်ဖိုးတစ်ခုကို ဖန်ရှင်ခေါ်တဲ့ဆီကို ပြန်ပေးတယ်လို့ သိထားပြီးပါပြီ။ ဖန်ရှင်ထဲကနေ return လုပ်လိုက်တာနဲ့ ခေါ်ထားတဲ့နေရာကို 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'
```

အခုအိပ်စ်ပရက်ရှင်ရဲ့ တန်ဖိုးရှာဖို့ get_sign(10) ခေါ်လိုက်တဲ့အခါ လက်ရှိနေရာကနေ လုပ်ဆောင်မှုက ဖန်ရှင်ဘလောက်ဆီ ပြောင်းရွှေ့ ရောက်ရှိသွားပါမယ်။ ဖန်ရှင်ထဲက စတိတ်မန့်တွေ အစဉ်အတိုင်း စတင် လုပ်ဆောင်တယ်။ ဖန်ရှင်က return လုပ်တဲ့အခါ လုပ်ဆောင်မှုက ဖန်ရှင်ဘလောက်ထဲကနေ ခေါ်ခဲ့တဲ့ နေရာကို တဖန်ပြန်၍ ပြောင်းရွှေ့သွားတယ်။ return ပြန်လိုက်တဲ့ တန်ဖိုးကို ဖန်ရှင်ခေါ်တဲ့နေရာမှာ ရရှိပြီး လုပ်လက်စ အိပ်စ်ပရက်ရှင်ကို 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)
```

မြှားအနက်က ဖန်ရှင်ခေါ်လိုက်တဲ့အခါ လုပ်ဆောင်မှု ပြောင်းရွှေ့သွားတာကို ပြတယ်။ မြှားအနီက return ပြန်တဲ့အခါ ခေါ်ခဲ့တဲ့နေရာ ပြန်ရောက်သွားတာကို ပြတာပါ။

ဆက်လက်ပြီး ပါရာမီတာ တစ်ခုထက်ပိုတဲ့ ဖန်ရှင်တချို့ကို ကြည့်ပါမယ်။ ပါရာမီတာဆိုတာ ဖန်ရှင် အတွက် လိုအပ်တဲ့ input ကို လက်ခံတဲ့ ဗေရီရေဘဲလ်ပါပဲ။ ထောင့်မှန်စတုဂံရဲ့ အလျားနဲ့ အနံကနေ ဧရိယာရှာပေးတဲ့ ဖန်ရှင်က ဒီလိုပါ

```
def rect_area(wid, len):
    return wid * len
```

ဖန်ရှင်တစ်ခုကို အခြေခံ အုပ်ချုပ်သဖွယ် 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 အိပ်စ်ပရက်ရှင်ကို တန်ဖိုးရှာပါတယ်

```
rect_area(w, l) * h
```

rect_area ဖန်ရှင်ခေါ်တယ်။ wid=w, len=l ဖြစ်မယ်။ အခုကိစ္စအတွက် ပါရာမီတာနှစ်ခုရဲ့ တန်ဖိုးက 10 နဲ့ 5 အသီးသီး ဖြစ်မှာပါ။ 50 ရပါမယ်။ 50 * h ကို တန်ဖိုးဆက်ရှာပြီး ရလာတဲ့ 150 ကို

box_vol ခေါ်ထားတဲ့နေရာကို return ပြန်ပေးမှာ ဖြစ်တယ်။ အခြေခံသဘောတရားတွေ သိပြီးတဲ့အခါ အတန်အသင့်ရှုပ်ထွေးတဲ့ ဖန်ရှင်တချို့ကို ကြည့်ပါမယ်။

ဖန်ရှင်များနှင့် အက်ဘ်စရက်ရှင်းလုပ်ခြင်း

မွေးသက္ကရာဇ် (date of birth) ကနေ အသက် တွက်ပေးတဲ့ ဖန်ရှင်ကို လေ့လာကြည့်ပါ။ အသက်တွက်တဲ့ လော့ဂျစ်ကို 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 လော့ဂျစ်တွေ ဘယ်လိုပဲ ရှုပ်ထွေးပါစေ၊ အသုံးပြုရတာကတော့ မခက်ပါဘူး။ ဖန်ရှင် ခေါ်တဲ့အခါ ဘယ်လို တည်ဆောက်ထားလဲ အတွင်းပိုင်း အယ်လ်ဂိုရစ်သမ်တွေ၊ လော့ဂျစ်တွေ သိစရာမလိုဘဲ သုံးရတာပါ။ ဖန်ရှင်က ၎င်းရဲ့ အတွင်းပိုင်း ကုဒ်တွေကို အက်ဘ်စရက်ရှင်း (*abstraction*) လုပ်ပေးလိုက်တာ ဖြစ်တယ်။ ဒါဟာ ဖန်ရှင်ရဲ့ အရေးပါဆုံး ဂုဏ်သတ္တိလို့ ဆိုရင်လည်း မမှားဘူး။

age_today ဖန်ရှင်ဟာ ပိုကြီးတဲ့ ပရိုဂရမ်တစ်ခုရဲ့ တစ်စိတ်တစ်ပိုင်း ဖြစ်လာနိုင်ပါတယ်။ ပရိုဂရမ် အသေးစားလေးတစ်ခုမှာ အသုံးပြုထားတာကို လေ့လာကြည့်ပါ။ နိုင်ငံအများစုမှာ (၁၈) နှစ် မပြည့်သေးတဲ့သူကို ဆေးလိပ်ရောင်းခွင့် မရှိဘူး။ ဥပဒေရှိပါတယ်။ စားသုံးသူရဲ့ မွေးသက္ကရာဇ် ထည့်ပေးလိုက်တာနဲ့ ရောင်းလို့ ရ/မရ ပြပေးတဲ့ ပရိုဂရမ်လေးပါ။

```
# 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 အခြားပရိုဂရမ်တွေမှာလည်း အသုံးကျနိုင်ပါတယ်။ ဒီဖန်ရှင်ကို ခဏခဏ မရေးရဘဲ လိုအပ်တဲ့ ပရိုဂရမ်တွေကနေ အလွယ်တကူ ပြန်သုံးလို့ရအောင် လိုက်ဘရီ၊ ဒါမှမဟုတ် မော်ဒျူး လို့ခေါ်တဲ့ ကုဒ်အဖွဲ့အစည်း ယူနစ်တစ်ခုအနေနဲ့ ထုတ်ထားနိုင်တယ်။ လိုက်ဘရီ (သို့) မော်ဒျူးတစ်ခု ဟာ ဆက်စပ်တဲ့ ဖန်ရှင်တွေ၊ ကလပ်စ်တွေနဲ့ ဖွဲ့စည်းထားတာ ဖြစ်တယ်။ နောက်ပိုင်းမှာ ဒါနဲ့ပါတ်သက်ပြီး သီးခြားလေ့လာရမှာပါ။

၈.၂ တန်ဖိုးပြန်မပေးတဲ့ ဖန်ရှင်များ

ဖန်ရှင်အားလုံးတော့ တန်ဖိုးပြန်ပေးတဲ့ 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။ ကားရဲလ်ဖန်ရှင်တွေမှာလည်း `return` မသုံးခဲ့တာ ပြန်အမှတ်ရမှာပါ။ `append_n_times` ကို လေ့လာကြည့်ပါ

```
def append_n_times(lst, itm, n):
    for i in range(n):
        lst.append(itm)

lst = []
append_n_times(lst, 'hello', 10)
print(lst)
```

အိုက်တမ်တစ်ခုကို သတ်မှတ်ထားတဲ့ အရေအတွက်ပြည့်အောင် list တစ်ခုနောက်ကနေ ဆက်ပေးတယ်။ နဂို list မှာ အိုက်တမ်တွေ တိုးသွားပြီး စတိတ်အပြောင်းအလဲ ဖြစ်စေတယ်။

Output ထုတ်တဲ့ ဖန်ရှင်တွေဟာ တန်ဖိုးပြန်ပေးလေ့မရှိဘူး။ စခရင်မှာ စာသား (သို့) ရုပ်ပုံ ပြပေးတာဟာ output ဖြစ်တယ်။ ဖိုင်တစ်ခုမှာ ရေးတာလည်း output ပဲ (ဥပမာ Python ကုဒ်ဖိုင်ကို ပြင်ပြီး save လုပ်တာ) ။ အော့ဘ်ဂျက် စတိတ်ကို ပြောင်းလဲစေတဲ့ ဖန်ရှင်တွေဟာလည်း တန်ဖိုးပြန်မပေးတဲ့ ဖန်ရှင်တွေ ဖြစ်လေ့ရှိတယ် (ဥပမာ list ရဲ့ append နဲ့ insert ဖန်ရှင်)။ စတိတ်အပြောင်းအလဲ ဖြစ်စေတဲ့ ဖန်ရှင်အားလုံး တန်ဖိုးပြန်မပေးတာတော့ မဟုတ်ဘူး။ ဥပမာ pop ဟာ တန်ဖိုးပြန်ပေးပါတယ်။ စတိတ်အပြောင်းအလဲလည်း ဖြစ်စေတယ်။

တန်ဖိုးပြန်တဲ့ ဖန်ရှင်ပဲ return ပြန်လို့ရတာ မဟုတ်ပါဘူး။ တန်ဖိုးပြန်မပေးတဲ့ ဖန်ရှင်တွေမှာလည်း return ပါနိုင်ပါတယ်။ print_sign ကို ဒီလိုရေးလို့လည်း 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/အိပ်စ်ပရက်ရှင် တွဲပြီး ပါလို့မရပါဘူး။ ဖန်ရှင်ဘလောက် ပြီးတဲ့အခါ ခေါ်တဲ့နေရာကို ပြန်ရောက်သွားရမှာ ဖြစ်တဲ့အတွက် return မပါတဲ့ ဖန်ရှင်တွေရဲ့ 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)။

> ပိုပြီးတိတိကျကျ ပြောမယ်ဆိုရင် Python မှာ တန်ဖိုးပြန်မပေးတဲ့ ဖန်ရှင်တွေက `None` အော့ဘ်ဂျက် ပြန်ပေးပါတယ်။ `None` အော့ဘ်ဂျက်ကို 'တန်ဖိုးတစ်စုံတစ်ရာမရှိ' (သို့) 'မည်သည့် တန်ဖိုးကိုမှ ကိုယ်စားမပြု' ဆိုတဲ့ အဓိပ္ပါယ်အတွက် အသုံးပြုတာပါ။

၈.၃ Exceptions

ပုံမှန်အားဖြင့်တော့ ဖန်ရှင်တစ်ခုဟာ သူလုပ်ဆောင်ပေးရမဲ့ ကိစ္စကို ပြီးမြောက် အောင်မြင်အောင် ဆောင်ရွက်ပေးရမှာပါ။ ဒါပေမဲ့ ပုံမှန်မဟုတ်တဲ့ (သို့) မမျှော်လင့်ထားတဲ့ အခြေအနေ တစ်စုံတစ်ရာကြောင့် ဖန်ရှင်တစ်ခုဟာ သူလုပ်ဆောင်ပေးရမဲ့ကိစ္စကို ပြီးအောင်ဆက်လုပ်ပေးလို့ မရနိုင်တော့တာ ဖြစ်နိုင်ပါတယ်။ 'ပုံမှန်မဟုတ်တဲ့ (သို့) မမျှော်လင့်ထားတဲ့ အခြေအနေ' ဆိုတာ ဘယ်လိုမျိုးပါလဲ။ အီးမေးလ်ပို့ဖို့ `send_email` ဖန်ရှင် ခေါ်တယ်ဆိုပါစို့။ အင်တာနက် ကွန်နက်ရှင် ရှိရပါမယ်။ မရှိရင် `send_email` က အီးမေးလ်ပို့လို့ မရနိုင်ပါဘူး။ `date` အော့ဘ်ဂျက်တစ်ခု ဖန်တီးမယ်ဆိုပါစို့။ လနဲ့ ရက် မဖြစ်နိုင်တဲ့ ဂဏန်းတွေ ထည့်ပေးရင် အော့ဘ်ဂျက် ဖန်တီးလို့မရနိုင်ပါဘူး (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 ဖိုင်နံမည် (သို့) path လမ်းကြောင်းမှားနေရင် ဖွင့်လို့မရပါဘူး။ ဖိုင်သိမ်းတဲ့အခါမှာလည်း ခွင့်မပြုထားတဲ့ နေရာမှာ သိမ်းလို့မရပါဘူး။ ဖျက်ပစ်မယ်ဆိုလည်း ခွင့်မပြုတဲ့ဖိုင်ကို ဖျက်လို့မရဘူး။ ဖိုင်စနစ်နဲ့ သက်ဆိုင်တဲ့ ဖန်ရှင်တွေကို အခန်း (၁၀) မှာ လေ့လာကြတဲ့အခါ ဒီလိုမျိုး ပြဿနာတွေ ကြုံတွေ့ရမှာပါ။

```
>>> open('abc.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'abc.txt'
```

အခုဖော်ပြခဲ့တာတွေဟာ 'ပုံမှန်မဟုတ်တဲ့ (သို့) မမျှော်လင့်ထားတဲ့ အခြေအနေ' ဥပမာတချို့သာ ဖြစ်ပါတယ်။ နောက်ပိုင်းမှာ အခြားဟာတွေ ထပ်တွေ့ရမှာပါ။ ပရိုဂရမ်တစ်ခု run နေတဲ့အချိန် ဒီလို အခြေအနေတွေ ဖြစ်လာခဲ့ရင် ပရိုဂရမ်က ရပ်သွားပြီး ဆက်လက် အလုပ် မလုပ်ပေးနိုင်တော့တာမျိုး မဖြစ်သင့်ပါဘူး။ ဒီလိုမဖြစ်အောင် ကိုင်တွယ်ထိန်းကြောင်း ပေးဖို့အတွက် ခေတ်မီ programming language တွေ အားလုံးလိုလိုမှာ ရိုးရှင်းတဲ့ နည်းစနစ်တစ်ခု ထည့်သွင်းပေးထားပါတယ်။ အဲဒီ နည်းစနစ်ကတော့ *exception-handling* ပဲ ဖြစ်ပါတယ်။

raise -ing Exceptions

Exception-handling မှာ အပိုင်း နှစ်ပိုင်းပါဝင်တယ်။ ပထမတစ်ခုက တစ်ခုခု ပြဿနာဖြစ်နေပြီ ဆိုတာ အသိပေးတဲ့ အပိုင်း။ ဖန်ရှင်တစ်ခုဟာ ပုံမှန်မဟုတ်တဲ့ ပြဿနာကြောင့် သူ့တာဝန်ကို ပြီးမြောက်မှန်ကန်အောင် မလုပ်ပေးနိုင်တော့တဲ့အခါ ဖန်ရှင်ခေါ်တဲ့သူ (သို့) ခေါ်ထားတဲ့နေရာ ကို အသိပေးနိုင်ဖို့ လို

ပါတယ်။ ဒါမှလည်း ဖြစ်တဲ့ ပြဿနာပေါ် မူတည်ပြီး ဘာလုပ်ရမလဲ ဆုံးဖြတ်လို့ ရမယ်။ (အီးမေးလ်ပို့တာ အင်တာနက်မရရင် ခဏစောင့်ပါလို့ 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 ပရင့်ထုတ်ပေးမှာပါ။ အကြိမ်အရေအတွက်က အပေါင်း ကိန်းပြည့်ဂဏန်း ဖြစ်သင့်ပါတယ်။ မဟုတ်ဘူးဆိုရင် ဖန်ရှင်ခေါ်တဲ့အခါ ထည့်ပေးတဲ့ အကြိမ်အရေအတွက် အကြောင်းတစ်ခုခုကြောင့် မှားနေတာပဲ ဖြစ်ရမယ်။ ဒီလိုဖြစ်လာခဲ့ရင် တစ်ခုခု မှားနေပြီဆိုတာ အသိပေးဖို့အတွက် `raise` စတိတ်မန့်ကို အသုံးပြုနိုင်တယ်။ `isinstance` ဖန်ရှင်က ဗေရီရေဘဲလ်ရဲ့ တိုက်ပ်ကို စစ်ပေးတာပါ။ `n` ဟာ `int` ဖြစ်ရင် `isinstance(n, int)` က `True` ရမှာပါ။ အပေါင်းကိန်းပြည့် မဟုတ်ခဲ့ရင်

```
raise ValueError('Positive integer expected')
```

နဲ့ ပြဿနာကို အသိပေးပါတယ်။ ဒါကို exception ကို *raise* လုပ်တယ်လို့ ခေါ်တယ်။ `ValueError` ကတော့ exception (ပုံမှန်မဟုတ်တဲ့/မမျှော်လင့်ထားတဲ့ အခြေအနေ/ပြဿနာကို ဆိုလို) ကို ဖော်ပြတဲ့ အော့ဘ်ဂျက်ပါ။ `ValueError` အပြင် `ArithmeticError`, `FileNotFoundError` စသည်ဖြင့် ဖြစ်တဲ့ exception ပေါ်မူတည်ပြီး သင့်တော်တဲ့ အော့ဘ်ဂျက်ကို `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 မက်ဆေ့ချ် တွေပြပြီး ပရိုဂရမ် 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
```

တတိယဖန်ရှင် ဆက်မလုပ်တဲ့အတွက် `Hola` သုံးခါ မပါတာ သတိပြုပါ။

Handling Exceptions

Exception-handling ရဲ့ ဒုတိယပိုင်းကတော့ handle (ကိုင်တွယ် ထိန်းကျောင်းတာကို ဆိုလို) လုပ်တဲ့ ကိစ္စဖြစ်ပါတယ်။ `raise` လုပ်လိုက်တဲ့ exception ကို handle မလုပ်ရင် ပရိုဂရမ်ဟာ ဆက်အလုပ်မလုပ်နိုင်တော့ဘဲ ရပ်သွားမှာပါ။ Handle လုပ်ဖို့ `try` နဲ့ `except` ကို သုံးရပါတယ်။

ဖန်ရှင်တစ်ခုက `raise` လုပ်လိုက်တဲ့ exception ကို handle လုပ်ဖို့ရည်ရွယ်ချက်ရှိရင် အဲဒီဖန်ရှင်ကို `try` ဘလောက်ထဲမှာ ခေါ်ရပါမယ်။ Exception ဖြစ်ခဲ့ရင် ဘယ်လို handle လုပ်ချင်လဲ။ ဒီအပိုင်းကိုတော့ `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။ အခြားဟာတွေဆိုရင် handle မလုပ်ဘူးပေါ့။ `FileNotFoundError` အတွက်ဆိုရင် `except FileNotFoundError as err:` ဖြစ်ပါမယ်။ `err` က exception ကို ကိုယ်စားပြုတဲ့ ဗေရီရေဘဲလ် (အခြား နံမည်ဖြစ်လို့ရတယ်)။ Exception ဖြစ်ခဲ့ရင် (သို့) raise လုပ်ခဲ့ရင် အဲ့ဒီ exception နဲ့ သက်ဆိုင်တဲ့ အချက်အလက်တွေကို `err` ကနေတစ်ဆင့် ရယူနိုင်ပါတယ်။ သုံးဖို့မလိုရင်တော့ `as err` မပါဘဲ `except ValueError:` နဲ့ ရတယ်။

`try` ဘလောက်ထဲမှာ exception ဖြစ်ခဲ့ရင် ဖြစ်တဲ့နေရာကနေ သက်ဆိုင်တဲ့ `except` ဘလောက်ဆီကို 'ချက်ချင်း' ရောက်သွားမှာပါ။ မဖြစ်ခဲ့ရင်တော့ `try` ဘလောက် ပြီးတဲ့အထိ လုပ်ဆောင်ပြီး `except` ဘလောက်ကို လစ်လျူရှု သွားမှာပါ။ 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 ကွန်းမန့်ရေးထားတဲ့ လိုင်းကို မလုပ်ပါဘူး။ `except` ဘလောက်နဲ့ အောက်ဆုံး `print` လုပ်တဲ့အထိ ဆက်အလုပ်လုပ်သွားတယ်။ 2 ထည့်ပြီး စမ်းကြည့်ရင် exception မဖြစ်ဘူး။ `try` ဘလောက် ပြီးတဲ့ထိ ပုံမှန်အတိုင်း လုပ်ဆောင်တယ်။ Exception မဖြစ်တော့ `except` ဘလောက် အလုပ် မလုပ်ဘဲ ကျော်သွားပါတယ်။

Exception-handling ကို အသုံးပြုပြီး အင်တီဂျာ တစ်ခု input ထည့်ခိုင်းတဲ့ ဖန်ရှင်ကို အခုလို ရေးနိုင်ပါတယ်။

```
# 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)
```

အင်တီဂျာမဟုတ်တဲ့ တန်ဖိုး ထည့်ပေးရင် `int(input(prompt))` မှာ `ValueError` exception ဖြစ်ပြီး handle လုပ်တဲ့ ဘလောက်ကို ရောက်သွားမှာပါ။ ဒီတော့ ဖန်ရှင်က return မဖြစ်ဘူး။ `while` loop နောက်တစ်ကြိမ် ထပ်ကျော့ပါတယ်။ အင်တီဂျာ ပြောင်းလို့ရမဲ့ တန်ဖိုး ထည့်ပေးမှပဲ exception မဖြစ်ဘဲ return လုပ်ပါလိမ့်မယ်။ (`return` လုပ်ရင် ဖန်ရှင်ခေါ်တဲ့ဆီကို ချက်ချင်း ပြန်ရောက်သွားတဲ့ အတွက် loop ကနေလည်း ထွက်သွားစေတယ်)။

လိုက်ဘရီ ဖန်ရှင်တွေ အသုံးပြုတာပဲဖြစ်ဖြစ်၊ ကိုယ်ပိုင် ဖန်ရှင် သတ်မှတ်တာပဲ ဖြစ်ဖြစ် exception တွေနဲ့ exception-handling အခြေခံအဆင့် နားလည်ထားဖို့ လိုအပ်တယ်။ Exception-handling နဲ့ ပါတ်သက်ပြီး အခန်း (၁၀) မှာ ဒီ့ထက် ကျယ်ကျယ်ပြန့်ပြန့် ဆက်လက် လေ့လာရအုံးမှာပါ။ အခုတော့ ဒီလောက်နဲ့ ခဏရပ်ထားပြီး လက်တွေ့နဲ့ ပိုနီးစပ်တဲ့ အသုံးချဥပမာတချို့ကို ဆက်ကြည့်ရအောင်။

၈.၄ ဖန်ရှင်နှင့် ပရိုဂရမ်ဒီဇိုင်း ဥပမာ (၁) "Insurance Premium"

နှစ် နှစ်ဆယ် သက်တမ်းကာလ $500,000 အသက်အာမခံထားရင် ကျန်းမာတဲ့ အသက် (၃၀) အရွယ် အမျိုးသမီး တစ်ယောက်အတွက် နှစ်စဉ်ပျမ်းမျှ အာမခံကြေး (premium) $229 ဒေါ်လာ ကုန်ကျတယ်။ ရွယ်တူ အမျိုးသားတစ်ယောက် ဆိုရင်တော့ $373 ဒေါ်လာပါ။ ဒါကယေဘုယျ သဘောကိုပြောတာ။ အပြင်မှာ အသက်အာမခံထားရင် သက်တမ်းကာလ၊ အကျိုးခံစားနိုင်မည့် ဌေပမာဏ (coverage)၊ ကျန်းမာရေး၊ အသက်အရွယ် စတဲ့ အချက်တွေပေါ် မူတည်ပြီး တစ်ဦးချင်းအတွက် အာမခံကြေး သီးသန့် တွက်ချက်တာပါ။

အာမခံသက်တမ်း နှစ်ဆယ်နှစ်အတွက်ပဲ စဉ်းစားပါမယ်။ အသက် (၃၀) အတွက် premium လို့ ပြောပေမဲ့ တကယ်တမ်းက အသက် (၁၈) နှစ် ကနေ (၃၀) (အပါအဝင်) အတွက် premium ကို ဆိုလိုတာပါ။ ထိုနည်းတူစွာ အသက် (၄၀)၊ (၅၀)၊ (၆၀) premium ဟာ (၃၁) နှစ် ကနေ (၄၀)၊ (၄၁) နှစ် ကနေ (၅၀)၊ (၅၁) နှစ် ကနေ (၆၀) ကြား (အပါအဝင်) ကို ဆိုလိုတာဖြစ်တယ်။ (၁၈) နှစ်အောက်နဲ့ (၆၀) အထက်ဆိုရင်တော့ အကျိုးမဝင်ပါဘူး (အာမခံ ထားလို့ မရဘူး ယူဆပါ)။ ဇယား (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 အသက်အာမခံကြေး (မ)

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

တေဘဲလ် ၈.၂ နှစ်နှစ်ဆယ် အသက်အာမခံကြေး (ကျား)

မွေးသက္ကရာဇ်၊ အကျိုးခံစားလိုသည့် ပမာဏ (coverage amount)၊ ကျား/မ အလိုက် တစ်နှစ်ပျမ်းမျှ ပရီမီယံကြေး တွက်ပေးတဲ့ ပရိုဂရမ်တစ်ခု 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

ပရိုဂရမ် တစ်ခု ဒီဇိုင်းလုပ် ရေးသားပုံအဆင့်ဆင့်ကို ဆက်လက်ဖော်ပြပါမယ်။ လုပ်ငန်းသဘောနဲ့ သက်ဆိုင်တဲ့ အပိုင်းနဲ့ input/output အပိုင်းခွဲ ပြီးရေးလေ့ရှိတယ်။ (ကီးဘုဒ်ကနေ ထည့်ပေးတာကို ဖတ်တာ၊ စခရင်မှာ ရလဒ်ကို ထုတ်ပြတာ စတာတွေကို input/output အပိုင်းမှာ ပါဝင်တယ်)။

လုပ်ငန်းသဘောနဲ့ ဆိုင်တဲ့အပိုင်းကို အရင်ကြည့်ရအောင်။ Premium ကြေး ဇယားနှစ်ခုပါ ဒေတာတွေကို သိမ်းထားဖို့ လိုတယ်။ Row, column တွေနဲ့ တေဘဲလ်ပုံစံမျိုး 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 ကျား/မ၊ အသက်နဲ့ coverage ပေါ်မူတည်ပြီး ထုတ်ယူရမှာပါ။ ကျား (၄၅) နှစ်၊ coverage (၅၀၀,၀၀၀) အတွက် premium ကြေးကို `MALE_PREM[1][2]` နဲ့ ယူရမှာပါ။

Premium ကြေးကို ဖန်ရှင်တစ်ခုနဲ့ အခုလိုမျိုး ယူလို့ရသင့်တယ်။ အသက် အရွယ်၊ ကျား/မ၊ 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 တချို့ ကြေငြာထားတယ်။ `COVERAGES` က coverage ပမာဏတွေ ပါတဲ့ list ဖြစ်တယ်။ ဘာအတွက်လဲ ခဏနေ တွေ့ရပါမယ်။ `age`, `gender` နဲ့ `coverage` က လိုအပ်တဲ့ ဖန်ရှင်ပါရာမီတာတွေပါ။ ဖန်ရှင် တစ်ခုဟာ လက်မခံနိုင်တဲ့ သို့မဟုတ် မဖြစ်သင့်တဲ့ ပါရာမီတာ တန်ဖိုးတွေအတွက် အဖြေထုတ်ပေးဖို့ မဖြစ်နိုင်ပါဘူး။ ဖန်ရှင်က သူလုပ်ဆောင်ရမဲ့ ကိစ္စကို အကြောင်းတစ်ခုခုကြောင့် မလုပ်ပေးနိုင်တဲ့အခါ exception raise လုပ်လေ့ရှိတယ်ဆိုတာ ဒီ့မတိုင်ခင် စက်ရှင်မှာ တွေ့ခဲ့ရတယ်။ အခုဖန်ရှင်ကလည်း အသက်အရွယ် လိုအပ်ချက် မကိုက်ညီရင် exception raise လုပ်တယ်။ Coverage ပမာဏ လေးမျိုးထဲမှာ မပါဝင်ရင်လည်း raise လုပ်ထားပါတယ်။ `COVERAGES.index(coverage)` က `ValueError` ဖြစ်ရင် `coverage` က list ထဲမှာ မရှိလို့ မှားနေလို့ပဲ။ ဒီလိုဖြစ်ခဲ့ရင် `except` ဘလောက်ကလည်း `ValueError` ကိုပဲ `raise` လုပ်တယ်။ ဖြစ်တဲ့ပြဿနာကို တိတိကျကျ ဖော်ပြတဲ့မက်ဆေ့ချ်ဖြစ်ဖို့ဟာ 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 သီးသန့်ဖန်ရှင်တစ်ခု ခွဲထုတ်လိုက်ရင် ပိုကောင်းနိုင်တယ်။ `age` ကနေ မှန်ကန်တဲ့ ကော်လံ index ထုတ်ပေးကိစ္စ တစ်ခုကိုပဲ သီးသန့်လုပ်ဆောင်ပေးတာပေါ့။

```
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 သက်ဆိုင်တာကို အဓိကလုပ်တာဆိုတော့ 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 အဲ့ဒီ exception ကိုပဲ ပြန် raise လုပ်လိုက်တယ် (raise age_err)။ Exception ဖြစ်မဲ့ input တွေရော၊ 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။ ဘာ့ကြောင့်လဲ အကြောင်းအရင်းကို နောက်ပိုင်းမှာ သိရမှာပါ။

Input/Output အပိုင်း

Input သုံးခုထည့်ပေးရမယ်။ မွေးသက္ကရာဇ် (date of birth) , coverage amount နဲ့ ကျား/မ (gender) တို့ဖြစ်တယ်။ အသုံးပြုသူအနေနဲ့ အဆင်ပြေဆုံး၊ အလွယ်ကူဆုံးဖြစ်အောင်၊ အမှားအယွင်းနည်းနိုင်သမျှနည်းအောင် စဉ်းစားသင့်တယ်။

မွေးသက္ကရာဇ်ကို ကြည့်ရအောင်။ ရက်စွဲကို နိုင်ငံနဲ့ နေရာဒေသ ပေါ်မူတည်ပြီး ဖော့မတ်အမျိုးမျိုးနဲ့ ရေးကြတယ်။ 04/05/2020, 04-05-2020, Apr-4-2020 စသည်ဖြင့်။ ခုနှစ်ကို ဂဏန်း နှစ်လုံးပဲ ရေးတာလည်း ရှိတယ်။ ရက်နဲ့လကို ရှေ့မှာ သုညမပါဘဲလည်း ရေးတယ်။ ဥပမာ 4/5/2020, 4/5/20, Apr-4-2020 ။ ဖော့မတ်တစ်မျိုးကိုပဲ သတ်သတ်မှတ်မှတ် သုံးသင့်တာ ဖြစ်ပေမဲ့ ဆော့ဖ်ဝဲအပ်သူက ဖော့မတ် သုံးမျိုးနဲ့ ထည့်လို့ရအောင် လုပ်ပေးဖို့ တောင်းဆိုထားတယ်။

01-01-2024
01/01/2024
Jan-1-2024

မွေးသက္ကရာဇ်ကို ဒီဖော့မတ်သုံးမျိုးနဲ့ ပရိုဂရမ်က လက်ခံရပါမယ်။ input ဖန်ရှင်က ကီးဘုဒ်ကထည့်ပေးတာကို string အနေနဲ့ ပြန်ပေးပါတယ်။

မွေးသက္ကရာဇ်ကနေ အသက်ကို တွက်ယူရမှာပါ။ ဒီအတွက် ဖန်ရှင် (ဥပမာ calc_age) သတ်မှတ်နိုင်တယ်။ နေ့ရက်၊ အချိန်နဲ့ သက်ဆိုင်တဲ့ အတွက်အချက်တွေ အတွက် စာသားကို အသုံးမပြုသင့်ဘူး။ date, datetime စတာတွေ သုံးသင့်တယ်။ ဒါကြောင့် စာသားကနေ မွေးသက္ကရာဇ်ကို ဖော်ပြတဲ့ date (သို့) datetime အော့ဘ်ဂျက် ပြောင်းဖို့ လိုပါမယ်။ '01-01-2024' ကနေ 1, 1, 2024 ဂဏန်းတွေ 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 ခြားထားရင် 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' ဖော့မတ်က နည်းနည်း ပိုရှုပ်မယ်။ လက 'Jan', 'Feb' စသည်ဖြင့် စာသားဖြစ်နေတယ်။ အခုလို dictionary တစ်ခု ရှိရင် လနံပတ်ကနေ သက်ဆိုင်တဲ့ ဂဏန်းကို အလွယ်တကူ ရနိုင်ပါတယ်

```
>>> 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 အော့ဘ်ဂျက် 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)
```

နေ့ရက်ကို ဖော့မတ် သုံးမျိုးနဲ့ လက်ခံနိုင်တဲ့ 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]), mths[parts[0]], int(parts[1]))
```

သတ်မှတ်ပါမယ်။ replace ဖန်ရှင်သုံးပြီး / ကို - နဲ့ အစားထိုးလိုက်တယ်။ ဒါဆိုရင် ဖော့မတ် နှစ်ခုပဲ စဉ်းစားဖို့လိုတော့မယ် (10/01/2024 ကနေ 10-01-2024 ဖြစ်သွားမှာပါ)။ ပြီးတော့မှ split လုပ်တယ်။ ရှေ့ဆုံးအပိုင်း parts[0] ဂဏန်း ဟုတ်လား isdigit နဲ့ စစ်ထားတယ်။ ဖော့မတ် အမှန်နဲ့ စမ်းကြည့်ရင် date အော့ဘ်ဂျက် return ပြန်ပေးပါတယ်

```
print(parse_date_str('12-03-1995'))
print(parse_date_str('12/03/1995'))
print(parse_date_str('Mar-12-1995'))
```

ဖော့မတ် မမှန်ရင်၊ ဒါမှမဟုတ် အခြားအကြောင်းတစ်ခုခုကြောင့် date ပြောင်းလို့မရရင် exception ဖြစ်တယ်။ Fab စာလုံးပေါင်း မှားရင် KeyError ဖြစ်တယ်

```
# Spelling error: Fab instead of Feb
print(parse_date_str('Fab-12-1995'))
```

Error Output:

```
Traceback (most recent call last):
  File ".../ch08/insurance_prem2.py", line 20, in <module>
    print(parse_date_str('Fab-12-1995'))
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File ".../ch08/insurance_prem2.py", line 13, in parse_date_str
    return date(int(parts[2]), mths[parts[0]], int(parts[1]))
                               ~~~~^^^^^^^^^^^
KeyError: 'Fab'
```

Dictionary ထဲမှာ Fab ကီး မရှိတဲ့အတွက် ဒီပြဿနာဖြစ်တာပါ။ မဖြစ်နိုင်တဲ့ ခုနှစ်၊ လ၊ ရက် တန်ဖိုးတွေဆိုရင် ValueError ဖြစ်တယ် (ဥပမာ parse_date_str('30-02-1995'))။

အခုဖော်ပြခဲ့တဲ့ နည်းအပြင် အခြားနည်းလမ်းတွေလည်း ရှိရမှာပါ။ datetime ကလပ်စ်မှာ နေ့ရက်ကို စာသားကနေ datetime ပြောင်းပေးတဲ့ strptime ဖန်ရှင်ရှိတယ်။ သူ့ကို အသုံးပြုပြီး parse_date_str ကို ဘယ်လိုရေးလို့ 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 အချိန်နေ့ရက်ကို datetime အော့ဘ်ဂျက် ပြောင်းပေးဖို့ ဖော့မတ်ကုဒ် (format code) လို့ခေါ်တဲ့ % နဲ့ စတဲ့ ကာရက်တာတွေကို လက်ခံတယ်။ **%d** က ရက်၊ **%m** က လကို ဂဏန်း တစ်လုံး (သို့) နှစ်လုံးနဲ့ ဆိုတဲ့ အဓိပ္ပါယ် (ဥပမာ ကိုးရက်နေ့ကို 09 သို့မဟုတ် 9၊ ငါးလပိုင်းကို 05 သို့မဟုတ် 5)။ **%Y** က ခုနှစ်ကို ဂဏန်းလေးလုံးနဲ့ ရေးတယ်လို့ ဆိုလိုတာပါ။ **%b** ကတော့ လရဲ့နံမည်ကို Jan, Feb, Mar စသည်ဖြင့် အတိုကောက်ရေးတယ်လို့ ဆိုလိုတယ်။ 'Jan-01-2024' ကို ပြောင်းမယ်ဆိုရင် သူ့ရဲ့ဖော့မတ်နဲ့ ကိုက်ညီတဲ့ '%b-%d-%Y' ကို ဖန်ရှင်ခေါ်တဲ့အခါ ထည့်ပေးရပါမယ်။ '30/Dec/2024' ဆိုရင် '%d/%b/%Y'၊ '30-12-2024' အတွက် '%d-%m-%Y' ဖြစ်မှာပါ။ ဖော့မတ်ကုဒ် သုံးထားတဲ့ parse_date_str2 ကို အခုလို သတ်မှတ်လို့ရပါမယ်

```
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 ပြောင်းလို့မရရင် ဒီကိုရောက်လာမယ်
    raise ValueError(f"{dtstr} doesn't match any of acceptable formats")
```

dtstr ကို ဖော့မတ်တစ်ခုချင်း ပြောင်းကြည့်တယ်။ ပြောင်းလို့ရရင် return ဖြစ်သွားမယ်။ ပြောင်းလို့မရရင် strptime က ValueError raise လုပ်တယ်။ ဖော့မတ်တစ်ခုနဲ့ ပြောင်းလို့မရရင် နောက်တစ်ခုနဲ့ရနိုင်ပါတယ်။ ဒါကြောင့် ValueError ဖြစ်ခဲ့ရင် pass ပဲ လုပ်ပေးလိုက်တယ် (ဘာမှလုပ်စရာမလိုရင် pass စတိတ်မန့် သုံးတာပါ)။ ဖော့မတ် သုံးမျိုးလုံးနဲ့ အဆင်မပြေရင်တော့ တစ်ခုခု မှားနေလို့ပဲ။ 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()
```

ပရိုဂရမ်တည်ဆောက်တဲ့အခါ ကိစ္စတစ်ခုကို ဖြေရှင်းလို့ရတဲ့ နည်းလမ်းက တစ်ခုတည်းပဲ ဖြစ်လေ့မရှိတာကတော့ ထုံးစံပါပဲ။ ဖြစ်နိုင်တဲ့ ဖြေရှင်းနည်းတွေထဲက အသင့်တော်ဆုံးတစ်ခုကို ရွေးချယ် အသုံးပြုရတာပါ။ နည်းလမ်းတစ်ခုက လက်ရှိမှာ အသင့်တော်ဆုံး ဖြစ်ပေမဲ့ နောင်တစ်ချိန်မှာ သူ့ထက် ပိုကောင်းတာ၊ 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 က ဖော့မတ် မမှန်မချင်း ပြန်ထည့်ခိုင်းမှာပါ။

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() ပြောင်းတဲ့နေရာ exception ဖြစ်မယ်။ Handle အပိုင်းမှာ ဒီအတွက် အယ်ရာပြပေးတယ်။ အင်တီဂျာတော့ ထည့်တာမှန်တယ်၊ ဒါပေမဲ့ တစ်နဲ့ လေးကြား မဟုတ်ရင်လည်း လက်ခံလို့ မဖြစ်ဘူး။ ဒီအတွက် if နဲ့ စစ်ပြီး 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!')
```

ကီးဘုဒ် input ဖတ်ဖို့ လိုအပ်တဲ့ ဖန်ရှင်အားလုံးလည်း ရှိပြီ။ နောက်ဆုံးတော့ ပရိုဂရမ်တစ်ခုလုံးရဲ့ top level ဖန်ရှင်က အခုလိုပါ။ ရိုးရိုး 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 ပြပေးပါတယ်။ ထွက်ချင်ရင် quit ထည့်ပေးရမယ်။ Enter/return နှိပ်ရင် နောက်တစ်ခါ input ပြန်တောင်းပြီး ဆက်အလုပ်လုပ်နေမှာပါ။ ကိုယ်ပိုင် သတ်မှတ်တဲ့ ဖန်ရှင်အားလုံးကနေ ဖြစ်လာနိုင်တဲ့ exception အားလုံးကို သင့်တော်တဲ့နေရာမှာ handle လုပ်ထားတဲ့အတွက် ဘာပြဿနာမှ မဖြစ်နိုင်တော့ဘူးလို့ ယူဆစရာ အကြောင်းရှိပါတယ်။ ဒါပေမဲ့ အခြားကိုယ် မမျှော်မှန်းနိုင်တဲ့ exception တွေလည်း ရှိနိုင်ပါသေးတယ်။ (ဥပမာ ကွန်ပျူတာ operating system နဲ့ သက်ဆိုင်တဲ့ `OSError` ၊ မမ်မိုရီ မလုံလောက်ရင် `MemoryError`)။ ဒီလို exception တွေဖြစ်ရင် ပရိုဂရမ် ပုံမှန်ဆက်အလုပ် လုပ်နိုင်ဖို့လည်း မဖြစ်နိုင်တာ များပါတယ်။ ပရိုဂရမ်ကို ဆက် run လို့ မရတော့လောက်အောင် မဆိုးပေမဲ့ ကိုယ်မမျှော်လင့်ထားမိတဲ့ exception မျိုးလည်း ဖြစ်နိုင်တာပဲ။ ဒီလိုအခါမျိုးမှာ တစ်ခုခုတော့ ပြဿနာဖြစ်သွားတယ်၊ ပရိုဂရမ်ရေးတဲ့သူကို ဆက်သွယ်ပါ ဆိုတာမျိုး user ကို မက်ဆေ့ချ် ကောင်းကောင်းမွန်မွန် ပြပေးချင်တယ်ဆိုရင် အခုလို လုပ်လို့ရပါတယ်။

```
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 အားလုံးကို handle လုပ်ပေးပါတယ်။ except Exception ကတော့ သီးသန့် exception တစ်မျိုးအတွက်မဟုတ်ဘဲ ဘယ်အမျိုးအစားဖြစ်ဖြစ် handle လုပ်ချင်တဲ့အခါ သုံးရတာပါ။ ဒါနဲ့ ပါတ်သက်ပြီး Exception-handling အခန်းမှာ ဆက်လက်လေ့လာကြပါမယ်။

နောက်ပိုင်းမှာတော့ retrieve_prem မှာ ပထမ try...except ဖြုတ်လိုက်လို့ရမယ် (သို့ ဖြုတ်သင့်တယ်) 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 ဘီဟေဗီယာကတော့ မပြောင်းလဲသွားပါဘူး။ နဂိုအတိုင်းပါပဲ။ ဘာ့ကြောင့်လဲ အပိုင်း (၂) မှာ ဆက်ကြည့်ပါမယ် ...

ပရိုဂရမ် အစအဆုံး လေ့လာကြည့်ပါ။ 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')]]
```

```python
# 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[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 ပရိုဂရမ်ဒီဇိုင်း ဥပမာ (၂) “Colorful House”

အခုတစ်ခါ ပုံ (၈.၁) မှ ရောင်စုံအိမ်ပုံလေး ရေးဆွဲကြပါမယ်။ အိမ်အမြင့်ဟာ အိမ်အောက်ခြေကနေ အိမ်ခေါင်မိုး ထိပ်ထိ ဖြစ်တယ်။ အမြင့်နဲ့ အကျယ်ဟာ အိမ်အရွယ်အစားကို ဖော်ပြတယ်။ ပုံဆွဲတဲ့အခါ အိမ်အရွယ်အစားနဲ့ အရောင် စိတ်ကြိုက် အလွယ်တကူ ပြောင်းလို့ရသင့်တယ်။ အိမ်တည်နေရာကို ဘယ်ဘက် အပေါ်ထောင့်အမှတ် နဲ့ဖော်ပြပါမယ် (ပုံတွင်ကြည့်ပါ) ။ အိမ်တစ်အိမ်လုံး ကွက်တိဝင်တဲ့ ထောင့်မှန်စတုဂံ ရဲ့ (x, y) လို့လည်း ယူဆနိုင်တယ်။

ခေါင်မိုးအမြင့် (အပြာရောင် တြိဂံပုံ) က အိမ်တစ်ခုလုံး အမြင့် လေးပုံတစ်ပုံ။ ကျန်လေးပုံ သုံးပုံက နံရံအမြင့် (အစိမ်းနုရောင် ထောင့်မှန်စတုဂံ) ဖြစ်တယ်။ နံရံကို အညီအမျှ လေးပိုင်းခွဲပြီး တစ်ပိုင်းစီရဲ့ ဗဟို နဲ့ ပြတင်းပေါက်ဗဟို တစ်ထပ်တည်းကျအောင် နေရာချထားရပါမယ်။

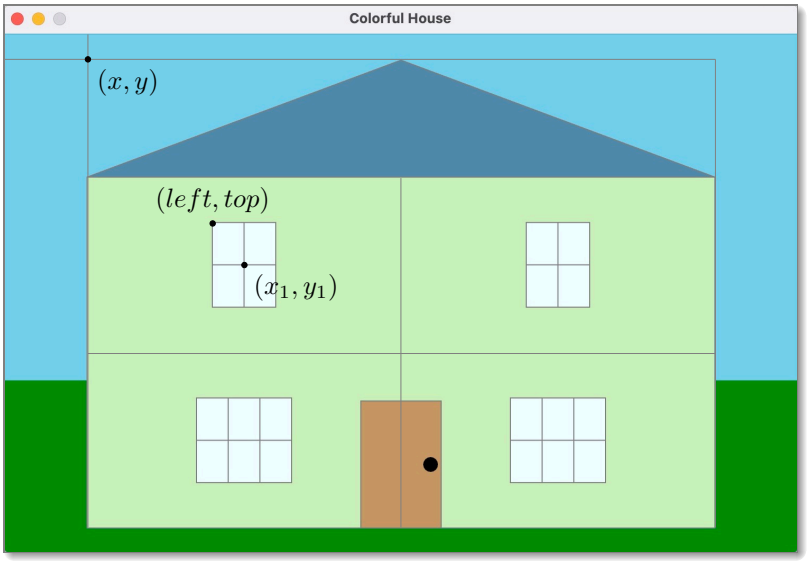
draw_window

ပြတင်းပေါက်တွေဆွဲဖို့ draw_window ဖန်ရှင် သတ်မှတ်သင့်တယ်။ ပြတင်းပေါက် အရွယ်အစား (မှန်ချပ် အရေအတွက်) ကို $rows \times columns$ နဲ့ ဖော်ပြနိုင်တယ်။ အပေါ်ပြတင်းပေါက် နှစ်ခုက 2×2 ဖြစ်ပြီး အောက်နှစ်ခုက 2×3 ။ အောက်ဖော်ပြပါ draw_window မှာ x, y, row, col တို့ဟာ ပြတင်းပေါက် ဗဟိုမှတ်, အရွယ်အစားနဲ့ အရောင် တို့အတွက် ပါရာမီတာတွေဖြစ်တယ်။ နံရံ ပထမတစ်ပိုင်းရဲ့ ဗဟိုမှတ် ဟာ (x_1, y_1) ဆိုပါစို့ (ပုံတွင်ကြည့်ပါ)။ ပြတင်းပေါက်ကို အခုလို

```
draw_window(x1, y1, 2, 2);
```

ဆွဲနိုင်ပါတယ်။ နံရံတစ်ပိုင်းစီအတွက် ဗဟိုမှတ်ရှာတာ သိပ်မခက်ဘူး။ ဒါကြောင့် ဗဟိုမှတ်ကို ပါရာမီတာ အနေနဲ့ ထားတာ အဆင်ပြေတယ်။ (ပြတင်းတစ်ပေါက်ချင်းအတွက် သူ့ရဲ့ ဘယ်ဘက်အပေါ် ထောင့်စွန်း မှတ် (x, y) တန်ဖိုးကို တွက်ကြည့်ပါ။ ဗဟိုမှတ်တွက်တာလောက် မလွယ်ကူတာ တွေ့ရလိမ့်မယ်)။

```python
# File: colorful_house.py
import arcade
from arcade.color import *

# x, y က ပြတင်းပေါက် အလယ်မှတ်ပါ။ ပုံထဲမှာ (x1, y1) 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):
        for j in range(col):
```



ပုံ ၈.၁ ရောင်စုံအိမ်လေး

```
            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 အခန်း (၇) စာမျက်နှာ (၁၀၈) checkerboard လိုပါပဲ။ တက္ကက်ချင်းရဲ့ left, right, top, bottom တွေ loop တစ်ကျော့တိုင်းမှာ မှန်အောင်တွက်နိုင်ရင် ပြတင်းပေါက်ဆွဲလို့ ရမှာပါ။ မှန်တစ်ချပ် အကျယ်၊ အမြင့် $30, 40$ သတ်မှတ်တယ်။ ပြတင်းပေါက်တစ်ခုလုံး အကျယ်၊ အမြင့်ကို

```
window_width = pane_width * col
window_height = pane_height * row
```

ဖော်မြူလာနဲ့ တွက်ရပါမယ်။ ပြတင်းပေါက် အလယ်မှတ် x, y ကနေ ဘယ်ဘက် အပေါ်ထောင့်စွန်းမှတ်ကို ရှာမယ်ဆိုရင်

```
top = y - window_height / 2
left = x - window_width / 2
```

(ပုံကိုကြည့်ပြီး x_1, y_1 အမှတ်ကနေ $left, top$ အမှတ်ကို ဘယ်လိုရှာမလဲ စဉ်းစားကြည့်ပါ)။ မှန်ချပ်တစ်ချပ်ချင်း ကိုဩဒိနိတ်တွေကို i နဲ့ j (မှန်ချပ်ရဲ့ row နဲ့ column နံပါတ်) ပေါ်မူတည်ပြီး အခုလို တွက်လို့ရမယ်

```
left + pane_width * j,          # မှန်ချပ်ရဲ့ left x
left + pane_width * (j+1),      # မှန်ချပ်ရဲ့ right x
top + pane_height * (i+1),      # မှန်ချပ်ရဲ့ bottom y
top + pane_height * i,          # မှန်ချပ်ရဲ့ top y
```

ဥပမာ $(left, top) = (100, 120)$ ဆိုပါစို့။ ညာဘက်အောက် မှန်ချပ်အတွက် i = 1, j = 1 ဖြစ်တယ် (i နဲ့ j တန်ဖိုး သုညက စတယ်ဆိုတာ အမှတ်ရပါ)။ အဲဒီမှန်ချပ်ရဲ့ ကိုဩဒိနိတ်တွေက

```
100 + 30 * 1            = 130
100 + 30 * (1 + 1)      = 160
120 + 40 * (1 + 1)      = 200
120 + 40 * 1            = 160
```

ဖြစ်ပါတယ်။

draw_house

draw_house ကို အောက်ပါအတိုင်း သတ်မှတ်ထားတယ်။ left နဲ့ top က အိမ်တည်နေရာ (x, y) လက်ခံတဲ့ ပါရာမီတာတွေပါ။ width နဲ့ height က အိမ်တစ်ခုလုံးရဲ့ အကျယ်နဲ့ အမြင့်အတွက်။

```
# left = x top = y (ပုံတွင်ကြည့်ပါ)
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)
```

အိမ်ခေါင်မိုး ထိပ်စွန်းမှတ်က အိမ်ကို အလယ်ကနေ ပိုင်းဖြတ်တဲ့ ဒေါင်လိုက်မျဉ်းပေါ်မှာပါ (ခေါက်ချိုးညီမျဉ်း)။ ဒီမျဉ်းပေါ်က x အမှတ်ကို middle လို့ ထားမယ်။ သူ့တန်ဖိုးက

```
middle = left + width / 2
```

ဖြစ်တယ်။ နံရံ အပေါ်အနားရဲ့ y အမှတ်ကို wall_top လို့ ထားတယ်။ သူ့တန်ဖိုးက

```
wall_top = top + height / 4
```

ဖြစ်မယ်။ top ကို အိမ်အမြင့်ရဲ့ လေးပုံတစ်ပုံ ပေါင်းပေးရမှာပါ။ နံရံ အောက်ဘက်အနားရဲ့ y အမှတ်ကို wall_bottom လို့ ထားတယ်။ သူ့တန်ဖိုးက

```
wall_bottom = top + height
```

နံရံ ညာဘက်အနားရဲ့ x အမှတ်ကို right လို့ ထားတယ်။ သူ့တန်ဖိုးက

```
right = left + width
```

(ရေပြင်ညီမျဉ်းပေါ်က အမှတ်အားလုံး y တန်ဖိုး တူညီတယ်၊ ဒေါင်လိုက်မျဉ်းပေါ်က အမှတ်အားလုံး x တန်ဖိုး တူညီတယ်လို့ သိထားပါ)။

ခေါင်မိုးကို draw_triangle_filled နဲ့ အခုလိုဆွဲနိုင်ပါတယ်။ တြိဂံရဲ့ ထောင့်စွန်းမှတ်တွေ ထည့်ပေးရမှာပါ။ (တြီဂံအနားသတ်ကို draw_triangle_outline နဲ့ ဆွဲတယ်။ ရှေ့စာမျက်နှာ draw_house မှာ ကြည့်ပါ)။

```
# draw roof
arcade.draw_triangle_filled(left, wall_top,    # ဘယ်ထောင့်စွန်း
                            right, wall_top,   # ညာထောင့်စွန်း
                            middle, top,       # အပေါ်ထိပ်စွန်း
                            AIR_FORCE_BLUE)
```

ပြတင်းပေါက်တွေရဲ့ အလယ်မှတ်တွေကို အောက်ပါအတိုင်း တွက်ထားတယ်။ ဘယ်ဘက် ပြတင်းနှစ်ပေါက် အလယ်မှတ် x တန်ဖိုး တူတယ်။ ညာဘက်နှစ်ခု အလယ်မှတ်လည်း x တန်ဖိုး တူတယ်။ အပေါ် ပြတင်းနှစ်ပေါက် အလယ်မှတ် y တန်ဖိုး တူတယ်။ အောက်နှစ်ခု အလယ်မှတ်လည်း y တန်ဖိုး တူတယ်။ နံရံအမြင့်က 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 ဒီလိုဆွဲရုံပဲ

```
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 ရှေ့စာမျက်နှာမှာ အစအဆုံး ပြန်လေ့လာကြည့်ပါ။

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 အခုလောက်ဆို 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
```

ဝင်းဒိုး အကျယ်နဲ့ အမြင့် $754, 492$ pixels ထားတယ်။ ဝင်းဒိုးရဲ့ ဘယ်၊ ညာနဲ့ အထက်၊ အောက် ဘောင်တွေကနေ အိမ်ကို $80, 25$ pixels ခွာထားတယ်။

နောက်ဆုံးတော့ Arcade လိုက်ဘရီ လိုအပ်ချက်အတိုင်း အခုလို တစ်ဆင့်ချင်း ပြင်ဆင်ပြီး ရောင်စုံ အိမ်ပုံလေး ဆွဲလို့ရပါပြီ

```
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()

# ဝင်းဦးအောက်ပိုင်း သုံးပုံတစ်ပုံ အစိမ်းရောင်ဖြစ်အောင် (မြက်ခင်းလေးပေါ့)
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 မိတ်ဆက်ပေးထားပါတယ်။ အဲဒီအပိုင်းဖတ်ထားမှ အခုရှင်းပြတာတွေကို သေချာနားလည်နိုင်မှာပါ။

ရောင်စုံအိမ်လေးဆွဲတဲ့ ပရိုဂရမ်ကို တစ်ဆက်တည်း ဆက်စပ် ကြည့်လို့အဆင်ပြေအောင် အစအဆုံး ထည့်ပေးထားတယ်။ လေ့လာကြည့်ပါ။

```
import arcade
from arcade.color import *


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):
        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)


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

    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 ပရိုဂရမ်က အတန်အသင့် အဆင်ပြေနေပါပြီ။ တချို့နေရာလေးတွေ ပိုကောင်းအောင် improve လုပ်လို့ရပါတယ်။ ပြင်ဖို့က သိပ်လည်းမခက်ဘူး။ ထောင့်မှန်စတုဂံတွေကို နှစ်ခါဆွဲနေရတယ်။ အရောင်ဖြည့်ဖို့ `draw_lrtb_rectangle_filled` နဲ့တစ်ခါ အနားသတ်အတွက် `draw_lrtb_rectangle_outline` နဲ့တစ်ခါ။ ဖန်ရှင်မှာ ထည့်ပေးတဲ့ တန်ဖိုးတွေကလည်း အားလုံး တူတူပဲ (အရောင်ကလွဲလို့)။ အခုလို ဖန်ရှင် သတ်မှတ်ထားလိုက်ရင် ပိုအဆင်ပြေသွားမှာပါ

```
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)
```

မီးခိုးရောင် 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 ရွေးချယ်သတ်မှတ်ခြင်း

ဖန်ရှင်သတ်မှတ်တဲ့အခါ 'ဘယ်ဟာတွေက ပါရာမီတာ ဖြစ်သင့်လဲ'၊ 'ဘယ်ဟာတွေကတော့ရော မဖြစ်သင့်ဘူးလား' ဝေခွဲဆုံးဖြတ်မရ ဖြစ်လေ့ရှိတာဟာ စလေ့လာတဲ့သူတွေရော အတွေ့အကြုံရှိပြီး သူတွေအတွက်ပါ ပုံမှန် ဖြစ်ရိုးဖြစ်စဉ် တစ်ခုပါပဲ။ ဖော်မြူလာ ရှိပြီးသား သင်္ချာဖန်ရှင်တွေအတွက်တော့ သိပ်ခေါင်းစားစရာ မလိုပါဘူး။ စက်ဝိုင်းဧရိယာရှာရင် အချင်းဝက်၊ တြိဂံဆိုရင် အခြေနဲ့ အမြင့်တို့ကို ပါရာမီတာ အနေနဲ့ ထားရမယ်ဆိုတာ သိသာတယ်။ ဒါပေမဲ့ တချို့နေရာတွေမှာ ထင်သလောက် မရိုးရှင်းပါဘူး။

ယေဘုယျအားဖြင့် 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 အရောင်ကွဲနေတယ်ဆိုပါစို့။ ဒါဆိုရင်တော့ `draw_window` ကို အခုလို ပြင်ပေးရပါမယ်

```
def draw_window(x, y, row, col, color):
    # ... ဒီနေရာက လိုင်းတွေ တချို့ကို ချန်ထားတယ်
    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)
```

အပေါ်ကို အဖြူရောင်၊ အောက်ကို မီးခိုး 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)
```

ဒီလိုဆွဲလို့ ရပါတယ်။

တကယ်လို့ မှန်ချပ်အရွယ်ပါ ဖန်ရှင်ခေါ်တဲ့အခါကျတော့မှ သတ်မှတ်လို့ရချင်ရင် 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 ထပ်ဖြည့်ခြင်းအားဖြင့် ပြတင်းပေါက်ကို အရောင် အမျိုးမျိုးနဲ့ ဆွဲလို့ရလာတယ်။ မှန်ချပ် အရွယ် အမျိုးမျိုးနဲ့ ဆွဲလို့ရလာတယ်။ ဖန်ရှင်ဟာ ပိုပြီး ကျန်နရယ်ကျလာတယ်၊ flexible ပိုဖြစ်လာတယ်။ ဒီတော့ တတ်နိုင်သမျှ ပါရာမီတာတွေ များများထားရင် ပိုကောင်းတယ် ယူဆရမှာလား။ ပါရာမီတာတွေ များလာတာနဲ့အမျှ သူတို့ကို ဘာအတွက်ထည့်ရတာလဲ၊ ဘာတန်ဖိုးထည့်သင့်လဲ စဉ်းစားဖို့လိုလာတယ်။ ဖန်ရှင်ကို အသုံးပြုရတာ ပိုခက်လာတယ်။ ဒါကြောင့် flexibility နဲ့ အသုံးပြုရ လွယ်ကူမှု ပေါ်မူတည်ပြီး ချင့်ချိန်သတ်မှတ်ရမှာပါ။

ကျန်နရယ်ကျတဲ့၊ flexible ဖြစ်တဲ့ ဖန်ရှင်တစ်ခုကို အခြေခံပြီး specific ဖြစ်တဲ့ ဖန်ရှင်တွေကို သတ်မှတ်ယူလို့ရတယ်။ ဒီဖန်ရှင်နှစ်ခုမှာ အပေါ်က 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)
```

2×2 ပုလဲရောင်နဲ့ 2×3 မီးခိုးရောင် ပြတင်းပေါက်ဆွဲဖို့ပါ။ နေရာအတွက် ပါရာမီတာ နှစ်ခုပဲ ပါတော့တယ်။ သုံးရတာ ပိုလွယ်သွားတာ သိသိသာသာတွေ့ရမှာပါ။ ဥပမာ ရောင်စုံအိမ်လေးမှာ အပေါ်ပြတင်းနှစ်ပေါက်ကို ပုလဲရောင်၊ အောက်နှစ်ခုကို မီးခိုးရောင် လိုချင်ရင် အခုလို ဆွဲရုံပါပဲ

```
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။ ဂျန်နရယ်ကျတဲ့၊ flexible ဖြစ်တဲ့ ဖန်ရှင်းကို အခြေခံပြီး specific ဖြစ်တဲ့ ဖန်ရှင် သတ်မှတ်တဲ့ နည်းလမ်းကို ပရိုဂရမ်မာတွေ မကြာခဏ အသုံးပြုလေ့ရှိတယ်။ ကိုယ်တိုင် အသုံးချတတ်အောင် လေ့လာထားသင့်ပါတယ်။ အသုံးတည့်မှာပါ။ နားလည်ဖို့ သိပ်ခက်ခဲတဲ့ ကိစ္စလည်း မဟုတ်ပါဘူး။

၈.၇ တန်ဖိုး ပြန်ပေးခြင်း၊ မပေးခြင်း

တချို့ ဖန်ရှင်တွေမှာ တန်ဖိုးပြန်ပေးသင့် (သို့) မပေးသင့် ဆုံးဖြတ်ရ ခက်တတ်ပါတယ်။ ဒါနဲ့ပါတ်သက်ပြီး ပထမဆုံး သိထားသင့်တာကတော့ တန်ဖိုး ပြန်ပေးခြင်း (သို့) မပေးခြင်းရဲ့ အကျိုးဆက်က ဘယ်လိုရှိမလဲ 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 အသုံးပြုနိုင်တယ်။ ဥပမာ greet လုပ်ပြီးရင် နေကောင်းလားဆက်မေးချင်တယ်၊ ဒါမှမဟုတ် စာသားထဲက အက္ခရာ/စကားလုံး အစားထိုးမယ် ဆိုပါစို့

```
print(greet_guest_rt("Joe") + '\nHow are you doing?')
print(greet_guest_rt('Kathy').replace('.', '!!!'))
```

ဖန်ရှင်ခေါ်တဲ့သူက ပြန်ရတဲ့ တန်ဖိုးကို လိုအပ်သလို ဆက်လက် အသုံးပြုလို့ ရပါမယ်။ တန်ဖိုးပြန်မရရင်တော့ ဒီလိုလုပ်လို့ ရမှာ မဟုတ်ပါဘူး။

နောက်တစ်ခုက တန်ဖိုးပြန်ပေးရင် မှန်/မမှန်စစ် (test) လုပ်ရတာ ပိုလွယ်တယ်။ ဒါက စာမျက်နှာ (၁၀၄) က ခရစ်စမတ် သစ်ပင်လေးကို ဖန်ရှင်ရေးထားတာပါ

```
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 ထုတ်မယ်ဆိုရင်

```
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၊ ဒါမှမဟုတ် ပိုနားလည်ရ လွယ်ကူအောင်) ပြင်ရေးမယ်ဆိုရင် ရလဒ် နဂိုအတိုင်း ဖြစ်မဖြစ် ပြန်စစ်ဖို့ လိုပါတယ်။ တကယ်လို့ christmas_tree က တန်ဖိုးပြန်မပေးဘဲ တစ်ခါတည်း print ထုတ်လိုက်ရင် စခရင်မှာ ပေါ်လာတဲ့ output ကိုကြည့်ပြီး လူကိုယ်တိုင် စစ်ဆေးမှပဲ မှန်/မမှန် သိနိုင်မှာပါ။ ပရိုဂရမ်ရေးပြီး အခုလို automated testing (လူက test မလုပ်ဘဲ ပရိုဂရမ်နဲ့ပဲ test လုပ်တာကို ဆိုလို) လုပ်ဖို့ မဖြစ်နိုင်ဘူး။

အခုဖော်ပြခဲ့သလို testing လုပ်တာဟာ *unit testing* ရဲ့ အခြေခံသဘောတရားပါပဲ။ ဖန်ရှင်တစ်ခု (သို့) မော်ဂျူးတစ်ခုကို သူ့ချည်းသီးခြား မှန်/မမှန် test လုပ်တာကို unit testing လို့ ခေါ်တာပါ။

တန်ဖိုးပြန်ပေးတဲ့ ဖန်ရှင်တွေဟာ unit test လုပ်ရ လွယ်ကူတာ တွေ့နိုင်ပါတယ်။ ဆော့ဖ်ဝဲတည်ဆောက်ရာမှာ *unit testing* ဟာ အရေးကြီးတဲ့ အခန်းကဏ္ဍတစ်ခု အနေနဲ့ ပါဝင်တယ်။ ကျယ်ပြန့်တဲ့ နယ်ပယ်ဖြစ်ပြီး သီးခြားလေ့လာဖို့ လိုအပ်ပါတယ်။ ဒီစာအုပ်မှာတော့ ဖန်ရှင်တန်ဖိုး ပြန်ပေးခြင်းနဲ့ ဆက်စပ်နေတဲ့ အတွက် အခြေခံလောက်လေးကိုပဲ ဗဟုသုတအနေနဲ့ ထည့်သွင်းဖော်ပြတာပါ။

၈.၈ `return`

`return` ဟာ ဖန်ရှင်ကနေ သူ့ကိုခေါ်တဲ့သူ (သို့) ခေါ်ထားတဲ့နေရာဆီ ပြန်ရောက်သွားစေတယ်လို့ သိထားပြီးပြီ။ 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 # ဒါမပါလည်းရတယ်
```

ရိုးရိုး `if` သုံးထားပေမဲ့ ကွန်ဒီရှင်မှန်ရင် `return` ဖြစ်သွားမှာမို့လို့ အောက်ကို မရောက်တော့ဘူး။ ဒီအတွက် cascading `if` အလုပ်လုပ်ပုံလိုပဲ ဖြစ်သွားတယ်။ အပေါ်မှာ မှန်ရင်အောက်ကို မရောက်လာနိုင်ဘူး။ အောက်က `if` တွေကို ရောက်လာတာဟာ အပေါ်က ကွန်ဒီရှင်တွေ မှားလို့ပဲ။ (အခုလိုမျိုးရေးသင့်တယ်လို့ မဆိုလိုပါ။ `return` ရဲ့ သဘောတရားကို သတိပြုမိအောင် ဥပမာပြတာပါ)။

ဖန်ရှင်မှာ loop ထဲကနေ `return` လုပ်လိုက်ရင် loop ကို `break` လိုက်သလို ဖန်ရှင်ခေါ်တဲ့နေရာဆီကိုလည်း ပြန်ရောက်သွားမှာပါ။

```
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)
```

မတော်တဆ 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 နောက်တစ်ကြောင်းကို ရောက်လာတဲ့ သဘောပဲ။ ဖန်ရှင်ဘလောက်အဆုံးထိ ရောက်လာပြီးရင် ခေါ်ခဲ့တဲ့နေရာကို ပြန်သွားဖို့ return (နောက်မှာ တန်ဖိုးမပါ) လုပ်တယ်လို့ ယူဆရမှာပါ။ တန်ဖိုးမပါတဲ့ return က None ကို ပြန်ပေးပါတယ်။ ဒီလောက်ဆိုရင် ဘာလို့ print(my_abs(0)) က None ရတာလဲ နားလည်းမှာပါ။ ဒီအမှားကို ပြင်ထားတဲ့ ဥပမာနှစ်ခုကို ပြထားပါတယ်။ ဒုတိယတစ်ခုက အတွေ့အကြုံရှိတဲ့ Python ပရိုဂရမ်မာတစ်ယောက် 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 (ကလပ်စ်များမှ အော့ဘ်ဂျက်များသို့)

အခန်း (၆) မှာ အော့ဘ်ဂျက်တချို့နဲ့ မိတ်ဆက်ပေးခဲ့တယ်။ `date`, `list`, `dict` စသည်ဖြင့်။ အော့ဘ်ဂျက်တွေကို ကလပ်စ်တစ်ခုကနေ တည်ဆောက်ယူရတာပါ။ `date` အော့ဘ်ဂျက်တွေကို `date` ကလပ်စ်၊ `Fraction` အော့ဘ်ဂျက်တွေကို `Fraction` ကလပ်စ်ကနေ ဖန်တီးယူတယ်။ အော့ဘ်ဂျက်တွေကို class တစ်ခုရဲ့ *instance* လို့လည်းခေါ်တယ်။ `Fraction` အော့ဘ်ဂျက်ကို `Fraction` ကလပ်စ် instance လို့လည်း ပြောလေ့ရှိတယ်။ အော့ဘ်ဂျက်တွေဟာ class instance တွေဖြစ်တယ်။ (အကြမ်းဖျဉ်းအားဖြင့် အော့ဘ်ဂျက်နဲ့ instance တူတယ်ပြောလို့ရပေမဲ့ အနက်အဓိပ္ပါယ်အရ အနုစိတ် ကွာဟချက်တွေ ရှိတာကိုလည်း inheritance အကြောင်း လေ့လာတဲ့အခါ တွေ့ရပါမယ်)။

ဒီအခန်းမှာ ကလပ်စ်အကြောင်းကို လေ့လာကြမှာပါ။ ကိုယ်ပိုင် ကလပ်စ်သတ်မှတ်ပြီး အဲ့ဒီကလပ်စ်ကနေ အော့ဘ်ဂျက်တွေ တည်ဆောက် ယူနိုင်တော့မှာဖြစ်တယ်။ အခြားသူတွေရေးထားပေးတဲ့ လိုက်ဘရီဖန်ရှင်တွေကို အသုံးပြုရာကနေ နောက်ပိုင်းမှာ ကိုယ်ပိုင် ဖန်ရှင်တွေ သတ်မှတ်လာနိုင်တာဟာ အဆင့်တစ်ဆင့် ပိုမြင့်လာသလိုပါပဲ။ ကိုယ်ပိုင်ကလပ်စ် ဒီဇိုင်းပြုလုပ် ဖန်တီးနိုင်လာတာဟာလည်း နောက်တစ်ဆင့် ထပ်မြင့်လာတယ်လို့ ဆိုရမှာပါ။ ပရိုဂရမ်တွေကို အမှားနည်းအောင်၊ ပြင်ဆင်ရလွယ်ကူအောင် စနစ်တကျ ဒီဇိုင်းလုပ် တည်ဆောက်လို့ရလာမှာ ဖြစ်ပါတယ်။

ကလပ်စ်ကို abstraction mechanism တစ်မျိုးလို့လည်း ရှုမြင်နိုင်တယ်။ Abstraction လုပ်တယ်ဆိုတာ ဘယ်လိုတည်ဆောက်ထားလဲ သိစရာမလိုဘဲ အသုံးပြုလို့ရစေတာကို ဆိုလိုတာ။ ဖန်ရှင်တွေနဲ့ abstraction လုပ်တာကို တွေ့ခဲ့ကြပြီးပါပြီ။ ကလပ်စ်တွေဟာလည်း abstraction အတွက် အထောက်အကူပြုတဲ့ နည်းလမ်းတစ်မျိုး (mechanism) ပါပဲ။ အများကြီး ပိုအစွမ်းထက်တဲ့ နည်းလမ်းလို့ ဆိုရမှာပါ။ အစွမ်းထက်လာတာနဲ့အမျှ သဘောတရားအရရော အသုံးချတဲ့အပိုင်းမှာပါ ပိုပြီးရှုပ်ထွေးခက်ခဲနိုင်ပါတယ်။ ဒီဇိုင်းပိုင်းအတွက် ကျယ်ကျယ်ပြန့်ပြန့် ဆက်လက်လေ့လာဖို့ လိုအပ်မှာဖြစ်ပြီး အတွေ့အကြုံနဲ့ပါ ပေါင်းစပ်ယူရမှာပါ။ စာချည်းဖတ်ရုံနဲ့ မရနိုင်ဘူး (စာရေးသူ ကိုယ်တွေ့ အတွေ့အကြုံအရ ကိုယ်ပိုင်အမြင်ကို ပြောခြင်းသာ)။ ဒါတွေက နောက်ပိုင်း ဆက်လက်လေ့လာဖို့အတွက်ပေါ့။ အခုတော့ အများကြီးမပြောတော့ဘဲ ကလပ်စ်တွေအကြောင်း စလိုက်ကြရအောင် . . .

၉.၁ `Account` Class (အကောင့် ကလပ်စ်)

ဘဏ်အကောင့်တွေကို `Account` အော့ဘ်ဂျက်နဲ့ ဖော်ပြမယ်ဆိုပါစို့။ အကောင့်နံပါတ်၊ ပိုင်ရှင်၊ လက်ကျန်ငွေ (balance) စတဲ့ အချက်အလက်တွေဟာ အပြင်က တကယ့် ဘဏ်အကောင့်တစ်ခုအတွက် အရေးပါပါတယ်။ မှတ်သား သိမ်းဆည်းထားရတယ်။ ဒါကြောင့် အပြင်က တကယ့် ဘဏ်အကောင့်ကို ထင်ဟပ်

ဖော်ပြတဲ့ ဆော့ဖ်ဝဲ အော့ဘ်ဂျက်တွေမှာလည်း ဖော်ပြပါ အချက်အလက်တွေ ပါဝင်သင့်တယ်ဆိုရင် ကျိုးကြောင်းဆီလျော်တယ်ပဲ ယူဆရမှာပါ။ အကောင့်နဲ့ ပါတ်သက်ပြီး အခြား အရေးပါတဲ့ အချက်အလက်တွေ ဒီ့ထက်မက ရှိပါတယ်။ ဥပမာ အကောင့် အမျိုးအစား (saving, current) ၊ ဖွင့်ရက် (open date)၊ ဘဏ်ခွဲအမည်၊ ငွေသွင်း/ငွေထုတ်/ငွေလွှဲမှတ်တမ်း စတဲ့အချက်တွေ ရှိပါတယ်။ တကယ့်လက်တွေ့မှာ လိုအပ်မှာဖြစ်ပေမဲ့ အခုဥပမာမှာ ရိုးရိုးရှင်းရှင်းဖြစ်ဖို့ အဲဒါတွေ ထည့်မစဉ်းစားဘဲ ချန်ခဲ့ရအောင်။ ကလပ်စ်ရဲ့ သဘောတရားကို ရှင်းပြဖို့အတွက် ပထမသုံးချက်နဲ့ လုံလောက်ပါတယ်။

ကေသီ့မှာ အကောင့်နံပါတ် 0086-6002-1111 နဲ့ လက်ကျန်ငွေ တစ်သိန်းရှိတဲ့ အကောင့်တစ်ခု ရှိတယ် ဆိုပါစို့။ Account ကလပ်စ်သာ ရှိမယ်ဆိုရင် အဲဒီ ကေသီ့အကောင့်ကို ကိုယ်စားပြုတဲ့ အော့ဘ်ဂျက်ကို အခုလို

```
acc1 = Account('Kathy',
               '0086-6002-1111',
               Decimal("100000.00"))
```

ဖန်တီးယူလို့ ရမှာပါ။ ထိုနည်းတူစွာ ငွေလက်ကျန် လေးသိန်းခွဲ၊ နံပါတ် 0086-6002-2211 နဲ့ စန္ဒီ့အကောင့် အော့ဘ်ဂျက်ကို ဒီလို

```
acc2 = Account('Sandy',
               '0086-6002-2211',
               Decimal("450000.00"))
```

ဖန်တီးယူနိုင်မယ်။ ဒါက Account ကလပ်စ်သာ ရှိခဲ့ရင် အော့ဘ်ဂျက် ဘယ်လို ဖန်တီးယူရမလဲ စဉ်းစားကြည့်တာပေါ့။ ကလပ်စ်မရှိတဲ့အတွက် အမှန်တကယ်တော့ မရသေး။

date, list, စတဲ့ အော့ဘ်ဂျက်တွေ အပေါ်မှာ အော်ပရေးရှင်းတွေ လုပ်ဆောင်လို့ရတာ တွေ့ခဲ့ရတယ်။ Account အော့ဘ်ဂျက်တွေမှာရော ဘယ်လို အော်ပရေးရှင်းတွေ လုပ်ဆောင်လို့ရသင့်လဲ။ အပြင်မှာ အကောင့်တစ်ခုကနေ ငွေထုတ်လို့ရတယ်၊ အကောင့်ထဲကို ငွေသွင်းလို့ရပါတယ်။ Account အော့ဘ်ဂျက်တွေမှာလည်း ဒါတွေလုပ်လို့ရသင့်တာပေါ့

```
acc1.deposit(Decimal("50000.00"));
acc2.withdraw(Decimal("70000.00"));
```

ဒီအော်ပရေးရှင်းတွေက အော့ဘ်ဂျက်တွေအပေါ် ဘယ်လိုသက်ရောက်မှုရှိမလဲ။ ကေသီ့အကောင့် လက်ကျန်ငွေက တစ်သိန်းခွဲ ဖြစ်သွားသင့်တယ်။ စန္ဒီ့အကောင့်က သုံးသိန်းရှစ်သောင်း ဖြစ်သင့်တယ်။ ဒီလို အော်ပရေးရှင်းတွေဟာလည်း ကလပ်စ်ပေါ်မှာ မူတည်တယ်။ ကလပ်စ်က ထောက်ပံ့ပေးထားမှပဲ ရမယ်။ ကလပ်စ်သည်သာ အခရာလို့ ပြောရမှာပါ။

ကလပ်စ်တစ်ခုဟာ အော့ဘ်ဂျက်တွေမှာ ပါရှိရမဲ့ အချက်အလက်တွေနဲ့ ၎င်းအော့ဘ်ဂျက်တွေအပေါ်မှာ လုပ်ဆောင်နိုင်တဲ့ အော်ပရေးရှင်းတွေကို သတ်မှတ်ပေးပါတယ်။ ဒါ့အပြင် ကွန်စရက်တာ (*constructor*) လို့ခေါ်တဲ့ အော့ဘ်ဂျက်တည်ဆောက်တဲ့ သီးသန့်ဖန်ရှင်တစ်ခု ပါရှိရတယ်။ Account အော့ဘ်ဂျက်တွေအတွက် ကလပ်စ်ကို ကြည့်ရအောင် ...

```
# 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 ပြောဖို့အတွက် class Account: နဲ့ စရပါတယ်။ သူ့အောက်မှာရှိတာက Account ကလပ်စ်ရဲ့ ဘော်ဒီ (body) ပါ။ Body ဆိုတာ block ကို ပြောတာပါပဲ။ ကလပ်စ်ဘော်ဒီထဲမှာ ဖန်ရှင် သုံးခု သတ်မှတ်ထားတာ တွေ့ရမယ်။ သုံးခုလုံးမှာ ပထမ ပါရာမီတာက self ဖြစ်နေတာကို သတိပြုမိမှာပါ။ သူ (self) က လက်ရှိအော့ဘ်ဂျက် (current object) လို့ ဆိုလိုတာဖြစ်ပြီး သိပ်မကြာခင် သူ့အဓိပ္ပါယ်ကို ရှင်းပြပါမယ်။

__init__ က ကွန်စရက်တာ (constructor) ဖန်ရှင်ဖြစ်တယ်။ အော့ဘ်ဂျက် တည်ဆောက်ပေးတဲ့ ဖန်ရှင်ပေါ့။ ကွန်စရက်တာနံမည်က အခြားဟာဖြစ်လို့မရဘူး။ __init__ ပဲ ဖြစ်ရပါမယ်။ အော့ဘ်ဂျက်မှာ ပါရှိရမဲ့ ဗေရီရေဘဲလ်တွေကို self.holder, self.acc_number, self.balance လို့ ကွန်စရက်တာထဲမှာ ကြေငြာပေးရပါတယ်။ ဒီနေရာမှာ dot အမှတ်အသား အဓိပ္ပါယ်ကို '၏/ရဲ့' လို့ ယူဆရင် ဖတ်ရတာ အဆင်ပြေတယ်။ self.holder ကို 'လက်ရှိအော့ဘ်ဂျက်ရဲ့ holder'၊ self.balance ကို 'လက်ရှိအော့ဘ်ဂျက်ရဲ့ balance' လို့ ဖတ်နိုင်တယ်။

Account instance တစ်ခုကို ဖန်တီးယူမယ်ဆိုရင် ကွန်စရက်တာကို တိုက်ရိုက်မခေါ်ရပါဘူး။ ကလပ်စ်နံမည်နဲ့ အခုလို ခေါ်ရမှာပါ

```
Account('Amy', '0086-6002-2233', Decimal('350_000.00'))
```

ဒီအခါမှာ __init__ က အလုပ်လုပ်မှာပါ။ ခေါ်တဲ့အခါ self အတွက် တန်ဖိုးထည့်မပေးရပါဘူး (self ဟာ ကလပ်စ်ရေးတဲ့သူအတွက် သီးသန့်ဖြစ်ပြီး ကလပ်စ်ဘော်ဒီထဲမှာပဲ အသုံးပြုရတာပါ)။ holder, acc_number, balance အတွက် တန်ဖိုးတွေကို ထည့်ပေးရတယ်။ ဒီတန်ဖိုးတွေက လက်ရှိအော့ဘ်ဂျက် ဗေရီရေဘဲလ်တွေရဲ့ ကနဦး တန်ဖိုးတွေဖြစ်သွားမှာပါ

```
self.holder = holder
self.acc_number = acc_number
self.balance = balance
```

ဒီသုံးကြောင်းအရ လက်ရှိအော့ဘ်ဂျက်ရဲ့ holder က 'Amy' ဖြစ်ပါမယ်။ လက်ရှိအော့ဘ်ဂျက် balance နဲ့ acc_number က Decimal('350_000.00') နဲ့ '0086-6002-2233' အသီးသီး ဖြစ်ပါမယ်။ အခုလို ပုံဖော်ကြည့်နိုင်ပါတယ်

ဒီအော့ဘ်ဂျက်ကို ရည်ညွှန်းအသုံးပြုနိုင်ဖို့ဆိုရင် ထုံးစံအတိုင်း ဗေရီရေဘဲလ်နဲ့ အဆိုင်းမန့် လုပ်ထားဖို့ လိုပါတယ်။

holder	Amy
accountNumber	0086-6002-2233
balance	350,000.00

(Account)

ပုံ ၉.၁ အေမီ့ Account အော့ဘ်ဂျက် (acc3)

```
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 ရှင်းပြဖို့ လိုလာပြီ။ လက်ရှိအသုံးပြုနေတဲ့ အေမီ့ `Account` အော့ဘ်ဂျက်ဟာ 'လက်ရှိအော့ဘ်ဂျက်' `self` ပဲ ဖြစ်တယ်။ ဒါကြောင့် အခုကိစ္စမှာ `acc3` နဲ့ `self` နှစ်ခုလုံးဟာ 'လက်ရှိအော့ဘ်ဂျက်' (တစ်ခုတည်း) ပဲ။ `acc3.holder` နဲ့ `self.holder` နှစ်ခုလုံးက လက်ရှိအော့ဘ်ဂျက်ရဲ့ `holder` ကို ဆိုလိုတာ ဖြစ်တယ်။ `holder` က `'Amy'` ပါ။ ဒီသဘောအတိုင်း `acc3.balance` ရော `self.balance` ပါ လက်ရှိအော့ဘ်ဂျက်ရဲ့ `balance` ကို ဆိုလိုတာ။ `Decimal("350_000.00")` ဖြစ်ပါမယ်။ လက်ရှိအော့ဘ်ဂျက်က ပြောင်းသွားရင်ရော ဘယ်လိုဖြစ်မလဲ။

```
acc1 = Account('Kathy', '0086-6002-1111', Decimal("100000.00"))
```

အခုလက်ရှိ အော့ဘ်ဂျက်က ကေသီ့ `Account` အော့ဘ်ဂျက် ဖြစ်သွားပြီ။ ဒီအချိန်မှာ `self.holder` က `'Kathy'`။ `acc1.holder` လည်း ဒါပဲဖြစ်မယ်။ `acc_number` နဲ့ `balance` တို့ကိုလည်း အလားတူ စဉ်းစားရမှာ ဖြစ်တယ်။ `'0086-6002-1111'` နဲ့ `Decimal("100000.00")` ဖြစ်ပါမယ်။ အခုလက်ရှိ အော့ဘ်ဂျက်ဟာ သီးခြားတည်ရှိနေတဲ့ အော့ဘ်ဂျက်တစ်ခုဖြစ်ပြီး အခုလို မြင်ကြည့်ရမှာပါ

holder	Kathy
accountNumber	0086-6002-1111
balance	100,000.00

(Account)

ပုံ ၉.၂ ကေသီ့ Account အော့ဘ်ဂျက် (acc1)

ဆိုလိုတာက အေမီ့ `Account` အော့ဘ်ဂျက်မှာ အကောင့်ပိုင်ရှင်၊ နံပါတ်နဲ့ လက်ကျန်ငွေအတွက် သူ့ကိုယ်ပိုင် ဗေရီရေဘဲလ်သုံးခု ရှိနေမှာဖြစ်ပြီး ကေသီ့ `Account` အော့ဘ်ဂျက်ကလည်း သူ့ဟာနဲ့သူ သီးခြားသုံးခု ရှိနေမှာပါ။

အခုလောက်ဆိုရင် self ရဲ့ သဘောကို နားလည်လောက်ပါပြီ။ ကလပ်စ်သတ်မှတ်တဲ့အခါ အော့ဘ်ဂျက်တစ်ခုစီမှာ သီးခြားကိုယ်ပိုင် ပါရှိမဲ့ ဗေရီရေဘဲလ်တွေကို dot အမှတ်အသားအသုံးပြုပြီး self နဲ့ ရည်ညွှန်းရပါတယ်။ deposit နဲ့ withdraw ကို ဆက်ကြည့်ရအောင်။

```
# File: account.py
class Account:
    ... # ဒီမှာ ကွန်စရက်တာ ရှိမယ်၊ ထပ်မပြတဲ့ ချန်ထားခဲ့တယ်
    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။ လက်ရှိအော့ဘ်ဂျက်ရဲ့ အခြေအနေအောက်မှာ အလုပ်လုပ်ပေးမဲ့ ဖန်ရှင်ရဲ့ ပထမ ပါရာမီတာက self ဖြစ်ရပါမယ်။ ဒီဖန်ရှင်တွေက အော့ဘ်ဂျက်အပေါ်မှာ လုပ်ဆောင်လို့ရတဲ့ အော်ပရေးရှင်းတွေပါပဲ။ Account instance တွေအပေါ်မှာ deposit နဲ့ withdraw လုပ်ဆောင်လို့ ရမှာဖြစ်တယ်

```
acc3.withdraw(Decimal('50_000.00'))
acc1.deposit(Decimal('25_000.00'))
```

ပထမတစ်ခုက အေမီ့ အကောင့်ကနေ ငွေထုတ် (withdraw) လုပ်တာပါ။ သိပြီးဖြစ်တဲ့အတိုင်း acc3 နဲ့ self ဟာ လက်ရှိအော့ဘ်ဂျက် ဖြစ်တယ်။ withdraw ကို ကြည့်ရင် amt က လက်ရှိအော့ဘ်ဂျက်ရဲ့ balance ထက် များနေရင် exception raise လုပ်ထားတယ်။ ရှိတဲ့လက်ကျန်ငွေထက် ပိုထုတ်လို့ မရသင့်ဘူး။

```
self.balance -= amt
```

ကတော့ လက်ရှိအော့ဘ်ဂျက်ရဲ့ balance ကနေ amt နှုတ်လိုက်တာပါ။ အေမီ့ အကောင့်မှာ လက်ကျန်ငွေ $300,000.00$ ဖြစ်သွားမယ်။ ဒုတိယတစ်ကြောင်းက ကေသီ့ အကောင့်ကို ငွေသွင်း (deposit) လုပ်တာ။ ဒီတစ်ခါကျတော့ လက်ရှိအော့ဘ်ဂျက်က ကေသီ့အကောင့်ပေါ့။ self.balance += amt က ကေသီ့အကောင့်နဲ့ သက်ဆိုင်တဲ့ အော့ဘ်ဂျက်ရဲ့ balance ကို amt ပမာဏ ပေါင်းပေးတာ။ ဒါကြောင့် ကေသီ့ အကောင့်လက်ကျန်ငွေ $125,000.00$ ဖြစ်သွားပါမယ်။

```
print(acc3.balance)  # 300000.00
print(acc1.balance)  # 125000.00
```

holder	Amy
accountNumber	0086-6002-2233
balance	**300,000.00**

(Account)

(ဂ) အေမီ့ Account အော့ဘ်ဂျက် (`acc3`)

holder	Kathy
accountNumber	0086-6002-1111
balance	**125,000.00**

(Account)

(ခ) ကေသီ့ Account အော့ဘ်ဂျက် (`acc1`)

ပုံ ၉.၃ After `withdraw` and `deposit` on `acc3` and `acc1` respectively

နောက်ဆက်တွဲ က

လိုအပ်သည့် ဆော့ဖ်ဝဲများ ထည့်သွင်းခြင်း

အခြား အင်ဂျင်နီယာ/သိပ္ပံ ပညာရပ်တွေလိုပဲ ပရိုဂရမ်မင်းလေ့လာတဲ့အခါ လက်တွေ့လုပ်ကြည့်ဖို့၊ စမ်းသပ်ကြည့်ဖို့ အင်မတန်အရေးကြီးပါတယ်။ လက်တွေ့ရေးမကြည့်ဘဲ၊ စမ်းသပ်မကြည့်ဘဲ သဘောတရားပိုင်းဆိုင်ရာတွေကို အမှန်တကယ်နားလည်မှာ မဟုတ်ပါဘူး။ အခြားပညာရပ်တွေထက် ကွန်ပျူတာ ပရိုဂရမ်မင်းရဲ့ အားသာချက်တစ်ခုကတော့ လက်တွေ့စမ်းသပ်ခန်းကြီးတွေ ရှိစရာမလိုတာပါပဲ။ စမ်းသပ်စွဲည်းတွေလည်း များများစားစား မလိုအပ်ဘူး။ ကွန်ပျူတာ တစ်လုံးနဲ့ လိုအပ်တဲ့ဆော့ဖ်ဝဲတချို့ ရှိရင်ရပြီ။ ဆော့ဖ်ဝဲတွေကလည်း ပိုက်ဆံကုန်စရာမလိုဘူး။ မပေးဘဲ သုံးလို့ရတာ။

ပရိုဂရမ်လက်တွေ့ရေး လေ့ကျင့်ဖို့အတွက် လိုအပ်တဲ့ ဆော့ဖ်ဝဲတွေ ထည့်ထားရပါမယ်။ Python programming language အတွက် Python ဆော့ဖ်ဝဲ ထည့်ရပါမယ်။ Python ဆော့ဖ်ဝဲမရှိဘဲ Python ကုဒ်တွေ၊ Python ပရိုဂရမ်တွေ run လို့ မရပါဘူး။ Python အပြင် ပရိုဂရမ်ကုဒ် ရေးဖို့အတွက် အထောက်အကူပြု ဆော့ဖ်ဝဲတစ်ခုလည်း လိုအပ်တယ်။

ပရိုဂရမ် ကုဒ်ရေးဖို့အတွက် PyCharm သို့မဟုတ် Visual Studio Code (VS Code) ကို အသုံးပြုနိုင်ပါတယ်။ အက်ဆေးတစ်ပုဒ်ရေးတဲ့အခါ မိုက်ခရိုဆော့ဖ် Word နဲ့ရေးလို့ရသလို ရိုးရိုးရှင်းရှင်း Notepad လောက်နဲ့ ရေးလို့လည်း ရတာပါပဲ။ အက်ဆေးရဲ့ အဓိပ္ပါယ်ကသာ အဓိကပါ။ ပရိုဂရမ်ကုဒ် ရေးတဲ့အခါမှာလည်း ဒီသဘောပါပဲ။ ဆော့ဖ်ဝဲ ပရောဂျက်ကြီးတွေ တည်ဆောက်တဲ့အခါ လိုအပ်မဲ့ ဖီချာတွေ အားလုံး စုံစုံလင်လင်ပါပြီးသား PyCharm လို Integrated Development Environment(IDE) ဆော့ဖ်ဝဲမျိုးနဲ့ ကုဒ်တွေရေးလို့ရသလို ပေါ့ပေါ့ပါးပါးနဲ့ လိုအပ်မှပဲ လိုတဲ့ဖီချာအတွက် extensions (plug-in လို့လည်းခေါ်တယ်) ထည့်သွင်းရတဲ့ Visual Studio Code (VS Code) လို ကုဒ်အယ်ဒီတာ (Code Editor) ဆော့ဖ်ဝဲမျိုး သုံးပြီး ရေးရင်လည်း ရတာပါပဲ။ နောက်ဆုံး ကုဒ်နည်နည်း၊ ဖိုင်နည်းနည်း ရိုးရှင်းတဲ့ ပရိုဂရမ်လေးတွေဆိုရင် Notepad နဲ့ ရေးလို့တောင် ရပါတယ်။ ကုဒ်တွေများမယ်၊ ဖိုင်တွေများမယ်၊ အသင်းအဖွဲ့လိုက် ပူးပေါင်းရေးရတဲ့ ပရိုဂရမ်မျိုးတွေ ဆိုရင်တော့လည်း Notepad လောက်နဲ့ အဆင်မပြေနိုင်တော့ဘူးပေါ့။

PyCharm ရော VS Code ထည့်သွင်းနည်းပါ ဖော်ပြပေးပါမယ်။ မိမိ နှစ်သက်ရာ အဆင်ပြောရာ သို့မဟုတ် နီးစပ်ရာ ပရိုဂရမ်မာ အသိမိတ်ဆွေ အကြံပြုတဲ့ တစ်ခုကို ရွေးချယ်သုံးပါ။ စလေ့လာသူအနေနဲ့ PyCharm ကို အသုံးပြုတာ ပိုအဆင်ပြေမယ်လို့ ထင်တယ်။ PyCharm သုံးကြည့်လို့ မိမိကွန်ပျူတာမှာ နှေးလွန်းတယ်ဆိုရင် VS Code ကိုစမ်းကြည့်ပါ။ နှစ်ခုလုံး စက်အရမ်းကောင်း/မြင့် ဖို့ မလိုပါဘူး။ တော်ရုံ အတန်အသင့်ကောင်းတဲ့ စက်လောက်နဲ့ အဆင်ပြေပါတယ်။

မိမိကိုယ်တိုင်က ကွန်ပျူတာ အသုံးပြုပုံ အခြေခံ အားနည်းပြီး ဖော်ပြပေးထားတဲ့ အတိုင်း တစ်ဆင့်ချင်း အင်စတောလ် လုပ်တာလည်း အဆင်မပြေဖြစ်နေရင် ဒီစာအုပ်ရဲ့ အောက်ပါ ဖေ့စ်ဘွတ်ခ်နဲ့ ယူကျူ

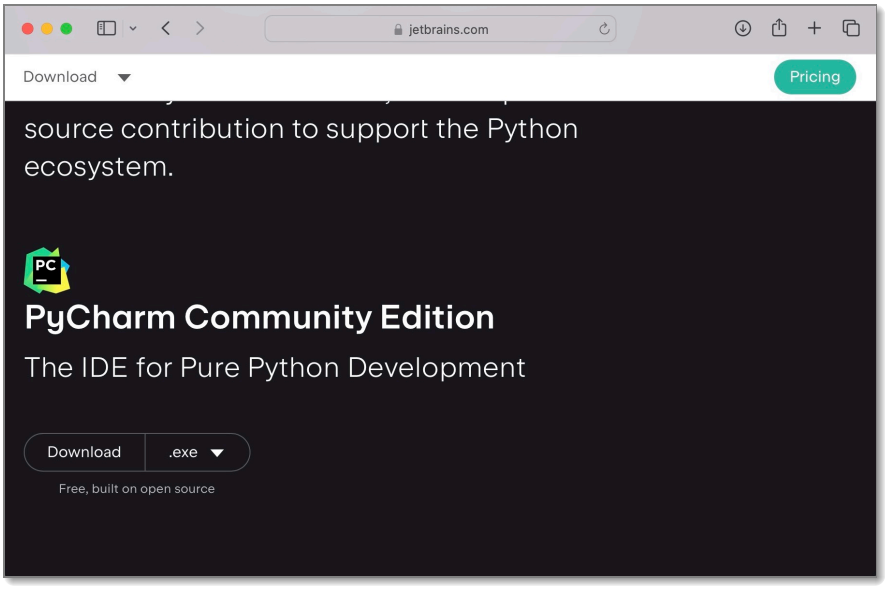
ချန်နယ်တွေမှာ ကြည့်ရှုမေးမြန်း အကူအညီ တောင်းနိုင်ပါတယ်။ ဒါမှမဟုတ် အတွေ့အကြုံရှိတဲ့ နီးစပ်ရာ အသိမိတ်ဆွေ/ညီကိုမောင်နှစ်မ တစ်ယောက်ယောက်ရဲ့ အကူအညီယူပြီး အင်စတောလ်လုပ်ပါ။

https://www.facebook.com/bpwp
https://www.youtube.com/bpwp

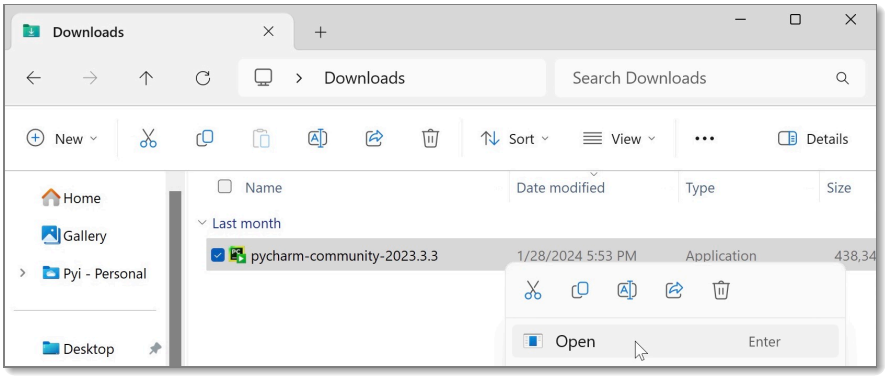
PyCharm အတွက် အရင်ဖော်ပြပေးပါမယ်။ VS Code အတွက် စာမျက်နှာ ၁၈၀ မှာ ကြည့်ပါ။

Python နှင့် PyCharm IDE ထည့်သွင်းခြင်း

https://www.jetbrains.com/pycharm/download/ လင့်ကိုဖွင့်ပါ။ ဝဘ်စာမျက်နှာ အောက်ဘက် နည်းနည်း ဆွဲချလိုက်ရင် PyCharm Community Edition ဒေါင်းလုဒ်ခလုတ်ကို တွေ့ရပါမယ်။ ပုံ (က/၁) ကိုကြည့်ပါ။



ပုံ က/၁

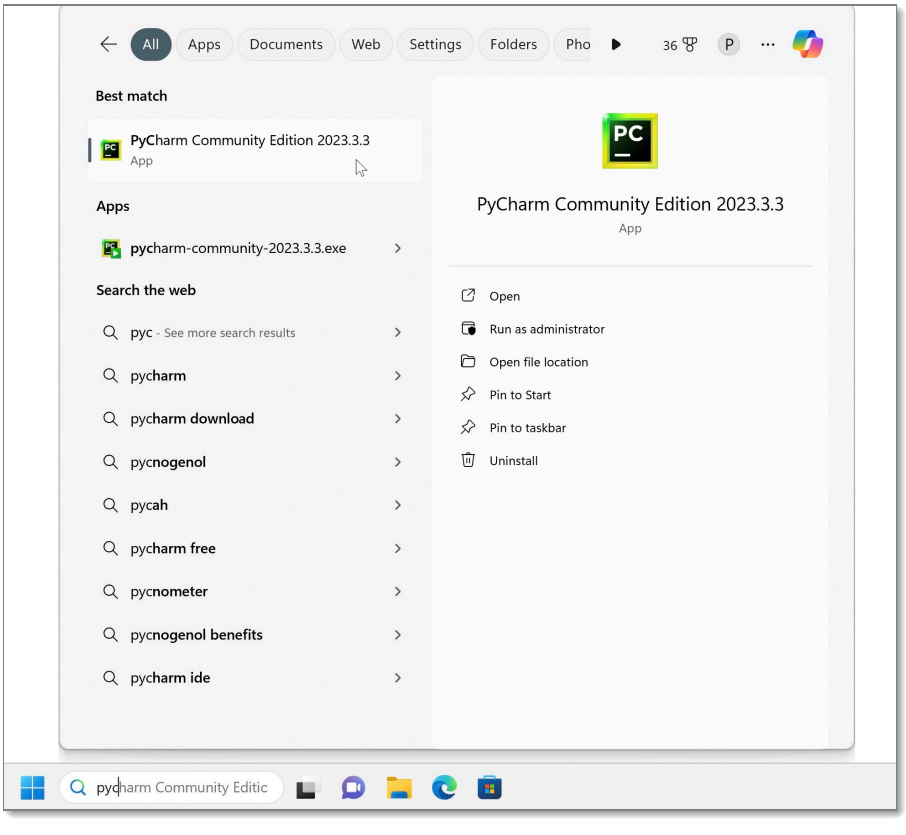


ပုံ က/၂

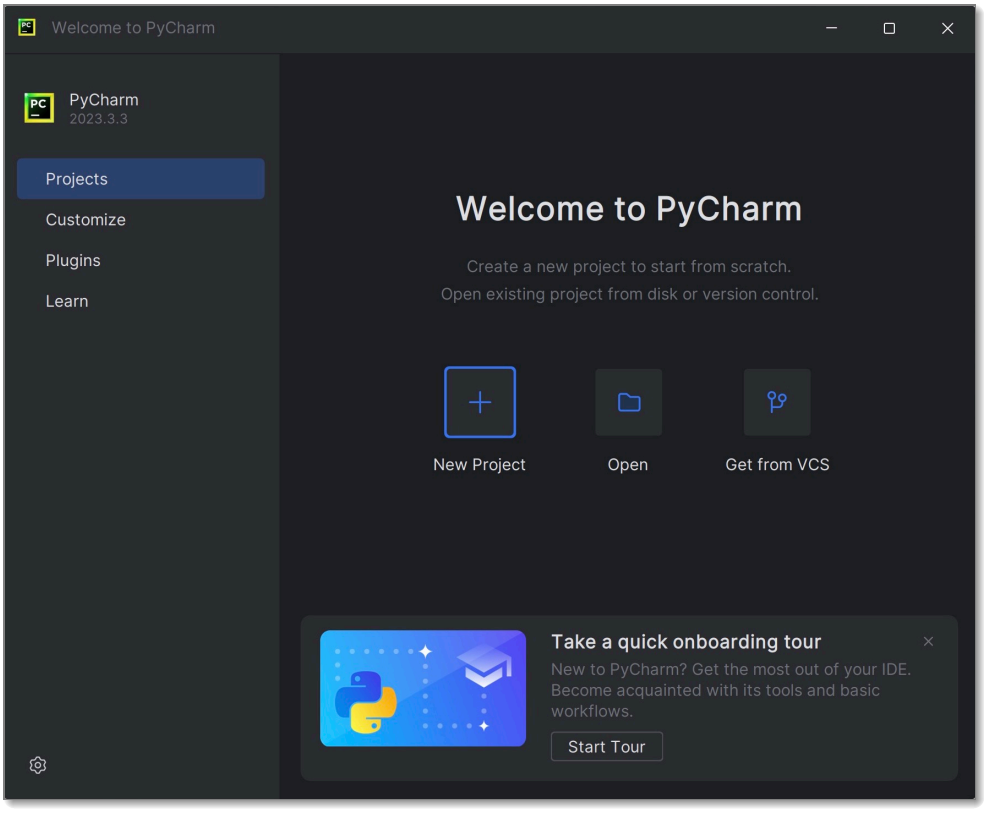
PyCharm Community Edition ကို ဒေါင်းလုဒ်လုပ်ပါ။ (ဝဘ်စာမျက်နှာ အပေါ်ပိုင်းက ဝယ်သုံးရတဲ့ PyCharm Professional ကို ဒေါင်းလုဒ် မှားမလုပ်မိဖို့ သတိပြုပါ)။ အင်စတော်လာဖိုင်ကို ညာကလစ်နှိပ်ပြီး Open လုပ်ပါ (ပုံ ၈/၂ ကိုကြည့်ပါ)။ Yes/No မေးတဲ့အခါ Yes နှိပ်ပါ။ Next > ကို နှိပ်၍ ရှေ့ကိုဆက်သွားပြီး နောက်ဆုံးမှာ Install နှိပ်ပြီး ကွန်ဖန်းလုပ်ပါ။ အင်စတောလ်ပြီးသွားရင် Finish နှိပ်ပါ။

ဝင်းဒိုး Taskbar Search ကနေ PyCharm ကိုရှာပြီးဖွင့်ပါ (ပုံ ၈/၃ ကို ကြည့်ပါ)။ သဘောတူကြောင်း ကွန်းဖန်းလုပ်ခိုင်းရင် ချက်ခ်ဘောက်စ် ချက်ခ်လုပ်ပြီး Continue နှိပ်ပါ။ ဒေတာပို့ချင်လား ထပ်မေးပါလိမ့်မယ်။ Don't Send နှိပ်ပါ။ Welcome စခရင်ကိုပေါ်လာမယ်။ ပုံ (၈/၄) မှာ ကြည့်ပါ။

> ဒီစာရေးနေချိန် လက်ရှိ PyCharm ဗားရှင်းက ၂၀၂၃ ပါ။ သိပ်မကြာခင် ၂၀၂၄ ထွက်ပါတော့မယ်။ အကယ်၍ လက်ရှိဗားရှင်းထက် နိမ့်တဲ့ဗားရှင်းတွေကို ဒေါင်းလုဒ် လုပ်ချင်ရင် အောက်ပါ လင့်ကို သွားပါ။
>
> `https://www.jetbrains.com/pycharm/download/other.html`
> ဗားရှင်း ၂၀၂၄/၂၅ ထွက်ပြီးတဲ့ အချိန်မှာ ၂၀၂၃ ဗားရှင်းကို လိုချင်ရင် ဝဘ်စာမျက်နှာမှ ရှာပြီး ဒေါင်းလုဒ် လုပ်ပါ။ ၂၀၂၃ မှာလည်း ဗားရှင်းအခွဲတွေ ရှိပါသေးတယ်။ လက်ရှိအမြင့်ဆုံး ဗားရှင်းအခွဲ (ဥပမာ ၂၀၂၃.၃.၃) ကို သုံးလို့ရပါတယ်။ ၂၀၂၄/၂၅ သုံးမယ်ဆိုရင်လည်း ပြဿနာတော့ မရှိပါဘူး။ Update ဗားရှင်းဖြစ်တဲ့အတွက် ပုံတွေမှာ ပြထားတာနဲ့တော့ ကွာခြားချက်တချို့ ရှိကောင်းရှိနိုင်ပါတယ်။



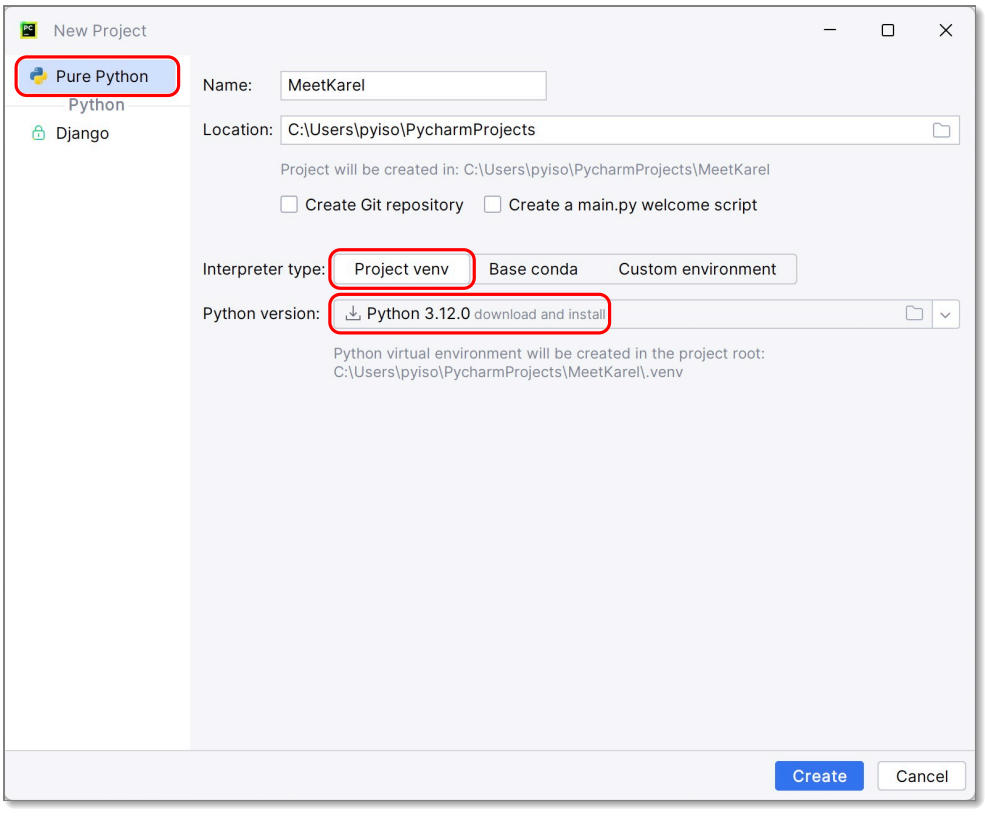
ပုံ ၈/၃



ပုံ ၈/၄

PyCharm IDE ဆိုတာဘာလဲ

PyCharm ဟာ Python နဲ့ ဆော့ဖ်ဝဲ ရေးဖို့အတွက် အထောက်အကူပြု Integrated Development Environment(IDE) ဆော့ဖ်ဝဲဖြစ်ပါတယ်။ စာစီစာရိုက်လုပ်တဲ့အခါ Microsoft Word ကို အသုံးပြုကြသလိုပဲ Python ကုဒ်ရေးဖို့ PyCharm ကိုသုံးတဲ့ သဘောပေါ့။ PyCharm IDE က Python ပရောဂျက်တွေ အတွက် အဓိကရည်ရွယ်တယ်။ အဆောက်အဦးတစ်ခု၊ တံတားတစ်ခု ဆောက်လုပ်တာကို ပရောဂျက်လို့ ပြောလေ့ရှိသလို ပရိုဂရမ်/ဆော့ဖ်ဝဲတစ်ခု တည်ဆောက်တာကိုလည်း ပရောဂျက်လို့ပဲ သုံးနှုန်းပါတယ်။ တစ်ဦးတစ်ယောက်တည်း ရေးတဲ့ ပရိုဂရမ်အသေးလေးတွေ အတွက် PyCharm ကို အသုံးပြုနိုင်သလို ပရိုဂရမ်မာတွေ အဖွဲ့လိုက်နဲ့ တည်ဆောက်ရတဲ့ ပရောဂျက်ကြီးတွေ အတွက်လည်း သုံးပါတယ်။ ဒီစာအုပ်မှာတော့ PyCharm ရဲ့ အဆင့်မြင့်ဖီချာတွေကို အသုံးပြုမှာ မဟုတ်ပါဘူး။ ပရိုဂရမ်းမင်း စလေ့လာသူတွေကို လွယ်ကူအဆင်ပြေစေတဲ့ အခြေခံ ဖီချာတွေလောက်ပဲ အသုံးပြုမှာပါ။



ပုံ ၈/၅ ပရောဂျက် အသစ်ယူခြင်း

PyCharm ပရောဂျက်ဆောက်ခြင်း

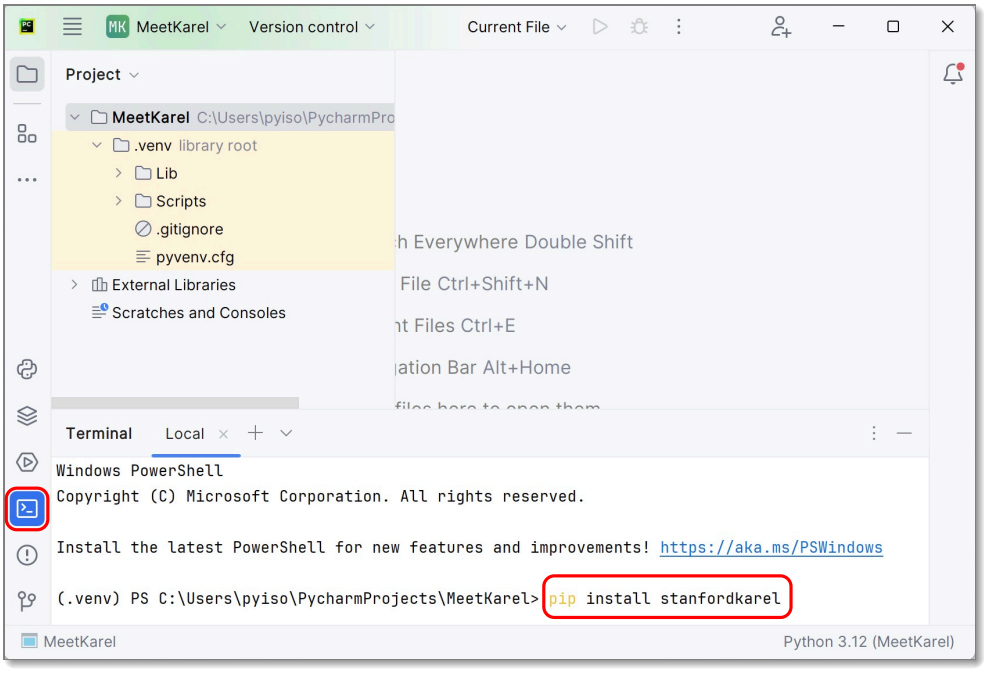
အင်စတောလ်လုပ်ပြီးရင် PyCharm IDE ကိုဖွင့်ပြီး Welcome စခရင်မှ သိုမဟုတ် File မီနူးမှ New Project နှိပ်ပြီး ပရောဂျက် အသစ်ယူပါ (ပြီးခဲ့တဲ့ ပုံ ၈/၄ ကို ပြန်ကြည့်ပါ)။ နံမည်ကို MeetKarel ပေးပါ။ ပုံ (၈/၅) မှာ တွေ့ရတဲ့အတိုင်း Pure Python, Proj venv နဲ့ Python ဗားရှင်း 3.12.xx ကို ရွေးပါ။ xx က အခြားဂဏန်း ဖြစ်နေနိုင်တယ်။ အဓိက ဗားရှင်း 3.12 သာ ဖြစ်ပါစေ။ Create ခလုတ်နှိပ်ပါ။ ပရောဂျက်အတွက် Python ကို အင်စတောလ် လုပ်ပါလိမ့်မယ်။

မိမိလေ့လာနေတဲ့ အခန်းတစ်ခန်းချင်းစီအတွက် ပရောဂျက်တစ်ခု ဆောက်နိုင်ပါတယ်။ အခန်း (၂) အတွက် Chapter02 ၊ အခန်း (၃) အတွက် Chapter03 စသည်ဖြင့်။ သက်ဆိုင်ရာအခန်းအလိုက် ကုဒ်ဖိုင်တွေကို ပရောဂျက်တစ်ခုစီမှာ ထားတဲ့အတွက် ဖိုင်တွေများပြီး ရှုပ်ထွေဖောင်းပွနေတဲ့ ပြဿနာ မရှိတော့ဘူး။ ကုဒ်ဖိုင်တွေကို ပရောဂျက်တစ်ခုထဲမှာပဲ ဖိုဒါအလိုက်ခွဲထားလို့ရပေမဲ့ စလေ့လာသူအနေနဲ့ ပရောဂျက်တစ်ခုချင်း ခွဲထားတာလောက် မလွယ်ကူဘူး။

ပရောဂျက် အသစ်ယူတဲ့အခါ တည်နေရာ Location: ကို သူ့နဂိုအတိုင်း ထားနိုင်သလို မိမိထားချင်တဲ့နေရာကို ညာဘက်စွန်း ဖိုဒါအိုင်ကွန်လေးနှိပ်ပြီး ပြောင်းလို့ရတယ်။

stanfordkarel လိုက်ဘရီ အင်စတောလ်လုပ်ခြင်း

ပုံ (၈/၆) မှာ အနီရောင် ဝိုင်းပြထားတဲ့ အိုင်ကွန်ကို နှိပ်ပြီး Terminal ကိုဖွင့်ပါ။ Terminal မှာ အောက်ပါ ကွန်မန်းဖြင့်



ပုံ ၈/၆ Karel လိုက်ဘရီ အင်စတောလ်လုပ်ခြင်း

```
pip install stanfordkarel
```

ကားရဲလ်လိုက်ဘရီကို အင်စတောလ်လုပ်ပါ။ ပုံ (ဂ/၆) မှာ အနီရောင် ဝိုင်းပြထားပါတယ်။ ခဏကြာတဲ့ အခါ အခုလို မက်ဆေ့ချ်တွေ 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 မက်ဆေ့ချ် တွေ့ရရင် အင်စတောလ်လုပ်တာ အောင်မြင်လို့ပါ။

> မှတ်ချက်။ ။ ပရောဂျက် အသစ်ဆောက်တိုင်း လိုအပ်တဲ့ လိုက်ဘရီကို တစ်ခါထပ်ပြီး အင်စတောလ် လုပ်ရပါမယ်။ ကားရဲလ်အခန်း ပရောဂျက်တစ်ခုစီအတွက် stanfordkarel လိုက်ဘရီကို အထက်ဖော်ပြပါအတိုင်း အင်စတောလ် လုပ်ဖို့လိုတယ်။

နမူနာ ကားရဲလ် ကမ္ဘာနှင့် ပရိုဂရမ်ကုဒ် ဖိုင်များထည့်ခြင်း

meet_karel.zip ဖိုင်ကို ဒီလင့် `http://tinyurl.com/3mmm9c7j` ကနေ ဒေါင်းလုဒ်လုပ်ပါ။ ၎င်း zip ဖိုင်ကို extract လုပ်ပါ။ meet_karel နံမည်နဲ့ ဖိုဒါတစ်ခု ရလာပါမယ်။ ၎င်းဖိုဒါထဲမှ 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 ၈/၇ မှာ မြှားပြထား) ပေါ်မှာ ညာကလစ်နှိပ်ပြီး Paste လုပ်ရမှာပါ။ ကော်ပီကူးထည့်လိုက်တဲ့ ဖိုင်တွေက ပုံမှာ တွေ့ရတဲ့ အတိုင်း MeetKarel ဖိုဒါအောက်မှာ ရှိသင့်ပါတယ်။

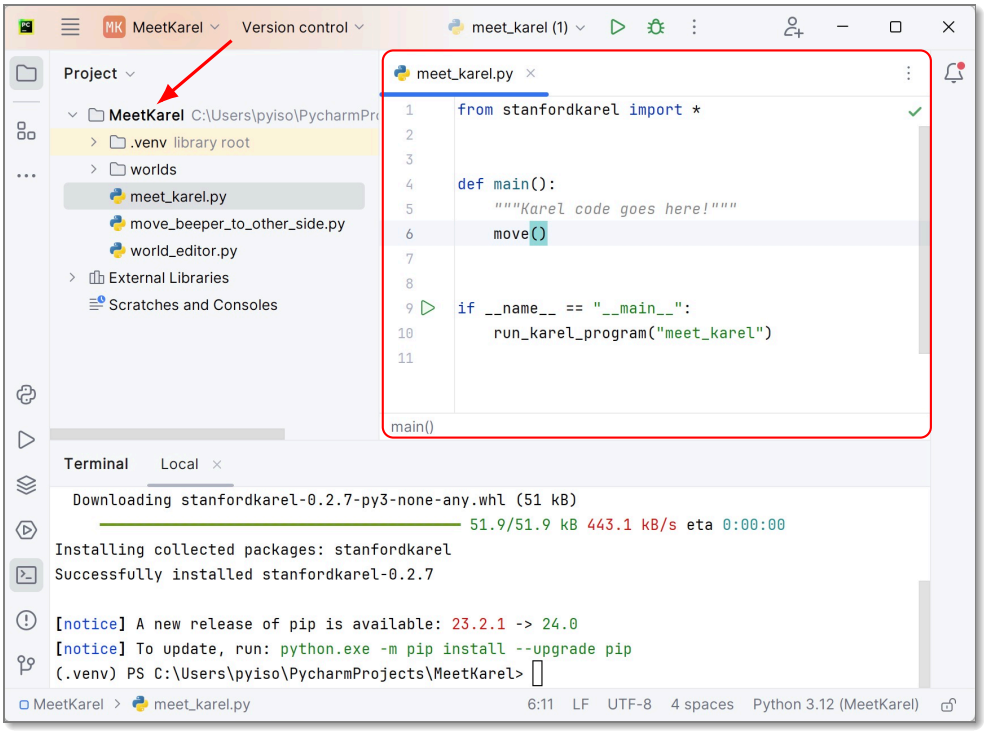
> အခန်းအလိုက် နမူနာ ကုဒ်ဖိုင်တွေ ထည့်ပေးထားတဲ့ .zip ဖိုင်တွေကိုလည်း အထက်ပါအတိုင်း အလားတူ လုပ်ရပါမယ်။ ပရောဂျက်အသစ်ဆောက်၊ .zip ဖိုင်ကို ဖြည်၊ ရလာတဲ့ ဖိုဒါထဲက ဖိုင်တွေကို ပင်မ ပရောဂျက် ဖိုဒါထဲ ကော်ပီကူးထည့် ရုံပါပဲ။

meet_karel.py ဖိုင်ကို ကလစ်နှစ်ချက်နှိပ် ဖွင့်ပါ။ ပုံ (၈/၇) မှာ အနီဝိုင်းထားတဲ့ ကုဒ်အယ်ဒီတာ (code editor) ပွင့်လာပါမယ်။ အဲဒီ မူရင်းဖိုင်မှာ အောက်ပါအတိုင်း ဆက်လက် ပြင်ဆင်ဖြည့်စွက်ပါ။

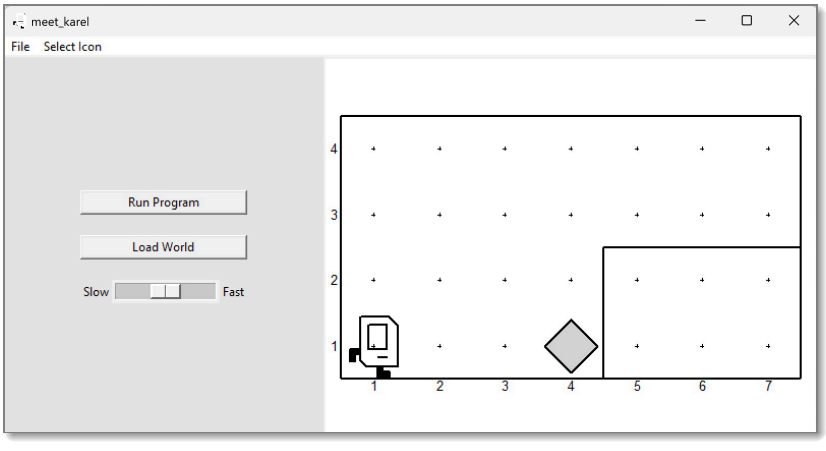
```
from stanfordkarel import *


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_program("meet_karel")
```

> run ထားတဲ့ ပရိုဂရမ်တစ်ခုကို တစ်ခါထပ် run ရင် Cancel (သို့) Stop and Rerun လုပ်မှာလားမေးတယ်။ Stop and Rerun လုပ်ရပါတယ်။

meet_karel.py ဖိုင်ကို ညာကလစ်နှိပ်ပြီး Run 'meet_karel' လုပ်ပါ (အယ်ဒီတာ ဧရိယာမှာ ညာကလစ်နှိပ်ပြီး Run 'meet_karel' လုပ်လို့လည်း ရပါတယ်) ။ ပုံ (က/၈) က ကားရဲလ် ပရိုဂရမ် တက်လာရင် Run Program ခလုတ်ကိုနှိပ်ပါ။ ဘိပါကို ကားရဲလ်က ရွှေ့ပေးပါလိမ့်မယ်။



ပုံ က/၈


meet_karel
File
Select Icon
Run Program
Load World
Slow
Fast
4
3
2
1
1
2
3
4
5
6
7

ပုံ ၈/၈

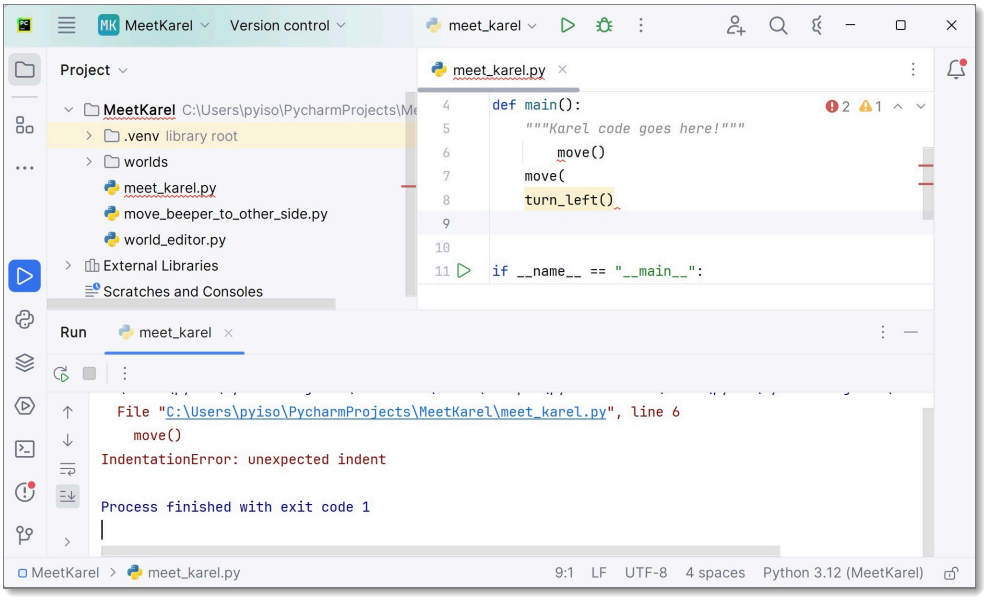
ဆင်းတက်စ်အမှားများ

အကယ်၍ ပရိုဂရမ် run လို့မရရင် ကုဒ်ရေးတာမှားနေလို့ ဖြစ်နိုင်တယ်။ မိမိရေးထားတာကို စာမျက်နှာ (၁၇၁) က ပရိုဂရမ်ကုဒ်နဲ့ နှိုင်းယှဉ် စစ်ဆေးကြည့်ပါ။ PyCharm အယ်ဒီတာမှာ အနီလှိုင့်တွန့်လေးတွေ (ပုံ က/၉) ပြတဲ့နေရင် အဲဒီနေရာတွေမှာ ဆင်းတက်စ်မှားနေလို့ (သို့) လိုက်ဘရီမထည့်ရသေးလို့ပဲ။

ဝိုက်ကွင်းကျန်နေတာက အဖြစ်များတဲ့ အမှားပါ။ ကျန်ခဲ့လို မရပါဘူး။ အင်ဒန့်တေးရှင်း (indentation) လုပ်ရမဲ့နေရာမှာ မလုပ်ထားရင်လည်း ပြဿနာဖြစ်တယ်။ `move`, `turn_left` တွေကို ဘေးမျဉ်းညာဘက်ခွာပြီး အင်ဒန့်လုပ်ပေးရမယ်။ အဲဒါတွေ ဂရုမစိုက်မိရင် ဆင်းတက်စ်အမှားဖြစ်ပြီး ပရိုဂရမ် run လို့ မရနိုင်ဘူး။

Terminal မှာ ထုတ်ပေးတဲ့ မက်ဆေ့ချ်တွေကို ကြည့်ပြီးတော့လည်း ဘာပြဿနာဖြစ်နေလဲ မှန်းဆလို့ရနိုင်တယ်။ ဘာကြောင့်ဖြစ်နိုင်လဲ ဆက်စပ်စဉ်းစားလို့ 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>
```



ပုံ က/၉

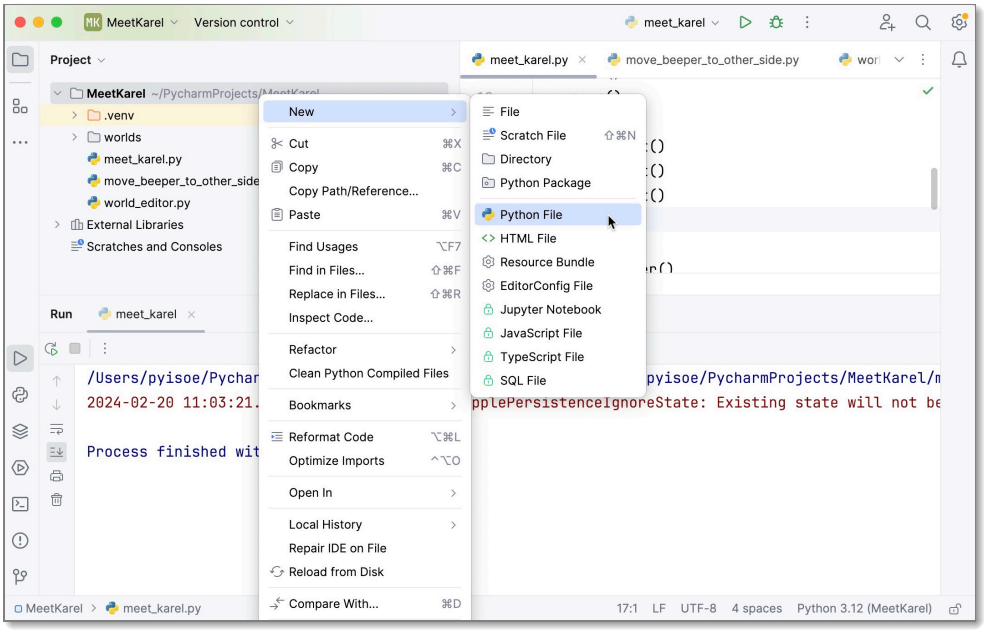
๑๒๗

```
from stanfordkarel import *
ModuleNotFoundError: No module named 'stanfordkarel'
```

Python ဖိုင် အသစ်ယူခြင်း

MeetKarel ပင်မ ပရောဂျက်ဖိုဒါပေါ်မှာ ညာကလစ်နှိပ်ပြီး Python ဖိုင် အသစ်ယူနိုင်ပါတယ်။ Python ဖိုင်တွေက .py အိပ်စ်တန်းရှင်းနဲ့ပါ။ ကားရဲလ်ပရိုဂရမ်တစ်ခုကို Python ဖိုင်တစ်ခု ထားပါမယ်။ ပင်မ ပရောဂျက်ဖိုဒါ အောက်မှာပဲ တိုက်ရိုက်ရှိရပါမယ်။

နောက်ပိုင်း အဆင့်မြင့်လာရင် ပရိုဂရမ်တစ်ခုအတွက် ပရောဂျက်တစ်ခု ထားနိုင်တယ်။ ကုဒ်ဖိုင်တွေ အပြင် ပရိုဂရမ်အတွက် လိုအပ်တဲ့ ရုပ်ပုံတွေ၊ အခြားဖိုင်တွေ (config ဖိုင်၊ setting ဖိုင် စသည်ဖြင့်) လည်း ပါနိုင်တယ်။ ပင်မပရောဂျက် အောက်မှာပဲ ဖိုင်တွေက တိုက်ရိုက်ရှိဖို့လည်း မလိုတော့ဘူး။ ဆက်စပ်ရာ ဖိုင်တွေကို အမျိုးအစားအလိုက်၊ ဖန်ရှင်အလိုက် ဖိုဒါတွေခွဲပြီး စနစ်ကျ စီစဉ်ဖွဲ့စည်း ထားရမှာပါ။ ပရောဂျက်တစ်ခုမှာ ဖိုင်တွေကို စနစ်တကျ စုဖွဲ့ထားဖို့ အရေးကြီးပါတယ်။



ပုံ ၈/၁၀

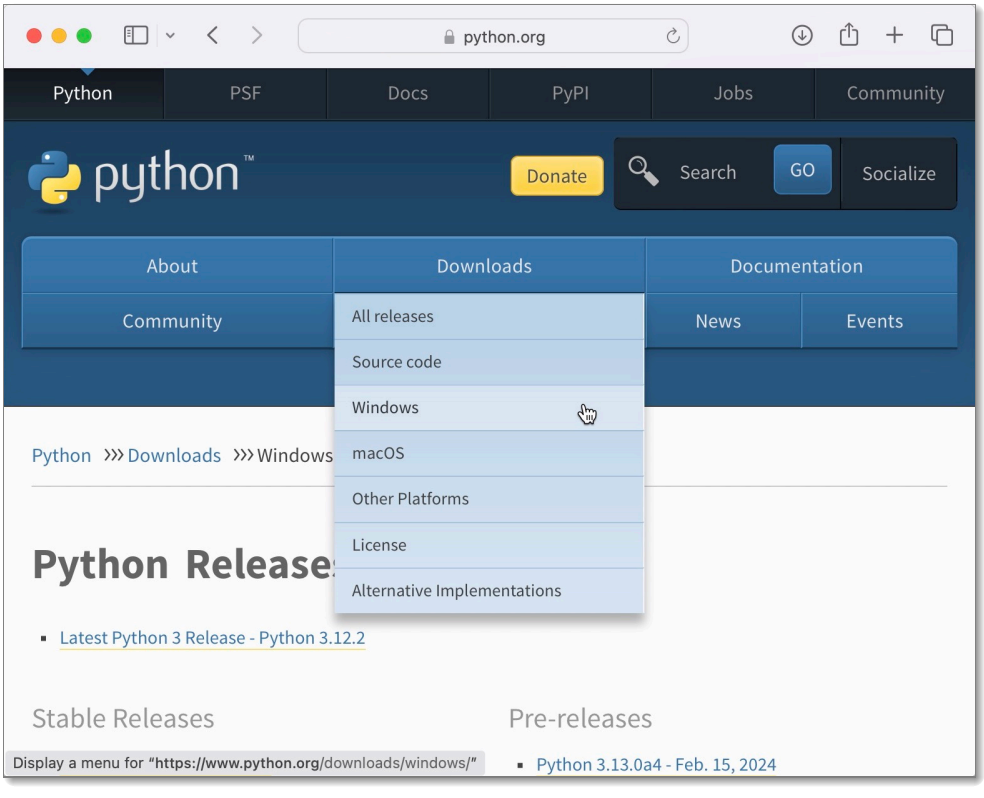
Visual Studio Code နှင့် Python အင်စတောလ်လုပ်ခြင်း

Visual Studio Code(VS Code) ဟာ ပရိုဂရမ်မာအများစု ကြိုက်နှစ်သက်တဲ့ မော်ဒန် ကုဒ်အယ်ဒီတာ တစ်ခုပါ။ Python, JavaScript, C++ စတဲ့ programming language အမျိုးမျိုးအတွက် အသုံးပြုနိုင်ပါတယ်။

PyCharm မှာတော့ ပရောဂျက်အသစ်ဆောက်ရင် Python ပါ တစ်ခါတည်း ဒေါင်းလုဒ်လုပ်ပြီး အင်စတောလ် လုပ်လို့ရတယ်။ VS Code နဲ့ Python ရေးမယ်ဆိုရင် Python Programming Language ကို သီးခြား ဒေါင်းလုဒ်လုပ်ပြီး အင်စတောလ် လုပ်ရပါမယ်။

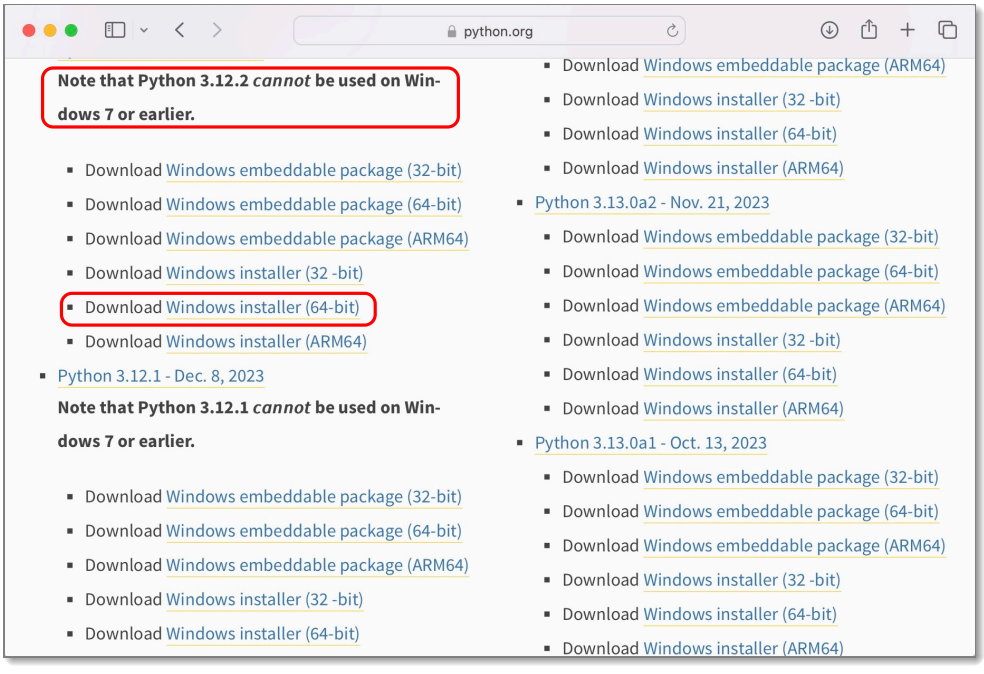
Python အင်စတောလ်လုပ်ခြင်း

ဒီလင့် `https://www.python.org/` ကိုဖွင့်ပါ။ Download မီနူးမှ Windows နှိပ်ပါ (ပုံ ၈/၁၁ ကို ကြည့်ပါ)။ Apple ကွန်ပျူတာအတွက် ဆိုရင် macOS ရွေးရပါမယ်။ လူများစုသုံးတဲ့ မိုက်ခရိုဆော့ဖ် ဝင်းဒိုးအတွက် အင်စတောလ်လုပ်နည်းကို အဓိကပြပေးမှာပါ။ ပုံ ၈/၁၂ မှာလို တွေ့ရပါလိမ့်မယ်။ အင်စတော်လာ ဗားရှင်း 3.12 ထဲက လက်ရှိအမြင့်ဆုံး ကိုရွေးပါ။ ဒီစာရေးချိန်မှာ 3.12.2 ဟာ အမြင့်ဆုံးဗားရှင်းပါ။ Windows Installer (64-bit) ကိုနှိပ်ပြီး ဒေါင်းလုဒ်လုပ်ပါ။



ပုံ ၈/၁၁

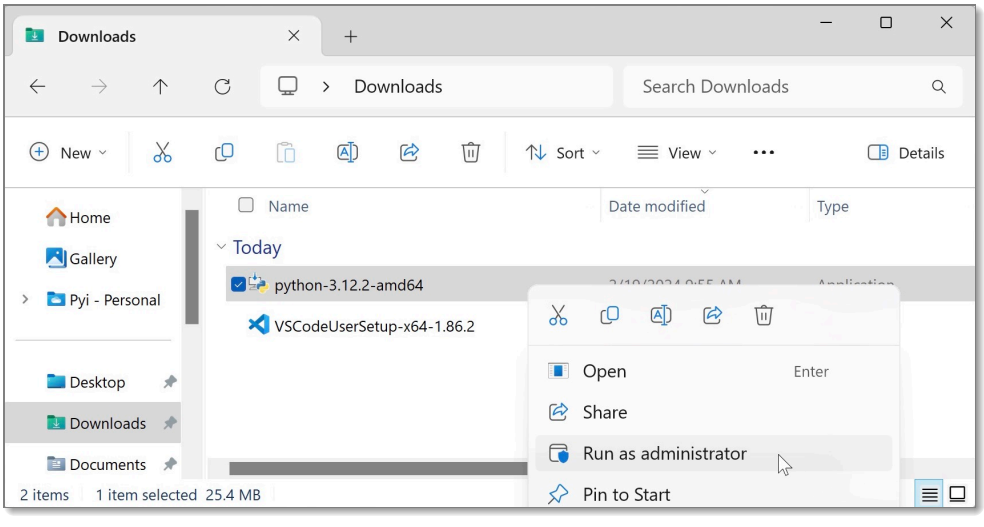
ဒေါင်းလုဒ်ပြီးရင် အင်စတော်လာဖိုင်ကို ညာကလစ်နှိပ်ပြီး Run as administrator လုပ်ပါ။ ပုံ (၈/၁၃) ကိုကြည့်ပါ။ ပုံ (၈/၁၄) မှာလို ဒိုင်ယာလော့ဂ် ဘောက်စ် ပွင့်လာပါမယ်။ အနီဝိုင်းထားတဲ့ ချက်ခ် ဘောက်စ်နှစ်ခုကို ချက်ခ်လုပ်ပြီး Install Now နှိပ်ပါ။ အင်စတောလ် ပြီးလို့ Setup was successful



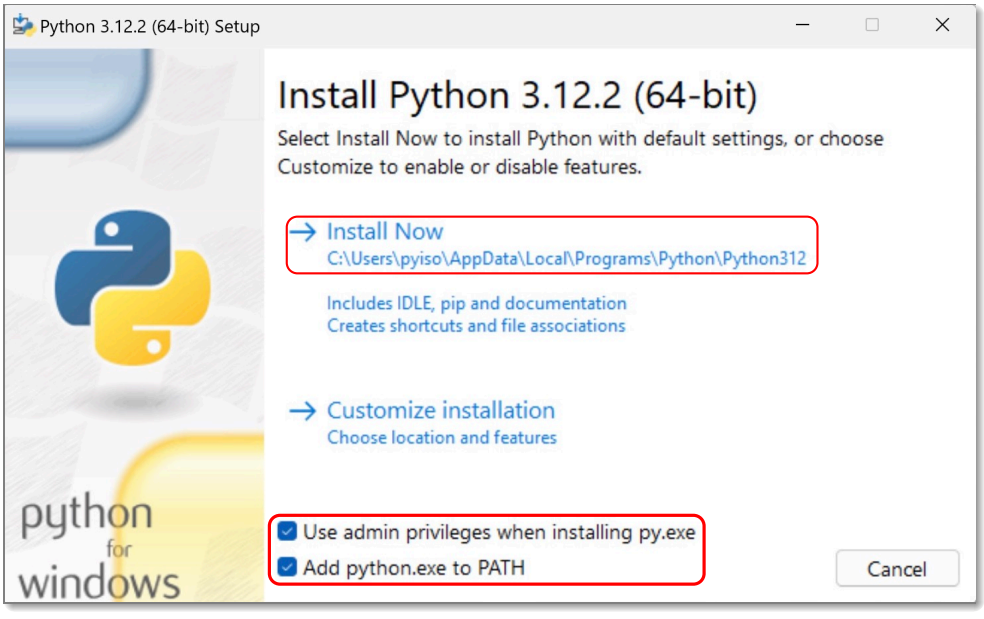
ပုံ ၈/၁၂

ပေါ်လာရင် `Close` ခလုတ်နှိပ်ပြီး ပိတ်ပါ။

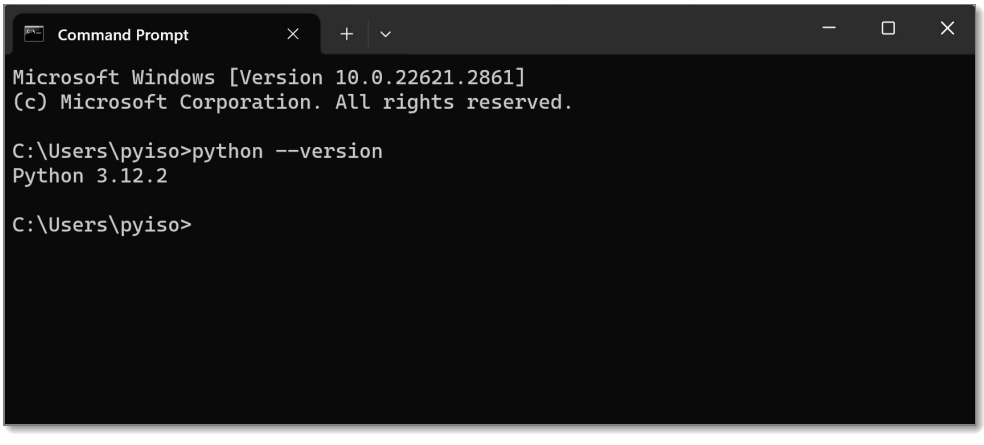
ဝင်းဒိုး command prompt (cmd) မှာ `python --version` run ရင် ပုံ (၈/၁၅) မှာလို အင်စတောလ်လုပ်ထားတဲ့ ဗားရှင်းကို ပြပေးသင့်ပါတယ်။



ပုံ ၈/၁၃


Python 3.12.2 (64-bit) Setup
Install Python 3.12.2 (64-bit)
Select Install Now to install Python with default settings, or choose Customize to enable or disable features.
Install Now
C:\Users\pyiso\AppData\Local\Programs\Python\Python312
Includes IDLE, pip and documentation
Creates shortcuts and file associations
Customize installation
Choose location and features
python for windows
Use admin privileges when installing py.exe
Add python.exe to PATH
Cancel

ပုံ ၈/၁၄


Command Prompt
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>

ပုံ ၈/၁၅

VS Code အင်စတောလ်လုပ်ခြင်း

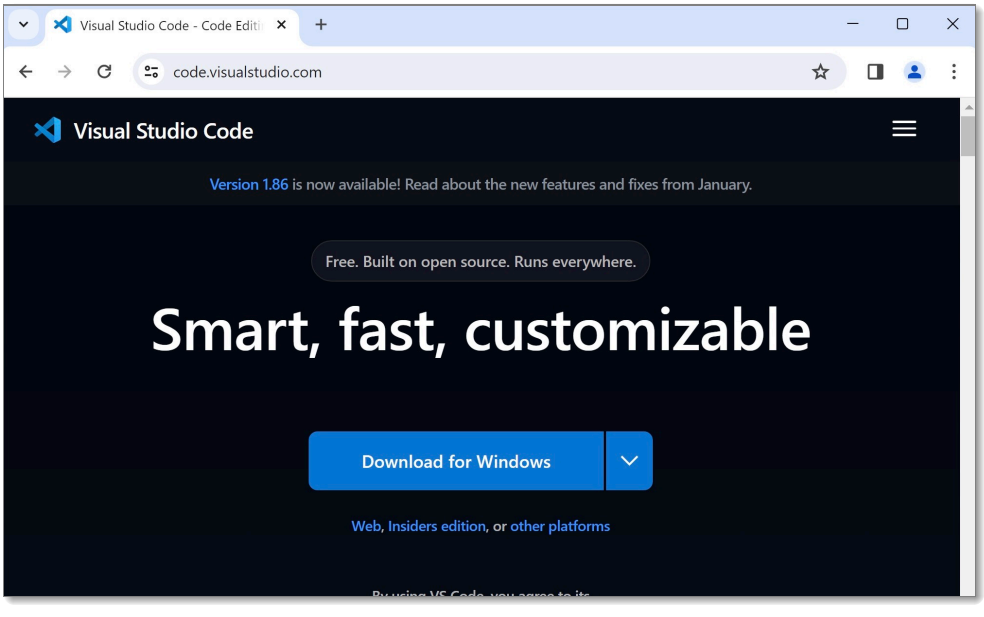
အခုဆိုရင် Python programming language အောင်မြင်စွာ ထည့်ပြီးသွားပါပြီ။ VS Code အင်စတောလ် ဆက်လုပ်ပါမယ်။ အင်စတော်လာ ဒေါင်းလုဒ်လုပ်ရန် ဝဘ်စာမျက်နှာကို အောက်ပါလင့်

`https://code.visualstudio.com`

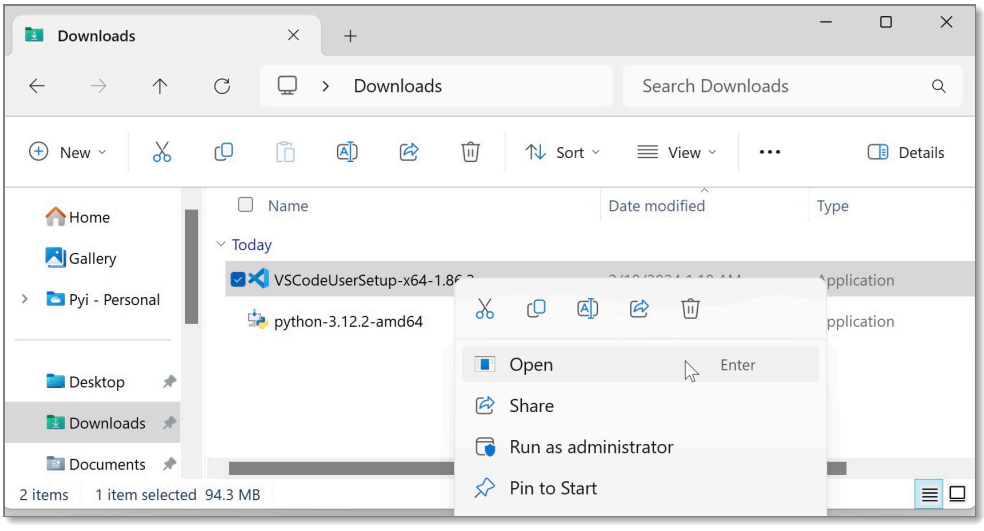
မှတစ်ဆင့် သွားပါ။ ပုံ (က/၁၆) ဝဘ်စာမျက်နှာကို တွေ့ရပါမယ်။ Download for Windows နှိပ်၍ ဒေါင်းလုဒ်လုပ်ပါ။

ပြီးတဲ့အခါ အင်စတော်လာဖိုင်ကို ညာကလစ်နှိပ်ပြီး ဖွင့်ပါ (ပုံ က/၁၇ မှာ ပြထားပါတယ်)။ အင်စတောလ် ဒိုင်ယာလော့ဂ်ဘောက်စ် ပွင့်လာရင် I accept the agreement ကို ချက်ခ်လုပ်၍ Next> တစ်ခုပြီးတစ်ခု ဆက်နှိပ်သွားပြီး နောက်ဆုံးမှာ Install နှိပ်ပါ။ အင်စတောလ်လုပ်နေတာကို ခဏစောင့်ပြီး၊ ပြီးသွားရင် Finish နှိပ်ပါ။

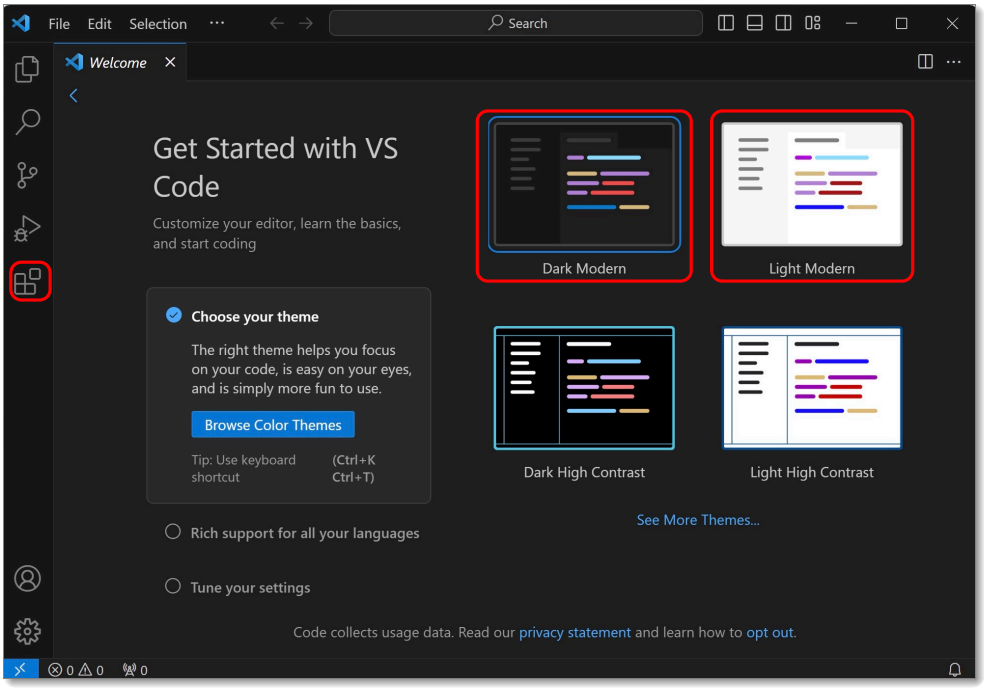
Welcome စခရင်ကို ပုံ (က/၁၈) လို တွေ့ရပါမယ်။ Dark Modern (သို့) Light Modern နှစ်သက်ရာ သီးမ်စ် ရွေးပါ။ ဒါဆိုရင် VS Code လည်း အင်စတောလ် လုပ်ပြီးသွားပါပြီ။ ကားရဲလ်ပ ရိုဂရမ် ရေးဖို့ ဆက်လုပ်ရပါမယ်။



ပုံ က/၁၆


Downloads
Downloads
Search Downloads
New
Sort
View
Details
Home
Gallery
Pyi - Personal
Desktop
Downloads
Documents
Name
Date modified
Type
Today
VSCodeUserSetup-x64-1.86
python-3.12.2-amd64
Open
Enter
Share
Run as administrator
Pin to Start
2 items
1 item selected 94.3 MB

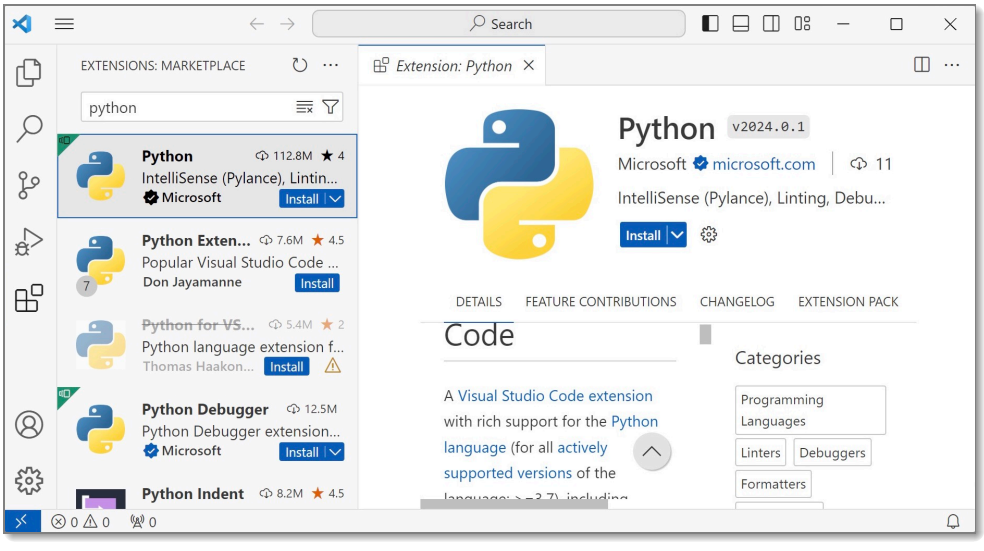
ပုံ ၈/၁၇


File
Edit
Selection
Search
Welcome
Get Started with VS Code
Customize your editor, learn the basics, and start coding
Choose your theme
The right theme helps you focus on your code, is easy on your eyes, and is simply more fun to use.
Browse Color Themes
Tip: Use keyboard shortcut (Ctrl+K Ctrl+T)
Rich support for all your languages
Tune your settings
Dark Modern
Light Modern
Dark High Contrast
Light High Contrast
See More Themes...
Code collects usage data. Read our privacy statement and learn how to opt out.

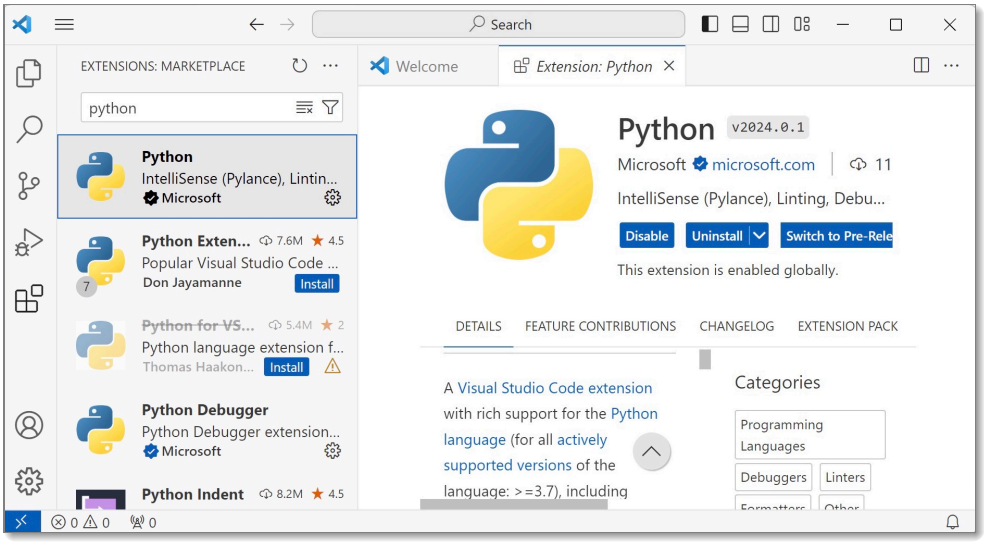
ပုံ ၈/၁၈

VS Code Python Extension ထည့်ခြင်း

Python ပရိုဂရမ်ရေးဖို့အတွက် VS Code က ဒီအတိုင်းဆိုရင် သိပ်အဆင်မပြေသေးပါဘူး။ Python အတွက် extension အင်စတောလ် လုပ်ပေးရပါအုံးမယ်။ ပုံ (က/၁၈) ဘယ်ဘက်ဘောင်နားမှာ အနီ ဝိုင်းထားတဲ့ အိုင်ကွန်လေးကို နှိပ်ပါ။ Python extension ကိုရှာပါ။ ပုံ (က/၁၉) မှာ ဘယ်လိုရှာရ မလဲ ပြထားပါတယ်။ ပုံမှာတွေ့ရတဲ့ Microsoft က ထုတ်တဲ့ extension ကို အင်စတောလ်လုပ်ပါ။ အောင်မြင်ရင် ပုံ (က/၂၀) မှာလို ဖြစ်သွားပါမယ်။ Python နဲ့ VS Code ကိစ္စတော့ ပြီးသွားပြီ။ ကား ရဲလ် example run ဖို့ ဆက်လုပ်ရမယ်။



ပုံ က/၁၉

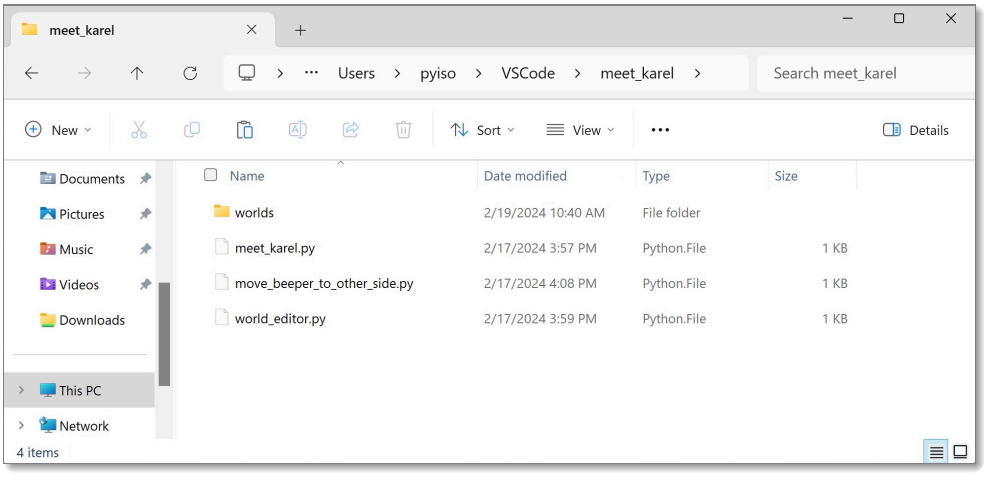


ပုံ က/၂၀

နမူနာ ကားရဲလ် ကမ္ဘာနှင့် ပရိုဂရမ်ကုဒ် ဖိုင်များထည့်ခြင်း

meet_karel.zip ဖိုင်ကို ဒီလင့် `http://tinyurl.com/3mmm9c7j` ကနေ ဒေါင်းလုဒ်လုပ်ပါ။ ၎င်း 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 ၈/၂၁

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` ရှော့ကတ် နှိပ်ပြီး `%userprofile%` ရိုက်ထည့်၊ Ok လုပ်ပြီး Home ဖိုဒါကို သွားနိုင်ပါတယ်။ အကယ်၍ မသွားတတ်ရင်လည်း ပြဿနာမရှိပါဘူး။ ကိုယ့်အတွက် လွယ်ကူမဲ့ Desktop, Downloads, Documents တစ်ခုခုထဲမှာ VS Code အတွက် ဖိုဒါတစ်ခု ထားလည်းရတယ်။

`meet_karel` ဖိုဒါ (.zip ဖိုင်မဟုတ်ပါ) ကို အထက်ပါအတိုင်း အသစ် ဆောက်ထားတဲ့ VS Code သီးသန့်ဖိုဒါထဲကို ကော်ပီကူးထည့်ပါ။ ၎င်း `meet_karel` ဖိုဒါကို VS Code `File` မီနူးမှ `Open Folder` နှိပ်၍ ဖွင့်ပါ။ ပုံ (က/၂၂) တွင်ကြည့်ပါ။

အခန်းအလိုက် နမူနာ ကုဒ်ဖိုင်တွေ ထည့်ပေးထားတဲ့ .zip ဖိုင်တွေကိုလည်း အထက်ပါအတိုင်း လုပ်ရပါမယ်။ .zip ဖိုင်ကို ဖြည်၊ ရလာတဲ့ ဖိုဒါကို သီးသန့်ဖိုဒါတစ်ခုထဲကို ကော်ပီကူးထည့်၊ VS Code နဲ့ အဲ့ဒီဖိုဒါကို ဖွင့်ရုံပါပဲ။

File
Edit
Selection
Search
New Text File Ctrl+N
New File... Ctrl+Alt+Windows+N
New Window Ctrl+Shift+N
Open File... Ctrl+O
Open Folder... Ctrl+K Ctrl+O
Open Workspace from File...
Open Recent
Add Folder to Workspace...
Save Workspace As...
Duplicate Workspace
Save Ctrl+S
Save As... Ctrl+Shift+S
Save All Ctrl+K S
Share
Auto Save
Preferences
Revert File
Close Editor Ctrl+F4
Dark Modern
Light Modern
Dark High Contrast
Light High Contrast
See More Themes...
ts usage data. Read our privacy statement and learn how to opt out.

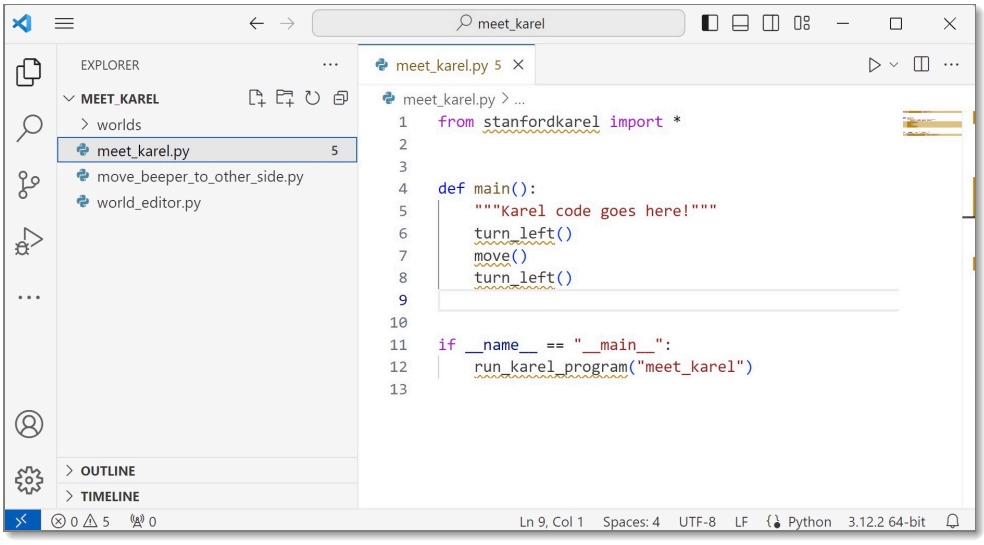
ပုံ ၈/၂၂

Search
EXPLORER
Welcome
Extension: Python
NO FOLDER OPENED
You have no
Opening a fo
currently op
open, add a
Open Folder
Pyi Soe › VSCode ›
Search VSCode
Organize
New folder
Home
Pyi - Personal
Desktop
Name
Date modified
Type
meet_karel
2/19/2024 10:40 AM
File folder
Folder: meet_karel
Select Folder
Cancel
OUTLINE
TIMELINE
language: >=3.7), including

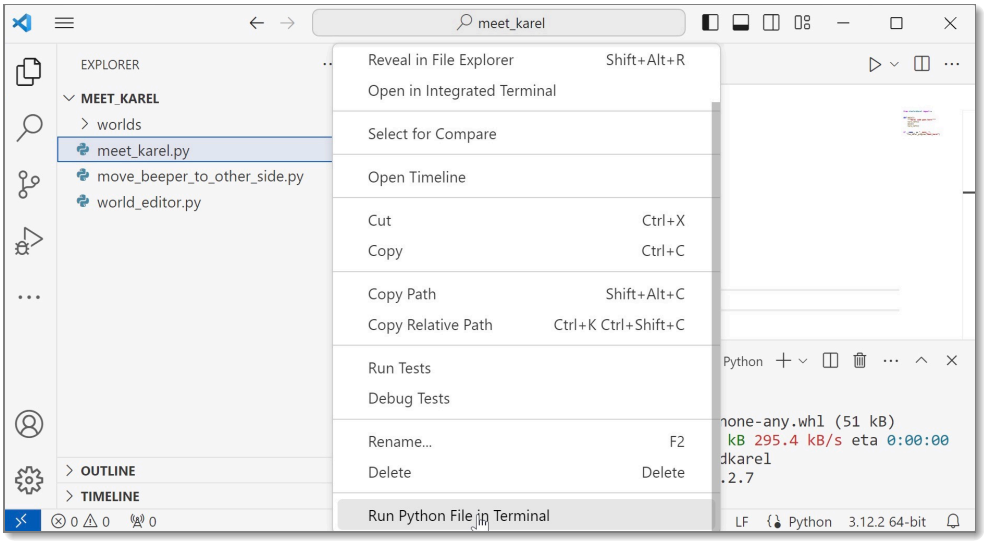
ပုံ ၈/၂၃

stanfordkarel လိုက်ဘရီ အင်စတောလ်လုပ်ခြင်း

meet_karel.py ဖိုင်ကို ကလစ်နှစ်ချက်နှိပ် ဖွင့်ပါ။ ကုဒ်အယ်ဒီတာ ပွင့်လာမယ် (ပုံ ၈/၂၄)။ အဲဒီ ကုဒ်အယ်ဒီတာပေါ် (သို့) meet_karel.py ဖိုင်ကို ညာကလစ်နှိပ်ပြီး Run Python File in Terminal လုပ်ပါ (ပုံ ၈/၂၅)။ Terminal ပွင့်လာပြီး အယ်ရာမက်ဆေ့ချ်တွေ ပြလိမ့်မယ်။ ပုံ (၈/၂၆) မှာကြည့်ပါ။ ကားရဲလ်ပရိုဂရမ်အတွက် လိုအပ်တဲ့ stanfordkarel လိုက်ဘရီ အင်စတောလ် မလုပ်ရသေးပါဘူး။ ဒါကြောင့် အယ်ရာဖြစ်နေတာ။

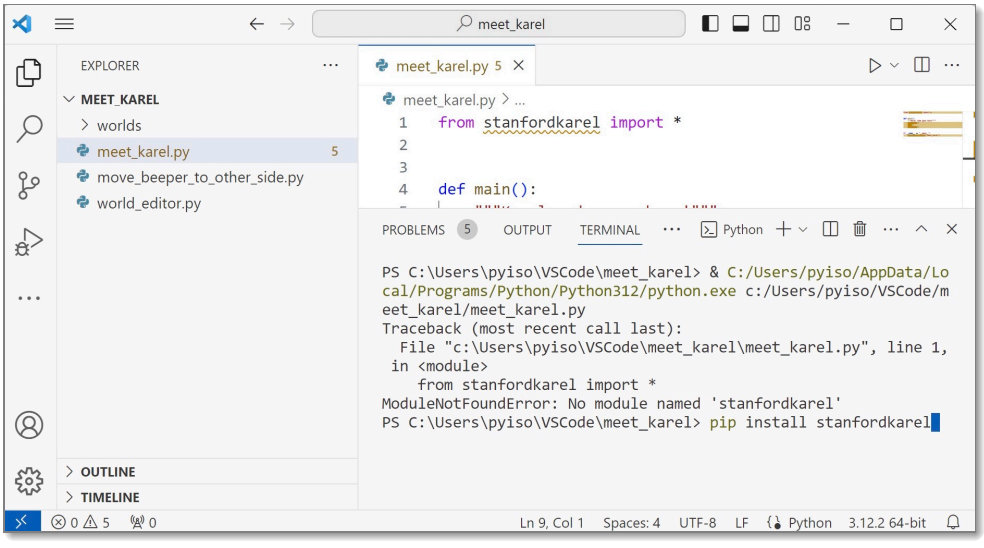


ပုံ ၈/၂၄



ပုံ ၈/၂၅

ခုနကပွင့်လာတဲ့ Terminal မှာပဲ အောက်ပါကွန်မန်းကို run ပြီး stanfordkarel လိုက်ဘရီကို အင်စတောလ်လုပ်ပါ။



ပုံ ၈/၂၆

```
pip install stanfordkarel
```

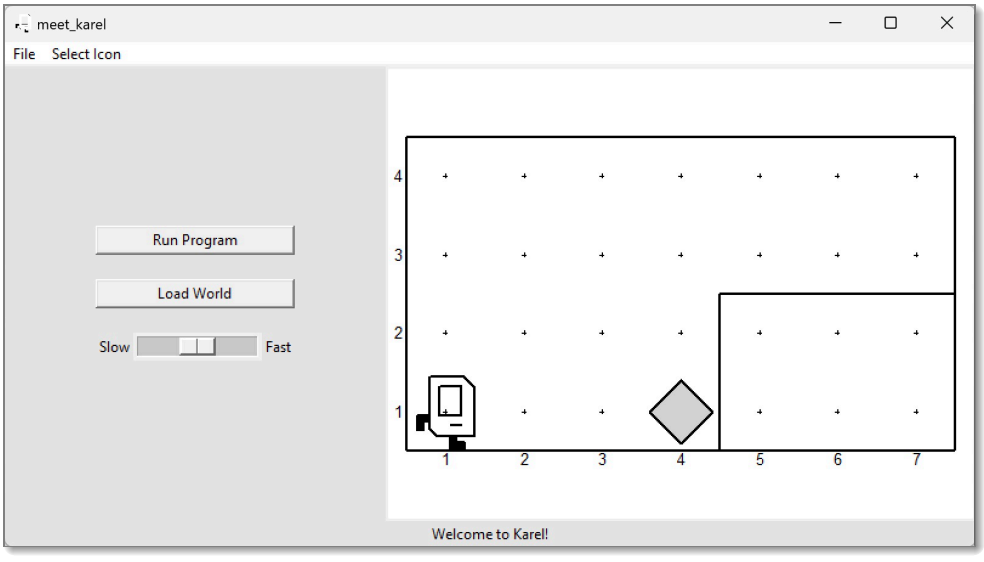
ပုံ (၈/၂၆) မှာ အနီဝိုင်းထားတာကို ကြည့်ပါ။ အဲဒီအတိုင်းရိုက်ထည့်ပြီး Enter ကီးနှိပ်ပါ။ ခဏကြာတဲ့ အခါ အခုလို မက်ဆေ့ချ်တွေ ကျလာပါလိမ့်မယ်။

```
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>
```

ဟိုက်လိုက်ပြထားတဲ့ မက်ဆေ့ချ် တွေ့ရရင် အင်စတောလ် အောင်မြင်လို့ပါ။ ပုံ (၈/၂၄) အယ်ဒီတာမှာလို သတိပေး လှိုင်းတွန့်မျဉ်းတွေ မရှိသင့်တော့ဘူး။ stanfordkarel လိုက်ဘရီ အင်စတောလ် လုပ်ပြီးပြီမို့လို့ သတိပေးတာတွေ ပျောက်သွားသင့်တယ်။

meet_karel.py ဖိုင်ကို ညာကလစ်နှိပ်ပြီး Run Python File in Terminal ပြန်လုပ်ကြည့်ပါ။ ပုံ (၈/၂၇) မှာပြထားတဲ့ ကားရဲလ်ပရိုဂရမ် ပွင့်လာသင့်ပါတယ်။ Run Program နှိပ်ကြည့်ပါ။ စက်ရုပ်လေး ကားရဲလ် နေရာရွှေ့သွားတာ တွေ့ရမယ်။ meet_karel.py ကို အောက်ပါအတိုင်း ဖြည့်စွက်ရေးပါ။

```
from stanfordkarel import *


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



ပုံ ၈/၂၃

```
    move()
    pick_beeper()
    turn_left()
    move()
    move()

    turn_left()
    turn_left()
    turn_left()

    move()
    put_beeper()


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

`meet_karel.py` ဖိုင်ကို ညာကလစ်နှိပ်ပြီး `Run Python File in Terminal` ပြန်လုပ်ကြည့်ပါ။ ကားရဲလ်ပရိုဂရမ် ပွင့်လာရင် `Run Program` နှိပ်ကြည့်ပါ။ ဘိပါလေးကို နေရာရွှေ့ပေးပါလိမ့်မယ်။ အကယ်၍ ကားရဲလ်ပရိုဂရမ် မတက်လာရင် ကုဒ်ရေးတာမှားနေလို့ ဖြစ်နိုင်တယ်။ အပေါ်ကုဒ်နဲ့ နှိုင်းယှဉ်ပြီး ကြည့်ပါ။ VS Code အယ်ဒီတာမှာ အနီလှိုင့်တွန့်လေးတွေ ပြတဲ့နေရင် အဲဒီနေရာတွေမှာ ဆင်းတက်စ်မှားနေတာ ဖြစ်နိုင်တယ်။

VS Code အယ်ဒီတာမှာ ပရိုဂရမ်ကုဒ်ပြင်ပြီး ပြန် run တဲ့အခါ ပထမ run ထားတဲ့ ပရိုဂရမ်ကို အရင်ပိတ်ဖို့လိုပါတယ်။ ဆိုလိုတာက meet_karel.py ကို run ထားတယ်ဆိုပါစို့။ ပုံ (က/၂၇) က ဝင်းဒိုးပွင့်လာမယ်။ meet_karel.py ကုဒ်ကို ပြင်ပြီး ပြန် run ချင်ရင် အဲဒီ ဝင်းဒိုးကို အရင်ပိတ်ရမယ်။ မဟုတ်ရင် ပြင်ထားတဲ့ ပရိုဂရမ်က ချက်ချင်း ပွင့်မလာဘူး။ ပထမ ဟာကို ပိတ်တော့မှပဲ နောက် run တဲ့ဟာ ပွင့်လာမှာ။

ဝိုက်ကွင်းကျန်နေတာက အဖြစ်များတဲ့ အမှားပါ။ ကျန်ခဲ့လို မရပါဘူး။ အင်ဒန့်တေးရှင်း (indentation) လုပ်ရမဲ့နေရာမှာ မလုပ်ထားရင်လည်း ပြဿနာဖြစ်တယ်။ `move`, `turn_left` တွေကို ဘေးမျဉ်းညာဘက်ခွာပြီး အင်ဒန့်လုပ်ပေးရမယ်။ အဲဒါတွေ ဂရုမစိုက်မိရင် ဆင်းတက်စ်အမှားဖြစ်ပြီး ပရိုဂရမ် run လို့ မရနိုင်ဘူး။

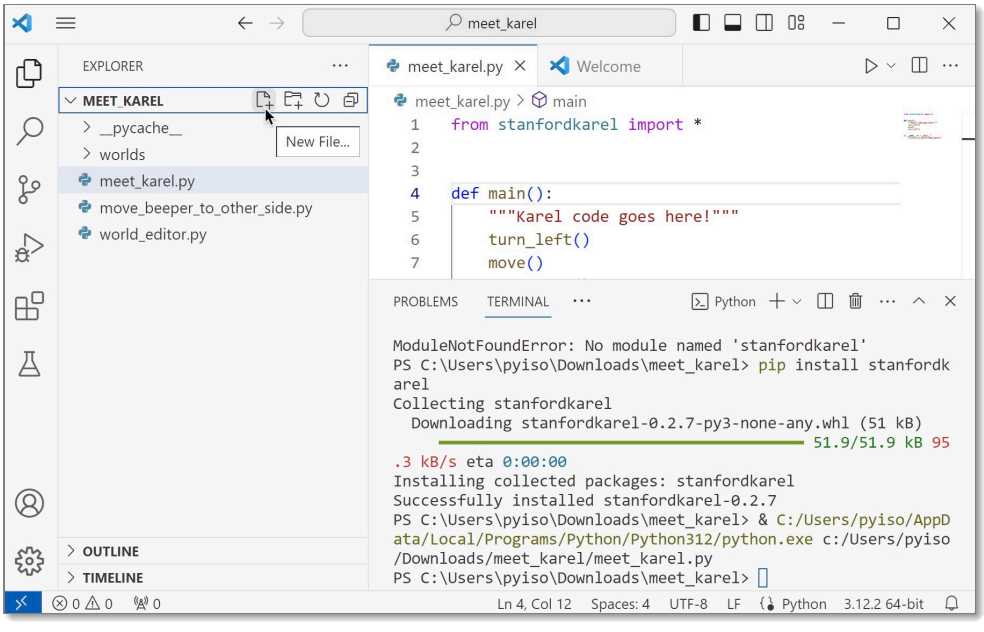
Terminal မှာ ထုတ်ပေးတဲ့ မက်ဆေ့ချ်တွေကို ကြည့်ပြီးတော့လည်း ဘာပြဿနာဖြစ်နေလဲ မှန်းဆလို့ရနိုင်တယ်။ ဘာကြောင့်ဖြစ်နိုင်လဲ ဆက်စပ်စဉ်းစားလို့ ရတယ်။ ဥပမာ ဖြစ်တဲ့ပြဿနာအလိုက် အခုလို တွေ့ရပါမယ်။

```
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 ပရောဂျက်ဖိုဒါပေါ်မှာ ကလစ်နှိပ်ပါ။ ပုံမှာ ပြထားတဲ့ New File အိုင်ကွန်ကိုနှိပ်ပါ။ ဖိုင်နံမည်ဖြည့်တဲ့ ဘောက်စ်လေး ပေါ်လာမယ်။ Python ဖိုင်တွေက .py အိပ်စ်တန်းရှင်းနဲ့ ဖြစ်ရပါမယ်။ ဒါကြောင့် နံမည် ဖြည့်တဲ့အခါ .py နဲ့ အဆုံးသတ်ပေးရပါမယ် (ဥပမာ hello.py)။ ကားရဲလ်ပရိုဂရမ်တစ်ခုကို Python ဖိုင်တစ်ခု ထားပါမယ်။ ပင်မ ပရောဂျက်ဖိုဒါ အောက်မှာပဲ တိုက်ရိုက်ရှိရပါမယ်။

နောက်ပိုင်း အဆင့်မြင့်လာရင် ပရိုဂရမ်တစ်ခုအတွက် ပရောဂျက်တစ်ခု ထားနိုင်တယ်။ ကုဒ်ဖိုင်တွေ အပြင် ပရိုဂရမ်အတွက် လိုအပ်တဲ့ ရုပ်ပုံတွေ၊ အခြားဖိုင်တွေ (config ဖိုင်၊ setting ဖိုင် စသည်ဖြင့်) လည်း ပါနိုင်တယ်။ ပင်မပရောဂျက် အောက်မှာပဲ ဖိုင်တွေက တိုက်ရိုက်ရှိဖို့လည်း မလိုတော့ဘူး။ ဆက်စပ်ရာ ဖိုင်တွေကို အမျိုးအစားအလိုက်၊ ဖန်ရှင်အလိုက် ဖိုဒါတွေခွဲပြီး စနစ်ကျ စီစဉ်ဖွဲ့စည်း ထားရမှာပါ။ ပရောဂျက်တစ်ခုမှာ ဖိုင်တွေကို စနစ်တကျ စုဖွဲ့ထားဖို့ အရေးကြီးပါတယ်။

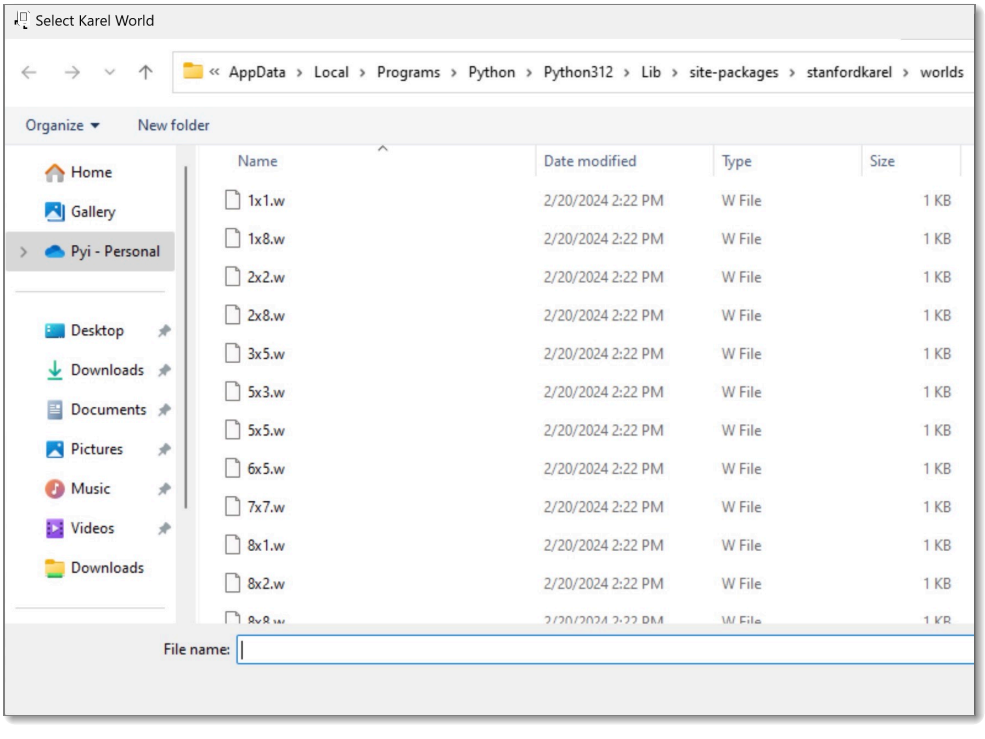


နောက်ဆက်တွဲ ခ

ကားရဲလ်ပရိုဂရမ် ဖီချာများ

ကားရဲလ် ကမ္ဘာဖိုင်များ

ကားရဲလ်ပရိုဂရမ် ဝင်းဒိုးမှာ Load World ခလုတ်နှိပ်ပြီး ကမ္ဘာဖိုင်အသစ် တင်လို့ရတယ်။ ခလုတ်နှိပ်လိုက်ရင် အခုလို ဖိုင် ဒိုင်ယာလော့ဂ် ပွင့်လာမယ်။



ပုံ ခ/၁

ဒါက stanfordkarel လိုက်ဘရီ သူ့နဂိုအရှိအတိုင်း ပါတဲ့ worlds ဖိုဒါပါ။ ဖိုင်တွေက .w နဲ့ ဆုံးပါတယ်။ ကမ္ဘာဖိုင်တွေကို ပင်မ ပရောဂျက်အောက် worlds ဖိုဒါထဲမှာ ထားလို့လည်းရတယ်။ အခြားနေရာတွေမှာ ထားလို့တော့ မရဘူး။

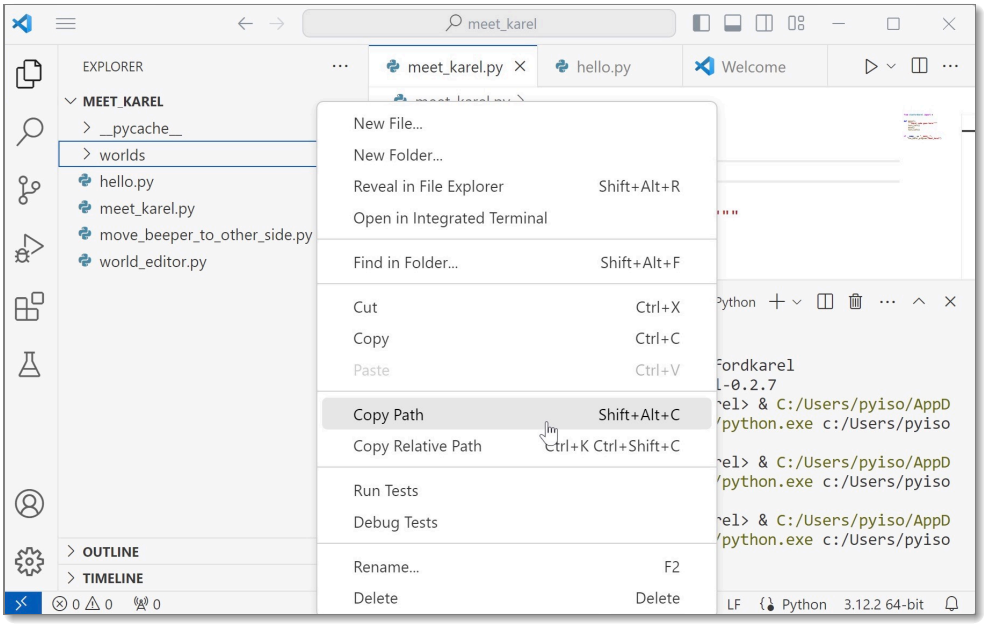
စာအုပ်ပါ ဥပမာတွေ၊ လေ့ကျင့်ခန်းတွေအတွက် ကမ္ဘာဖိုင်တွေကို ပရောဂျက် တစ်ခုချင်းအလိုက် သီးခြား worlds ဖိုဒါထဲမှာ ထည့်ပေးထားမှာပါ။ ပရိုဂရမ်တစ်ခုဟာ ကမ္ဘာတစ်ခုတည်းမှာပဲ အလုပ်လုပ်တာမဟုတ်ဘဲ အလားတူ ကမ္ဘာအမျိုးမျိုးအတွက် အလုပ်လုပ်အောင် ရေးပေးရတာ။ အခုပြောတာကို နားမလည်ရင် အခန်း (၂) ဖတ်ပြီးရင် နားလည်သွားမှာပါ။

Load World လုပ်တဲ့အခါ ပွင့်လာတဲ့ ဖိုင် ဒိုင်ယာလော့ဂ်က ကိုယ်လိုချင်တဲ့ worlds ဖိုဒါ မဖြစ်နေဘူး။ သူ့နဂိုပါတဲ့ worlds ဖိုဒါ ဖြစ်နေတယ်။ ကိုယ်ခေါ်တင်ချင်တဲ့ ဖိုင်တွေရှိတဲ့ လက်ရှိပရောဂျက်ရဲ့ 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

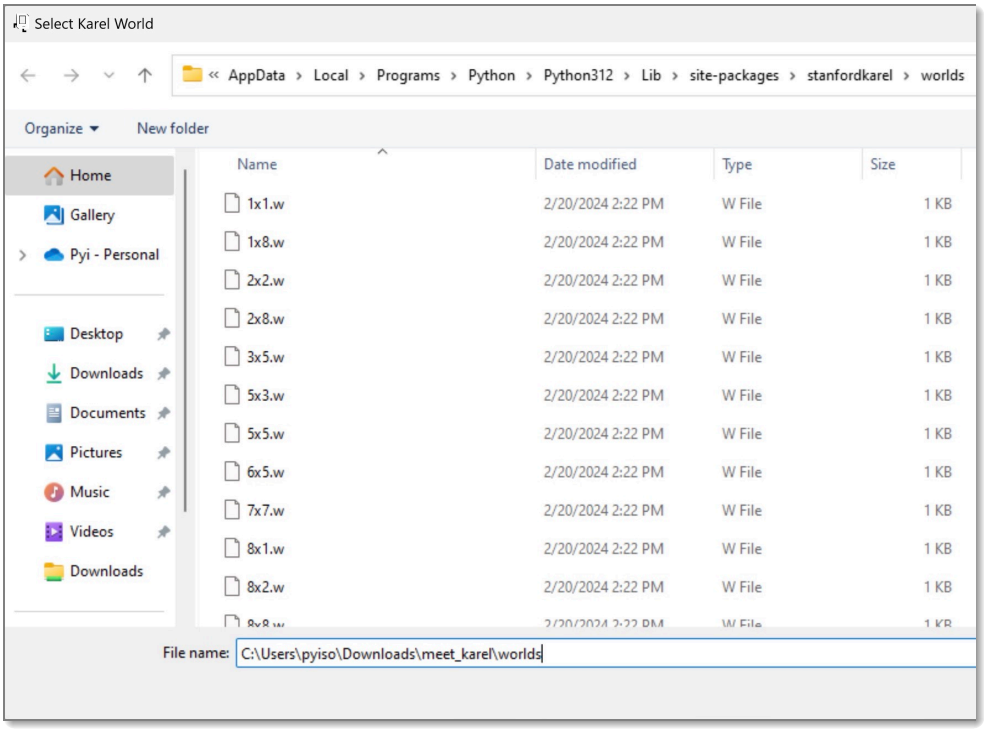
ဖြစ်ပါမယ်။ ဖိုင်ဒိုင်ယာလော့ဂ်ကနေ ဖော်ပြပါ လက်ရှိပရောဂျက် worlds ဖိုဒါကို တစ်ဆင့်ချင်း သွားပြီး တင်ချင်တဲ့ ကမ္ဘာဖိုင် (.w ဖိုင်) ကို ရွေးရမှာပါ။

အပေါ်ကနည်းနဲ့ အဆင်မပြေခဲ့ရင် အခုလိုစမ်းကြည့်ပါ။ လက်ရှိပရောဂျက် worlds ဖိုဒါကို ညာကလစ်နှိပ်ပြီး Copy Path လုပ်ပါ (ပုံ ခ/၂) ။ ဖိုင် ဒိုင်ယာလော့ဂ် File name မှာ ကူးထည့်ပါ (ပုံ ခ/၃)။ Enter ကီးနှိပ်ပါ။ ပရောဂျက် worlds ဖိုဒါကိုရောက်သွားပါမယ်။ လိုတဲ့ကမ္ဘာဖိုင် ရွေးတင်ရုံပါပဲ။ ပုံ (ခ/၄) မှာ meet_karel worlds ကို နမူနာပြထားပါတယ်။

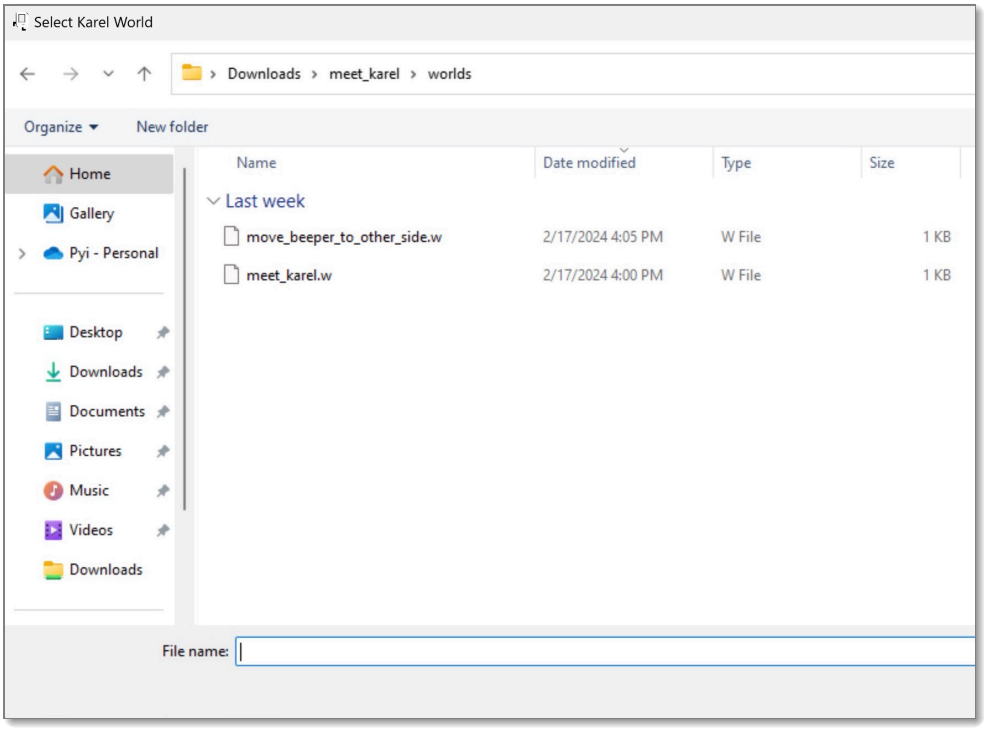


ပုံ ခ/၂

> အကယ်၍ အထက်ဖော်ပြပါနည်းတွေက ရှုပ်နေတယ်ထင်ရင် မှတ်ရ/သွားရ လွယ်တဲ့ Desktop, Downloads, Documents လို နေရာတစ်ခုခုမှာ သီးသန့်ဖိုဒါတစ်ခု ဆောက်ပြီး ပရောဂျက်အားလုံး ထည့်ထားတာ အရှင်းဆုံးပါပဲ။ ပရောဂျက်ဖိုဒါနေရာ သိရင် ဖိုင်ဒိုင်ယာလော့ဂ်ကနေ ဘယ်လိုမဆို ရောက်အောင် သွားလို့ရပါတယ်။


Select Karel World
« AppData › Local › Programs › Python › Python312 › Lib › site-packages › stanfordkarel › worlds
Organize
New folder
Home
Gallery
Pyi - Personal
Desktop
Downloads
Documents
Pictures
Music
Videos
Downloads
Name
Date modified
Type
Size
1x1.w 2/20/2024 2:22 PM W File 1 KB
1x8.w 2/20/2024 2:22 PM W File 1 KB
2x2.w 2/20/2024 2:22 PM W File 1 KB
2x8.w 2/20/2024 2:22 PM W File 1 KB
3x5.w 2/20/2024 2:22 PM W File 1 KB
5x3.w 2/20/2024 2:22 PM W File 1 KB
5x5.w 2/20/2024 2:22 PM W File 1 KB
6x5.w 2/20/2024 2:22 PM W File 1 KB
7x7.w 2/20/2024 2:22 PM W File 1 KB
8x1.w 2/20/2024 2:22 PM W File 1 KB
8x2.w 2/20/2024 2:22 PM W File 1 KB
File name:
C:\Users\pyiso\Downloads\meet_karel\worlds

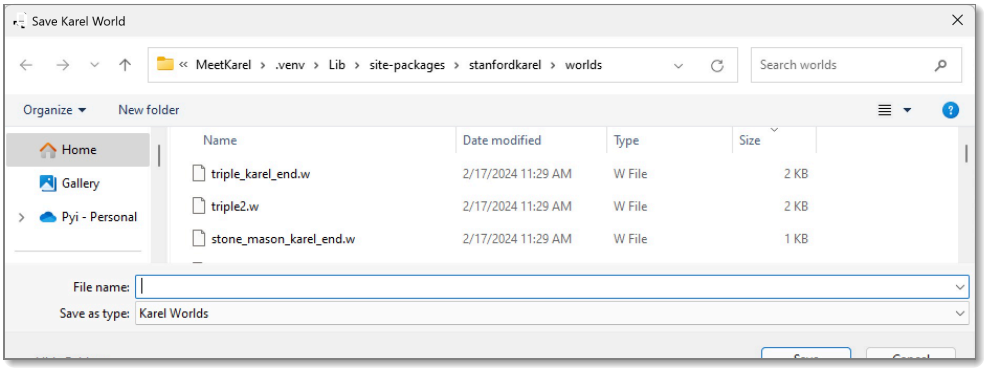
ပုံ ၁/၃


Select Karel World
› Downloads › meet_karel › worlds
Organize
New folder
Home
Gallery
Pyi - Personal
Desktop
Downloads
Documents
Pictures
Music
Videos
Downloads
Name
Date modified
Type
Size
Last week
move_beeper_to_other_side.w 2/17/2024 4:05 PM W File 1 KB
meet_karel.w 2/17/2024 4:00 PM W File 1 KB
File name:

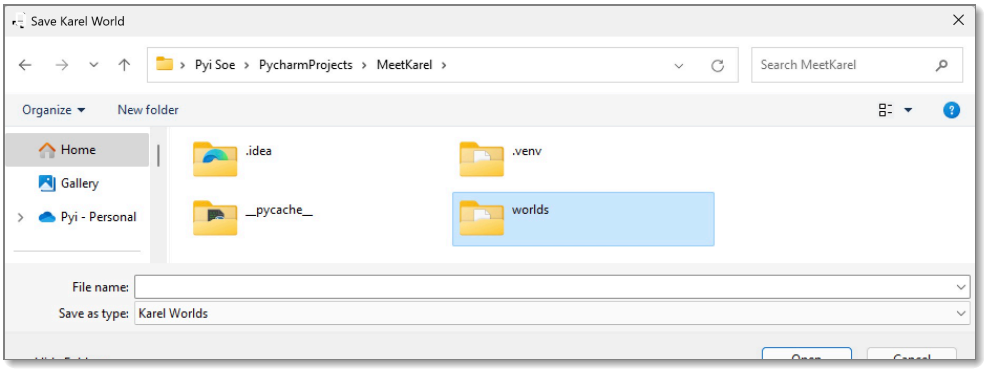
ပုံ ၁/၄

ကိုယ်ပိုင် ကားရဲလ်ကမ္ဘာ ဆောက်ခြင်း

ကားရဲလ်ရဲ့ ကမ္ဘာ အသစ်တစ်ခုဆောက်မယ်ဆိုရင် world_editor.py ဖိုင်ကို ညာကလစ်နှိပ် Run ပါ။ Would you like to load an existing world? လို့ ပေါ်လာပြီး Yes/No ရွေးခိုင်ပါလိမ့်မယ်။ No ကို နှိပ်ပါ။ ကမ္ဘာအရွယ်အစားအတွက် ကော်လံ ဘယ်နှစ်ခုလဲ၊ row ဘယ်နှစ်ခုလဲ ထည့်ပေးပါ။ ကိုယ်ပိုင် ကားရဲလ် ကမ္ဘာ တည်ဆောက်လို့ရတဲ့ ဝင်းဒိုးပွင့်လာပါလိမ့်မယ်။ ကားရဲလ် မျက်နှာမူရာအရပ်၊ ဘိပါအိတ်ထဲရှိ ဘိပါ အရေအတွက်၊ နံရံဆောက်/ဖျက်တာ၊ ဘိပါထည့်/ဖယ်ထုတ်တာ စတာတွေ လုပ်နိုင်ပါတယ်။ Save World နှိပ်ပြီး သိမ်းနိုင်ပါတယ်။ ဖိုင်ကိုသိမ်းတဲ့အခါ သူ့နဂို သိမ်းခိုင်းတဲ့ဖိုဒါ (default world folder) ထဲမှာ သို့မဟုတ် ပင်မ ပရောဂျက်ဖိုဒါထဲက worlds ဖိုဒါထဲမှာ သိမ်းရပါမယ်။



ပုံ ၁/၅



ပုံ ၁/၆

Python programming language
`import`, **`from`**, **`def`**, **`if`**
New term

fEnEmpstatement

ပါရာမီတာမပါရင်လည်း ဝိုက်ကွင်းအလွတ် တစ်စုံ ‘`()`’ တော့ပါရမယ်။
colon ‘`:`’
ဖန်ရှင်နဲ့ သက်ဆိုင်တဲ့ ကုဒ်ဘလောက် (*code block*)
quote သုံးခုတွဲ ‘`"""`’
ပရိုဂရမ်ဝင်းဒိုးမှာ `Run Program` ခလုတ်
`meet_karel.w` ကမ္ဘာကို