

Begin Modern Programming  
with

Python

Pyi Soe

# မာတိကာ

|     |  |    |
|-----|--|----|
| ၁   | Exceptions and Exception Handling                | ၁  |
| ၁.၁ | Raising Exception . . . . .                      | ၂  |
| ၁.၂ | Handling Exception. . . . .                      | ၄  |
| ၁.၃ | ဘယ်နေရာမှာ handle လုပ်သင့်လဲ . . . . .           | ၆  |
| ၁.၄ | Exception Handling နှင့် Control Flow . . . . .  | ၈  |
| ၁.၅ | Built-in and User-defined Exceptions . . . . .   | ၁၂ |
| ၁.၆ | Handling Multiple Exception. . . . .             | ၁၃ |
| ၂   | ဒေတာဘေ့စ်များနှင့် ဆက်သွယ်ဆောင်ရွက်ခြင်း         | ၁၉ |
| ၂.၁ | Database Management Systems . . . . .            | ၁၉ |
| ၂.၂ | PostgreSQL . . . . .                             | ၂၀ |
| ၂.၃ | Database API . . . . .                           | ၂၄ |
| ၃   | PostgreSQL ဒေတာဘေ့စ် ဆာဗာဆော့ဖ်ဝဲ ထည့်သွင်းခြင်း | ၂၉ |



# အခန်း ၁

## *Exceptions and Exception Handling*

တကယ့်လက်တွေ့ အသုံးချ ပရိုဂရမ်တွေမှာ ရာနှုန်းပြည့် အမှားကင်းစင်ဖို့ဆိုတာ မဖြစ်နိုင်ပါဘူး။ ဒီလိုလုပ် ပေးနိုင်တဲ့ နည်းပညာလည်း ခုချိန်ထိ မရှိသေးဘူး။ ဒါကြောင့် ပရိုဂရမ်တွေဟာ bug အနည်းနဲ့အများတော့ ပါကြတာပါပဲ။ ပရိုဂရမ်မှာ အမှားကြောင့် ဖြစ်တဲ့ bug တွေ လုံးဝမရှိအောင် တစ်နည်းတစ်လမ်းနဲ့ လုပ် နိုင်တယ် ဆိုအုံးတော့၊ တစ်ဖက်မှာ ပရိုဂရမ်တစ်ခုကို အသုံးပြုနေစဉ် ကြုံတွေ့ရတဲ့ ချွင်းချက် အခြေအနေ တွေက ရှိနေပါသေးတယ်။ အီးမေးလ်ပို့တဲ့အချိန် နက်ဝပ်က ဒေါင်းနေတာ၊ ဖွင့်တဲ့ ဖိုင်က ပျက်နေတာ၊ သုညနဲ့ စားတာ (division by zero)၊ မမ်မိုရီမလုံလောက်တာ စတဲ့ကိစ္စတွေ ပရိုဂရမ် အလုပ်လုပ်နေ စဉ် ကြုံတွေ့ရတတ်ပါတယ်။ ဒီလို ပြဿနာတွေက အမြဲတမ်း ဖြစ်နေတာတော့ မဟုတ်ဘူး၊ ရံဖန်ရံခါပဲ ဖြစ် တာဆိုပေမဲ့ ရှောင်လွှဲလို့ (သို့) လုံးဝမဖြစ်အောင် ကာကွယ်လို့လည်း မရပြန်ဘူး။ ဒီလို အဖြစ်အပျက်တစ် ခု ဖြစ်လာခဲ့ရင် ပရိုဂရမ်က လိုအပ်သလို စီမံထိန်းကွပ်လို့ရအောင် ရိုးရှင်းတဲ့ နည်းစနစ်တစ်မျိုး ရှိသင့်ပါ တယ်။ ပရိုဂရမ်ရဲ့ ပုံမှန်စီးဆင်းမှု (ပြဿနာ မဖြစ်ခဲ့ရင် ပုံမှန်အတိုင်း လုပ်ဆောင်သွားမဲ့ ကုန်တွေကို ဆိုလို တာ) လမ်းကြောင်းကိုလည်း ဒီနည်းစနစ်ကြောင့် အများကြီးပိုပြီး မရှုပ်ထွေးစေသင့်ဘူး။ တစ်နည်းအားဖြင့် ပြဿနာ မဖြစ်ရင် လုပ်ဆောင်မဲ့ အပိုင်းနဲ့ ဖြစ်ခဲ့ရင် လုပ်ဆောင်ရမဲ့ အပိုင်း ရောထွေးမနေသင့်ဘူး။ ခွဲခြား ထားရပါမယ်။

ဒီလို လိုအပ်ချက်တွေကို ဖြည့်ဆည်းပေးနိုင်တဲ့ နည်းလမ်းတွေထဲက အသုံးအများဆုံး တစ်ခုကတော့ *exception-handling mechanism* ပါပဲ။ ခေတ်ပေါ် programming language အားလုံးလိုလိုမှာ ထောက်ပံ့ပေးထားပါတယ်။ တချို့ language တွေမှာ အခြားနည်းလမ်းတွေ အသုံးပြုတာ တွေ့ရပေမဲ့ လက်တွေ့မှာ အခုပြောတဲ့ exception-handling လောက် မတွင်ကျယ်သေးဘူး။

*Exception-handling* သဘောတရားကို ဒီအခန်းမှာ အသေးစိတ် လေ့လာကြမှာပါ။ အောက်ပါ အတိုင်း အပိုင်းတွေခွဲ လေ့လာကြမှာပါ။

- Raising exception
- Handling exception
- Control flow
- Built-in exception class hierarchy
- User-defined exceptions
- Handling multiple exceptions

## ၁.၁ Raising Exception

ဖန်ရှင်တစ်ခုဟာ ပြဿနာတစ်ခုကြောင့် သူ့တာဝန် ပြီးမြောက်အောင်မြင်အောင် ဆက်လက်လုပ်ဆောင်ဖို့ မဖြစ်နိုင်တဲ့အခါ exception တစ်ခုကို raise လုပ်နိုင်ပါတယ်။ (မြန်မာလိုတော့ exception တက်အောင် လုပ်တာလို့ ပြောလေ့ရှိတယ်)။ အောက်ပါ fun\_c ဟာ အကြိမ်တစ်ရာမှာ (၅၀) လောက် IOError exception တက်အောင် တမင်ရည်ရွယ် လုပ်ထားတယ်။ (ဥပမာပြဖို့ အတွက်ပါ။ လက်တွေ့မှာ ဒီလိုလုပ်ဖို့ အကြောင်းမရှိပါဘူး)။ Random number ထုတ်ပြီး simulate လုပ်ထားတယ်။ အင်တာနက်ကနေ ဒေတာတချို့ ဖတ်ပေးတဲ့ ဖန်ရှင်လို့ ယူဆချင် ယူဆပါ။ လိုင်း မကောင်းတဲ့ ဒေသမှာဆိုတော့ ဒီဖန်ရှင်က မကြာခဏ ပြဿနာပေးတယ်ပေါ့။

```
import random

def fun_c():
    print('Starting fun_c...')
    # simulate IOError, will fail 50% of the time
    if random.uniform(0.0, 1.0) <= 0.5:
        raise IOError("Failed to read!")
    print('fun_c ends!')
```

fun\_c ကို main ကနေ ခေါ်ပြီး အကြိမ်အနည်းငယ် run ကြည့်ပါ။

```
def main():
    fun_c()
    print('main ends')

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

အဆင်ပြေတဲ့ အခါမှာ အခုလို

```
Starting main...
Starting fun_c...
fun_c ends!
main ends!
```

ထွက်တယ်။ Exception တက်ရင်တော့ ဒီလိုမျိုး error မက်ဆေ့ချ်တွေ

```
Traceback (most recent call last):
  File ".../ch11/how_exceptions_works1.py", line 19, in <module>
    main()
  File ".../ch11/how_exceptions_works1.py", line 14, in main
    fun_c()
  File ".../ch11/how_exceptions_works1.py", line 8, in fun_c
    raise IOError("Failed to read!")
OSError: Failed to read!
Starting main...
Starting fun_c...
```

ကျလာမှာပါ။ main() ကနေ fun\_c() ခေါ်ပြီး အဲဒီမှာ IOError ဖြစ်သွားတယ်လို့ ဖော်ပြထားတာ တွေ့ရတယ်။ (most recent call last) လို့လည်း တွေ့ရတယ်။ အောက်ဆုံးမှာ နောက်ဆုံးခေါ်ခဲ့တဲ့ ဖန်ရှင်လို့ ဆိုလိုတာ (နောက်ဆုံး ခေါ်ခဲ့တာ fun\_c)။ ဖန်ရှင်တစ်ခုမှာ exception တက်တဲ့အခါ (သို့) ဖန်ရှင်တစ်ခုက exception ကို raise လိုက်တဲ့အခါ ၎င်းဖန်ရှင်ကို ခေါ်တဲ့ ဖန်ရှင်တွေအားလုံး 'တောက်လျှောက် fail ဖြစ်မယ်'။ အခုဥပမာမှာ main ကနေ fun\_c ကို ခေါ်တယ်။ fun\_c မှာ exception ဖြစ်တော့ main လည်း ပြီးအောင်ဆက် အလုပ်မလုပ်ပေးနိုင်ဘူး။ fail ဖြစ်သွားတယ်။

စောစောက မက်ဆေ့ချ်တွေကို သေချာဂရုစိုက်ကြည့်ပါ။ fun\_c မှာဆိုရင် exception ဖြစ်စေတဲ့ နေရာ အောက်ပိုင်းက စတိတ်မန့်တွေ၊ main မှာဆိုရင် fun\_c ကို ခေါ်ထားတဲ့နေရာရဲ့ အောက်က စတိတ်မန့်တွေ အလုပ်မလုပ်သွားဘူး (main အတွက် fun\_c ခေါ်တဲ့လိုင်းက exception ဖြစ်စေတဲ့နေရာ)။ အခုကိစ္စမှာ ဖန်ရှင်နှစ်ခုလုံးရဲ့ exception ဖြစ်တဲ့နေရာ အောက်ပိုင်းမှာ print စတိတ်မန့် တစ်ကြောင်းစီပဲ ရှိပါတယ်။ အခြားစတိတ်မန့်တွေ ရှိခဲ့ရင်လည်း အလုပ်လုပ်မှာ မဟုတ်ဘူး။

အကယ်၍ main က fun\_a ကိုခေါ်၊ fun\_a က တစ်ဆင့် fun\_b ကိုခေါ်၊ fun\_b ကနေမှ fun\_c ကို နောက်ဆုံး ခေါ်ထားရင် fun\_c မှာ exception ဖြစ်တဲ့အခါ fun\_b က စပြီး fail ဖြစ်မယ်။ ပြီးရင် သူ့ကိုခေါ်တဲ့ fun\_a ဆက် fail မယ်။ နောက်ဆုံးမှာ fun\_a ကိုခေါ်ထားတဲ့ main ဖန်ရှင် fail ဖြစ်ပြီး ပရိုဂရမ်တစ်ခုလုံး ရပ်ဆိုင်းသွားမှာ ဖြစ်တယ်။ ရှေ့စာပိုဒ်မှာ ပြောခဲ့တဲ့ 'တောက်လျှောက် fail ဖြစ်မယ်' ဆိုတာ အဲဒီလိုဖြစ်စဉ်ကို ဆိုလိုတာ။

```
# fun_c exception ဖြစ်ရင် သူ့ကို ခေါ်ထားတဲ့ ဖန်ရှင်အားလုံး fail ဖြစ်ပါမယ်
def main():
    print('Starting main...')
    fun_a()
    print('main ends!')

def fun_a():
    print('Starting fun_a...')
    fun_b()
    print('fun_a ends!')

def fun_b():
    print('Starting fun_b...')
    fun_c()
    print('fun_b ends!')

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

Exception raise လုပ်တာဟာ ဖန်ရှင်တစ်ခုက ၎င်းလုပ်ဆောင်ရမဲ့ တာဝန်ကို ပြဿနာ တစ်ခုခုကြောင့် ပြီးမြောက် အောင်မြင်အောင် မလုပ်ဆောင်နိုင်တော့ဘူးဆိုတာ ဖန်ရှင်ခေါ်တဲ့သူကို အသိပေးတဲ့ နည်းလမ်းတစ်မျိုးလို့ ယူဆနိုင်ပါတယ်။ ဖန်ရှင်တစ်ခုကနေ အစပြု ဖြစ်ပေါ်တဲ့ exception ဟာ အဲဒီဖန်ရှင်ကို ခေါ်ထားတဲ့ ကွင်းဆက် (call chain) တစ်လျှောက် ပါဝင်တဲ့ဖန်ရှင် တစ်ခုပြီးတစ်ခု exception ဖြစ်စေပြီး နောက်ဆုံးမှာ ပရိုဂရမ်တစ်ခုလုံးကို ရပ်တန့်သွားစေမှာပါ။ Exception ဖြစ်ခဲ့ရင် ပရိုဂရမ်တစ်ခုလုံးကို ပြန်မသွားဘဲ၊ မသက်ရောက်စေဘဲ ထိန်းကွပ်ပေးလို့ရတဲ့ နည်းလမ်းရှိရပါမယ်။ အဲဒါကတော့ exception ကို handle လုပ်ပေးခြင်းပါပဲ။

## ၁.၂ Handling Exception

စောစောက ဥပမာမှာ fun\_c ကိုခေါ်တဲ့အခါ ဖြစ်နိုင်တဲ့ exception ကို fun\_b က အခုလို handle လုပ်နိုင်ပါတယ်။

```
def fun_b():
    print('Starting fun_b...')
    try:
        fun_c()
        print("fun_c was successful!")
    except IOError as e:
        print(e)
        print("Poor connection!")
    print('fun_b ends!')
```

Python မှာ try...except က exception handling အတွက်ပါ။ Exception ဖြစ်နိုင်တဲ့ ဖန်ရှင် ကို ခေါ်တဲ့အခါ try ဘလောက်ထဲမှာ ခေါ်ရပါမယ် (ဖြစ်ခဲ့ရင် handle လုပ်မယ်ဆိုတဲ့ ရည်ရွယ်ချက်ရှိ ရင်ပေါ့)။ except ဘလောက်က exception ဖြစ်ခဲ့ရင် handle လုပ်မဲ့ ကိစ္စအတွက်။

```
except IOError as e:
```

IOError က handle လုပ်မဲ့ exception အမျိုးအစားကို သတ်မှတ်တာ။ ဆိုလိုတာက IOError သီးသန့်ကိုပဲ handle လုပ်မယ်။ IOError မဟုတ်တဲ့ အခြား exception တွေကို handle မလုပ်ဘူး။ (ဒါနဲ့ ပါတ်သက်ပြီး နောက်ပိုင်းမှာ ထပ်ရှင်းပြမှာပါ)။ e က ဖြစ်ပေါ်တဲ့ IOError exception အတွက် ဗေရီရေဘဲလ်ပါ။ Exception ဖြစ်ရင် fun\_c က raise လုပ်လိုက်တဲ့ IOError အောက်ဂျက်ကို ဒီ ဗေရီရေဘဲလ်မှာ ထည့်ပေးမှာ ဖြစ်တယ်။ (Python မှာ IOError, ValueError, NameError စတဲ့ ကလပ်စ်တွေ ပါရှိပြီး ဖြစ်ပေါ်တဲ့ exception အမျိုးအစားအလိုက် သက်ဆိုင်ရာ exception အောက် ဂျက်ကို raise လုပ်ရတာပါ)။

စောစောက ဥပမာမှာ fun\_b ကို အထက်ပါအတိုင်း exception handling ထည့်ပြီး စမ်းသပ် ကြည့်တဲ့အခါ exception ဖြစ်တဲ့အခါ အခုလို

```
Starting main...
Starting fun_a...
Starting fun_b...
Starting fun_c...
Failed to read!
Poor connection!
fun_b ends!
fun_a ends!
main ends!
```

တွေ့ရမှာပါ။ fun\_b မှာ exception handling လုပ်ထားတဲ့အတွက် fun\_c က exception ဖြစ် ခဲ့ရင် အဲ့ဒီ exception ဟာ fun\_a နဲ့ main ဆီကို ထပ်ဆင့် မကူးစက်သွားတော့ဘူး။ Exception handling ဆိုတာ ပရိုဂရမ် အခြားအစိတ်အပိုင်းတွေကို exception မကူးစက်သွားအောင် ကွာရန်တင်း လုပ် ထိန်းချုပ်တာလို့ ယူဆနိုင်ပါတယ်။

## Failure နဲ့ Exception ဘာကွာခြားလဲ

အခုရှင်းပြပြီးခဲ့သလောက်မှာ fail ဖြစ်တာနဲ့ exception ဖြစ်တာ၊ ဒီသဘောတရားနှစ်ခု မရောထွေးသင့်ပါဘူး။ ဖန်ရှင်တစ်ခု fail ဖြစ်တယ်ဆိုတာ ပြဿနာတစ်ခုခုကြောင့် သူ့တာဝန်ကို အောင်မြင်အောင် မလုပ်နိုင်၊ အဲဒီ ပြဿနာကိုလည်း ကိုင်တွယ်ထိန်းကွပ်မထားတဲ့ အခြေအနေလို့ အကြမ်းဖျဉ်းယူဆပါ။ ဖန်ရှင်တစ်ခုက ပုံမှန်လမ်းကြောင်းအတိုင်း ပြီးမြောက်အောင် လုပ်ဆောင်သွားရင်၊ သို့မဟုတ် ပြဿနာ တစ်ခုခု ဖြစ်ခဲ့ရင်လည်း ဆက်မပြန့်သွားအောင် ကိုင်တွယ် ထိန်းကွပ်ပေးလိုက်ရင် successful ဖြစ်တယ်လို့ ယူဆရပါမယ်။ Exception က failure ဖြစ်စေ 'နိုင်' တဲ့ အကြောင်းအရင်း။ ဒါပေမဲ့ exception ဖြစ်ရင် fail ဖြစ်မယ် ပုံသေမှတ်လို့မရဘူး။ Exception ကို handle လုပ်လိုက်ရင် fail မဖြစ်တော့ဘူး။

fun\_c exception ဖြစ်တော့ ခေါ်တဲ့ဖန်ရှင်အားလုံး တစ်ခုပြီးတစ်ခု ဆက်တိုက် fail ဖြစ်တယ်လို့ ရှေ့ပိုင်းမှာ ပြောခဲ့တယ်။ ဒါကို ပိုပြီးတိကျအောင် ပြောရမယ်ဆိုရင် fun\_c exception ဖြစ်တဲ့အခါ သူ့ကိုခေါ်တဲ့ fun\_b ကိုလည်း exception ဖြစ်စေတယ်။ fun\_b က အဲဒီ exception ကို handle လုပ်ထားရင် fail မဖြစ်ဘူး။ မလုပ်ထားရင်တော့ သူ့ကိုယ်တိုင်လည်း fail ဖြစ်ပြီး သူ့ကိုခေါ်တဲ့ fun\_a ကို exception ဆက်ဖြစ်စေပါတယ်။ fun\_a မှာလည်း ဒီသဘောအတိုင်း ဆက်ဖြစ်မှာပါ။ Exception handle လုပ်လိုက်ရင် fail မဖြစ်တော့ဘူး။ မလုပ်ထားရင်တော့ သူ့ကိုခေါ်တဲ့ main ဖန်ရှင်ကို exception ဆက်ဖြစ်စေပါလိမ့်မယ်။

နောက်ထပ် သိထားဖို့ အရေးကြီးတာ တစ်ခုက exception ဖြစ်တဲ့အခါ try ဘလောက်ထဲမှာ ပါတဲ့ အောက်က စတိတ်မန်တွေကို ကျော်ပြီး except ဘလောက်ထဲ ချက်ချင်း ရောက်သွားမှာပါ။ try ဘလောက်ဟာ ပုံမှန် လုပ်ဆောင်မဲ့ လမ်းကြောင်း (normal execution flow) အတွက်ပါ။ Exception ဖြစ်ခဲ့ရင်တော့ ဒီလမ်းကြောင်းအတိုင်း ဆက်အလုပ်လုပ်လို့ မရတော့ဘူး။ ပုံမှန်မဟုတ်တဲ့ အခြေအနေမှာ လုပ်ဆောင်ရမဲ့ except ဘလောက်ကို လွှဲပြောင်း လုပ်ဆောင်ပေးရပါမယ်။ အခု ပြထားတာက ပုံမှန်အတိုင်း သွားမဲ့လမ်းကြောင်းပါ။

```
def fun_b():
    print('Starting fun_b...')
    try:
        fun_c()
        print("fun_c was successful!")
    except IOError as e:
        print(e)
        print("Poor connection!")
    print('fun_b ends!')
```

အောက်မှာပြထားတာက IOError ဖြစ်ခဲ့ရင် လုပ်ဆောင်မဲ့ပုံ (fun\_c ခေါ်ထားတဲ့လိုင်းကနေ except ကို ခုန်ပြီးရောက်သွားတာ သတိပြုပါ။)

```
def fun_b():
    print('Starting fun_b...')
    try:
        fun_c()
        print("fun_c was successful!")
    except IOError as e:
        print(e)
        print("Poor connection!")
    print('fun_b ends!')
```



ရှေ့က ဥပမာမှာ fun\_b handle မလုပ်ဘဲ fun\_a က handle လုပ်လို့လဲရတယ်။ ဒါမှမဟုတ် fun\_b နဲ့ fun\_a မှာ handle မလုပ်ဘဲ main က လုပ်နိုင်ပါတယ်။ အောက်ပါအတိုင်း fun\_a မှာ handle လုပ်မယ်ဆိုပါစို့

```
def fun_a():
    print('Starting fun_a...')
    try:
        fun_b()
        print("fun_b was successful!")
    except IOError as e:
        print(e)
        print("Poor connection!")
    print('fun_a ends!')

def fun_b():
    print('Starting fun_b...')
    fun_c()
    print('fun_b ends!')
```

fun\_c exception တက်ရင် handle မလုပ်ထားတဲ့ fun\_b လည်း exception ဆက်ဖြစ်ပါမယ်။ အဲဒီ exception ကို fun\_a က handle လုပ်လိုက်တဲ့ အတွက် main ဆီကို ဆက်လက်မကူးစက် သွားတော့ ပါဘူး။ Output အခုလို ထွက်တာ တွေ့ရမှာပါ။

```
Starting main...
Starting fun_a...
Starting fun_b...
Starting fun_c...
Failed to read!
Poor connection!
fun_a ends!
main ends!
```

Exception ဖြစ်တဲ့အခါ "fun\_b ends!" နဲ့ "fun\_b was successful!" အတွက် print စတိတ် မနဲ့တွေ့ကို ကျော်သွားတာ သတိပြုကြည့်ပါ။

## ၁.၃ ဘယ်နေရာမှာ handle လုပ်သင့်လဲ

ဖန်ရှင်တွေ တစ်ခုပြီးတစ်ခုဆင့် ခေါ်ထားတဲ့အခါ exception ဖြစ်ခဲ့ရင် ဘယ်ဖန်ရှင်က handle လုပ် သင့်လဲ စဉ်းစားဆုံးဖြတ်ဖို့ လိုလာတယ်။ ဒီကိစ္စက ဘယ်မှာ handle လုပ်ရမယ် ပုံသေပြောလို့တော့ မရ ဘူး။ အခြေအနေနဲ့ လိုအပ်ချက်ပေါ် မူတည် ဆုံးဖြတ်ရတာမျိုး။

```
def read_sensor():
    if random.uniform(0.0, 1.0) <= 0.2:
        raise IOError("Failed to read!")
    return random.randrange(1, 11)
```

ဒီဖန်ရှင်က sensor device တစ်ခုဆီကနေ ဒေတာဖတ်တာကို simulate လုပ်ထားတဲ့ ဖန်ရှင်ပါ။ Sensor ကြောင့်လို့သော်လည်းကောင်း၊ network ကြောင့်သော်လည်းကောင်း (၂၀) ရာနှုန်း fail ဖြစ်တယ် ဆိုပါတော့။

Sensor တန်ဖိုး သုံးခုတစ်တွဲ ဖတ်ပြီး စောင့်ကြည့်လေ့လာရမယ်လို့ စိတ်ကူးကြည့်ပါ။ တန်ဖိုးသုံး ခု အတွဲလိုက်ရအောင် ဖတ်ရမှာပါ။ Exception ဖြစ်လို့ သုံးခုမပြည့်သေးရင် ပြည့်တဲ့ထိ ထပ်ကြိုးစားရပါ မယ်။ ဖတ်တဲ့အခါ တစ်ခါနဲ့တစ်ခါ စက္ကန့်တစ်ဝက်ခြား ဖတ်ပါတယ်။ တစ်ကယ့် လက်တွေ့မှာလည်း sensor ကနေ ဒေတာဖတ်တဲ့အခါ တရစပ် ဖတ်လေ့မရှိဘူး။ အချိန်အနည်းငယ် ခြားပြီးဖတ်တယ်။

```
def read_3vals():
    vals = []
    while True:
        try:
            vals.append(read_sensor())
            if len(vals) == 3:
                return vals
        except IOError as err:
            pass
        time.sleep(0.5)
```

အခုကိစ္စအတွက် read\_3vals ဖန်ရှင်မှာ handle လုပ်ပေးရပါမယ်။ မလုပ်ဘဲထားရင် fail ဖြစ်ပြီး တန်ဖိုးသုံးခု ပြည့်အောင်ဖတ်လို့ မရနိုင်ဘူး။

အခုတစ်ခါ ကြားထဲမှာ ပြဿနာတစ်စုံတစ်ရာ မရှိဘဲ ဆက်တိုက် ဖတ်လို့ရတဲ့ တန်ဖိုးသုံးခု လိုချင် တယ် ယူဆပါ။ ဒီကိစ္စအတွက် exception handle မလုပ်ဘဲ ဖန်ရှင်တစ်ခု အခုလို ရေးနိုင်တယ်။

```
def read_3_times():
    vals = []
    for i in range(3):
        vals.append(read_sensor())
        time.sleep(0.5)
    return vals
```

ဒီဖန်ရှင်က ပုံမှန်ဆိုရင်တော့ sensor ကို သုံးကြိမ်ဖတ်မှာပါ။ ဒါပေမဲ့ read\_sensor မှာ IOError exception ဖြစ်ခဲ့ရင် handle မလုပ်ထားတဲ့အတွက် အခု read\_3\_times လည်း ဆက် fail ဖြစ် မယ်။ ကံကောင်းလို့ သုံးခါလုံး အဆင်ပြေခဲ့ရင်တော့ ဖတ်ထားတဲ့ တန်ဖိုးသုံးခုပါတဲ့ vals ကို ပြန်ပေး မှာပါ။ စောစောကဖန်ရှင်နဲ့ အဓိက ကွာခြားချက်ကို သတိပြုပါ။ read\_3vals က တန်ဖိုး သုံးခုပြည့်တဲ့ ထိ ဖတ်ပေးရမှာပါ။ ဒါကြောင့် exception handle လုပ်ဖို့ လိုကို လိုတယ်။ ခုဖန်ရှင်က တန်ဖိုးသုံးခုကို ကြားထဲမှာ fail မဖြစ်ဘဲ ဆက်တိုက် ဖတ်လို့ရမှပဲ ပြန်ပေးရမယ်။ ဒါကြောင့် handle မလုပ်ဘဲ ထားလို့ရ တယ်။ read\_3\_times ကို ခေါ်တဲ့သူက handle လုပ်/မလုပ် ဆက်လက်ဆုံးဖြတ်နိုင်ပါတယ်။

ခုဏက ဖန်ရှင် အသုံးတည့်လာမဲ့ အခြေအနေတစ်ခု စဉ်းစားကြည့်ရအောင်။ သုံးခုတစ်တွဲ (၁၀) ခါ ဖတ်ရင် ဘယ်နှစ်ခါ fail ဖြစ်လဲ ဆန်းစစ်ကြည့်ချင်တယ် စိတ်ကူးကြည့်ပါ။ read\_3\_times ကို အခြေခံ ပြီး failure\_rate ဖန်ရှင်ကို အခုလို သတ်မှတ်နိုင်ပါတယ်။

```
def failure_rate():
    fail = 0
    for i in range(10):
```

```

try:
    read_3_times()
except IOError as e:
    fail += 1
print(fail)
return Fraction(fail, 10)

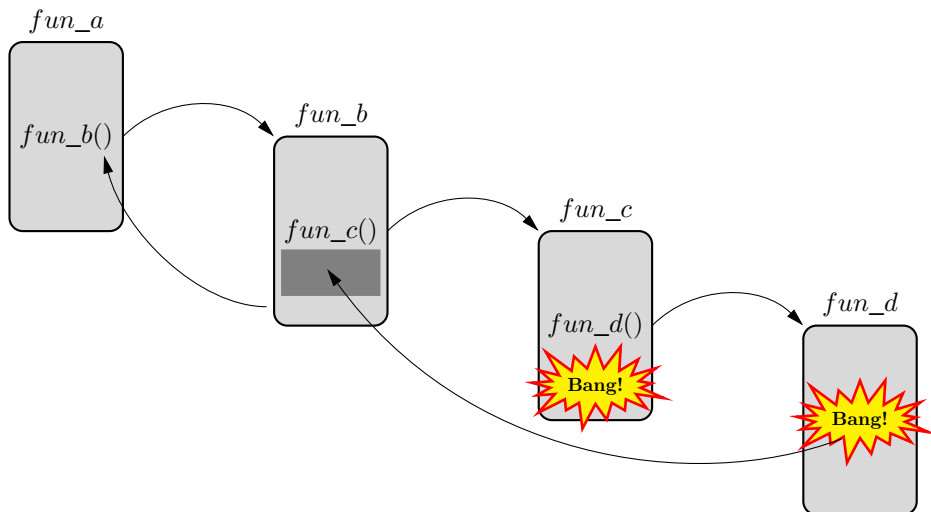
```

`read_3_times` က ဖြစ်ပေါ်တဲ့ exception ကို ဘယ်နှစ်ကြိမ် `fail` ဖြစ်လဲ ရေတွက်ဖို့ အသုံးပြုထားတာကို တွေ့ရပါတယ်။

အခု လေ့လာတွေ့ရှိချက်တွေကို အနှစ်ချုပ်ပြန်ကြည့်ရင် ဖန်ရှင်တစ်ခု exception ဖြစ်တဲ့အခါ ၎င်းဖန်ရှင်ကို ခေါ်ထားတဲ့ ကွင်းဆက်တစ်လျှောက်လုံးက ဖန်ရှင်တွေ `fail` ဖြစ်နိုင်ပါတယ်။ `Fail` မဖြစ်အောင် ဖန်ရှင်တစ်ခုက exception ကို `handle` လုပ်ရပါမယ်။ ဘယ်ဖန်ရှင်က `handle` လုပ်ရမလဲကတော့ ပုံသေမရှိဘူး။ လိုအပ်ချက်ပေါ် မူတည်ပြီး ဆုံးဖြတ်ရလေ့ရှိတယ်။

## ၁.၄ Exception Handling နှင့် Control Flow

Exception handling ကို ပရိုဂရမ်တစ်ခု ပုံမှန်စီးဆင်းရာ လမ်းကြောင်းနဲ့ ပုံမှန်အခြေအနေ မဟုတ်တဲ့အခါ စီးဆင်းရာလမ်းကြောင်း ခွဲခြားသတ်မှတ်ပေးတဲ့ နည်းစနစ်အဖြစ် ရှုမြင်နိုင်ပါတယ်။ Exception ဖြစ်တဲ့အခါ စီးဆင်းပုံကို အကြမ်းဖျဉ်းအားဖြင့် ပုံ (၁.၁) မှာတွေ့ရသလို မြင်ကြည့်နိုင်ပါတယ်။



### ပုံ ၁.၁ Exception Handling

`fun_a`, `fun_b`, `fun_c`, `fun_d` တစ်ခုပြီးတစ်ခု ဆင့်ပြီး ခေါ်ထားတယ်။ မီးခိုးဖျော့ ထောင့်စွန်းဝိုင်း ထောင့်မှန်စတုဂံတွေက ဖန်ရှင်တစ်ခုစီကို ကိုယ်စားပြုတယ်။ အထဲမှာ ဖန်ရှင်ခေါ်ထားတာ တွေ့ရမယ်။ `Handle` လုပ်တာက `fun_b` မှာ (မီးခိုးရင့် ထောင့်မှန်စတုဂံအသေးလေးက `handle` လုပ်တဲ့ အပိုင်းလို့ ယူဆပါ)။

`fun_d` မှာ exception ဖြစ်တယ်။ `fun_c` လည်း exception ဆက်ဖြစ်ပြီး `fail` ဖြစ်မယ်။ သူ့ကို ခေါ်တဲ့ `fun_b` ကိုလည်း exception ဆက်ဖြစ်စေတယ်။ `fun_b` `handle` လုပ်လိုက်တဲ့ အတွက် `fail` မဖြစ်ဘူး။ `handle` လုပ်ပြီးသွားတော့ ၎င်းကို ခေါ်ခဲ့တဲ့ `fun_a` ထဲကို ပြန်ရောက်သွားပြီး `fun_a`

ဆက်အလုပ်လုပ်မှာပါ။

ပုံအရ exception ဖြစ်တဲ့အခါ မူလစဖြစ်တဲ့နေရာကနေ handle လုပ်ထားတဲ့ အနီးဆုံး နေရာကို ခုန်ပြီးရောက်သွားတာကို တွေ့ရမှာပါ။ ဒီလို ခုန်သွားနိုင်တာဟာ exception handling မှာ အဓိကကျတဲ့ လုပ်ဆောင်ချက်ဖြစ်တယ်။

## else နှင့် finally

try...except အောက်မှာ else ဘလောက် နဲ့ finally ဘလောက် ရှိနိုင်ပါတယ်။ Exception မဖြစ်တဲ့အခါမှပဲ လုပ်ဆောင်ချင်တဲ့ စတိတ်မန်တွေကို else ဘလောက်ထဲမှာ ထည့်နိုင်ပါတယ်။ တစ်နည်းအားဖြင့် try ဘလောက် ပြဿနာမရှိဘဲ အောင်မြင်ပြီးစီးမှသာလျှင် else ဘလောက်ကို လုပ်ဆောင်မှာပါ။ Exception ဖြစ်ခဲ့ရင်တော့ လုပ်ဆောင်ပေးမှာ မဟုတ်ပါဘူး။

```
def process_sensor_data():
    while True:
        try:
            val = read_sensor()
        except IOError as e:
            print(e)
        else:
            use_data(val)
            notify_if_necessary()
            print("Sensor data read successfully...")

    time.sleep(1)
    print("One iteration completed...")
```

```
Sensor data read successfully...
One iteration completed...
Sensor data read successfully...
One iteration completed...
Failed to read!
One iteration completed...
Failed to read!
```

ဒီ output ကို လေ့လာကြည့်ရင် else ဘလောက်ကို exception မတက်မှပဲ လုပ်ဆောင်ပေးတယ်ဆိုတာ မြင်နိုင်မှာပါ။ try...except...else အပြင် အောက်ဆုံးက နှစ်ကြောင်းနဲ့ သဘောတရား မတူတာကိုလည်း သတိပြု ကြည့်ပါ။

```
time.sleep(1)
print("One iteration completed...")
```

Exception ဖြစ်ဖြစ်၊ မဖြစ်ဖြစ် ဒီ စတိတ်မန် နှစ်ခုကိုက လုပ်ဆောင်ပေးတယ်။ else အပိုင်းကိုတော့ exception မတက်မှပဲ လုပ်ဆောင်တယ်။



စောစောက ဖန်ရှင်ကို else မပါဘဲ အောက်ပါကဲ့သို့ ရေးမယ်ဆိုရင်လည်း ရလဒ်အားဖြင့် တူတူပါပဲ။ else ကို အသုံးပြုရတဲ့ အဓိက အကြောင်းအရင်းက exception ဖြစ်စေနိုင်တဲ့ အပိုင်းနဲ့ မဖြစ်စေနိုင်

တဲ့ အပိုင်း သီးသန့်ခွဲထားဖို့အတွက်ပါ။

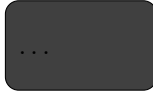

```
def process_sensor_data():
    while True:
        try:
            val = read_sensor()
            use_data(val)
            notify_if_necessary()
            print("Sensor data read successfully...")
        except IOError as e:
            print(e)
            time.sleep(1)
            print("processed single sensor data successfully...")
```

အခုပုံစံမှာက try ထဲက ဖန်ရှင်တွေထဲက ဘယ်ဟာက exception ဖြစ်စေတဲ့ အရင်းအမြစ်လဲ အလွယ်တကူ မသိနိုင်တော့ဘူး။ စောစောက else နဲ့ ပုံစံမှာက read\_sensor က exception ဖြစ်နိုင်တဲ့ ဖန်ရှင်ဖြစ်ရမယ်၊ အဲဒီ exception ကို handle လုပ်ထားတယ်ဆိုတာ သိသာ မြင်သာပါတယ်။

```
try:
    ...
    some_fun()
except:
    ...
else:
    ...
```

```
try:
    ...
    some_fun()
except:
    ...
else:
    ...
```

Exception မဖြစ်တဲ့ အခါနဲ့ ဖြစ်တဲ့အခါ try...except...else အလုပ်လုပ်ပုံ နှိုင်းယှဉ်ပြထားတာပါ။ မီးခိုးရောင် အပျော့ ဘလောက်တွေက လုပ်ဆောင်မဲ့ ဘလောက်တွေပါ။ မီးခိုးရင့်ရောင် ဘလောက်တွေကိုတော့ လုပ်ဆောင်မှာ မဟုတ်ပါဘူး။

## finally

except အပြီးမှာ finally ဘလောက် ရှိနိုင်ပါတယ်။ Exception ဖြစ်သည်ဖြစ်စေ၊ မဖြစ်သည်ဖြစ်စေ finally အပိုင်းကို လုပ်ဆောင်ပေးမှာ ဖြစ်တယ်။ return လုပ်ရင်တောင်မှ finally လုပ်ဆောင်ပြီးမှပဲ လုပ်မှာပါ။

```
def test_read():
    try:
        val = read_sensor()
        print("Value: " + str(val))
        return
    except IOError as e:
        print("Error while reading!")
        return
    finally:
        print("Don't skip this!!!")
```

ဒီဖန်ရှင်ကို ထပ်ခါထပ်ခါ run ပြီး စမ်းကြည့်ပါ။ Exception မဖြစ်တဲ့ အခါ

```
Value: 5
Don't skip this!!!
```

ဖြစ်တဲ့အခါ

```
Error while reading!
Don't skip this!!!
```

ကို တွေ့ရမှာပါ။

ဖန်ရှင် return မဖြစ်မီ finally ဘလောက်ကို လုပ်ဆောင်တာကို တွေ့ရတယ်။ return လုပ်ရင် ခေါ်ခဲ့တဲ့နေရာ ချက်ချင်းပြန်ရောက်တယ် ဆိုပေမဲ့ finally ပါရင်တော့ ချင်းချက်အနေနဲ့ မှတ်ရပါမယ်။ try...except ဘလောက်တွေဟာ တကယ့်လက်တွေ့မှာ အခုဥပမာတွေလို ရိုးရှင်းမှာ မဟုတ်ဘူး။ ဒီထက် အများကြီး ပိုပြီးရှုပ်ထွေး နိုင်ပါတယ်။ ကွန်ဒီရှင်နယ်လ်တွေ၊ loop တွေ၊ break, early return စတဲ့ဟာတွေ ရောယှက်နေတဲ့အခါ နောက်ဆုံးပိတ် လုပ်ဆောင်ပေးရမဲ့ final steps တချို့ ကျန်ခဲ့တာ ဖြစ်ဖို့ အလားအလာများတယ်။ ဥပမာ test\_read မှာ အခြေအနေပေါ်မူတည်ပြီး early return လုပ်မယ်ဆိုပါစို့။ finally မသုံးဘူးဆိုရင် return မတိုင်ခင် print တွေ အခုလို ထည့်ပေးရပါမယ်။

```
def test_read():
    try:
        val = read_sensor()
        print("Value: " + str(val))
        if val < 3:
            # Do not forget here
            print("Don't skip this!!!")
            return
        elif val < 5:
            use(val)
        elif val < 8:
            # ...
            etc.,
            # Do not forget here
            print("Don't skip this!!!")
    except IOError as e:
        print("Error while reading!")
```

```
# Do not forget here
print("Don't skip this!!!")
return
```

တစ်နေရာရာမှာ ကျန်ခဲ့ပြီး မှားနိုင်ခြေ များတယ်။ ဒီအားနည်းချက်အပြင် ပိုကြီးတဲ့ ပြဿနာက ဘယ်ဟာက မဖြစ်မနေ လုပ်ဆောင်ရမဲ့ကိစ္စ လဲဆိုတာ ကြည့်ရုံနဲ့ ကွဲပြားပြား မမြင်နိုင်တော့ဘူး။ နေရာအတော်များများမှာ ဖြန့်ပြီးရှိနေတယ်။ finally သုံးခြင်းအားဖြင့် ဒီပြဿနာကို ဖြေရှင်းနိုင်ပါတယ်။

ဒေတာဘေ့စ် ချိတ်ဆက်ခြင်း၊ ဖိုင်ဖတ်ခြင်း/ရေးခြင်း၊ နက်ဝပ်ချိတ်ဆက်ခြင်း စတဲ့ကိစ္စတွေ လုပ်ဆောင်တဲ့အခါ အသုံးပြုထားတဲ့ ဖိုင်၊ ကွန်နက်ရှင် စတဲ့ resource တွေကို release လုပ်ပေးဖို့ အရေးကြီးပါတယ်။ ဒီလို မလုပ်ရင် ကွန်ပျူတာ စနစ်ရဲ့ CPU cycles, memory အစရှိတဲ့ resource တွေ အလဟဿ ပြုန်းတီးစေတဲ့အတွက် ပြဿနာရှိပါတယ်။ အလုပ်ပြီးတဲ့အခါ လက်စသတ်တာ (clean up code)၊ resource release လုပ်တာ စတဲ့ကိစ္စတွေအတွက် finally ကို သုံးလေ့ရှိတယ်။

```
try:
    ...
    some_fun()
```



```
...
```

```
except:
```

```
...
```

```
finally:
```

```
...
```

```
...
```

```
try:
    ...
    some_fun()
```



```
...
```

```
except:
```

```
...
```

```
finally:
```

```
...
```

```
...
```

Exception မဖြစ်တဲ့ အခါနဲ့ ဖြစ်တဲ့အခါ try...except...finally အလုပ်လုပ်ပုံ နှိုင်းယှဉ်ပြထားတာပါ။ မီးခိုးရောင် အဖျော့ ဘလောက်တွေက လုပ်ဆောင်မဲ့ ဘလောက်တွေပါ။ မီးခိုးရင့်ရောင် ဘလောက်တွေကိုတော့ လုပ်ဆောင်မှာ မဟုတ်ပါဘူး။ စောစောကဖော်ပြခဲ့သလို early return ဖြစ်မယ်ဆိုလည်း finally အပိုင်း လုပ်ဆောင်ပြီးမှပဲ return ဖြစ်မယ်။ finally မဟုတ်တဲ့ အခြား ဘလောက်တွေကတော့ ၎င်းဘလောက် မတိုင်မီ early return ဖြစ်သွားရင်တော့ လုပ်ဆောင်မှာ မဟုတ်ပါဘူး။

## ၁.၅ Built-in and User-defined Exceptions

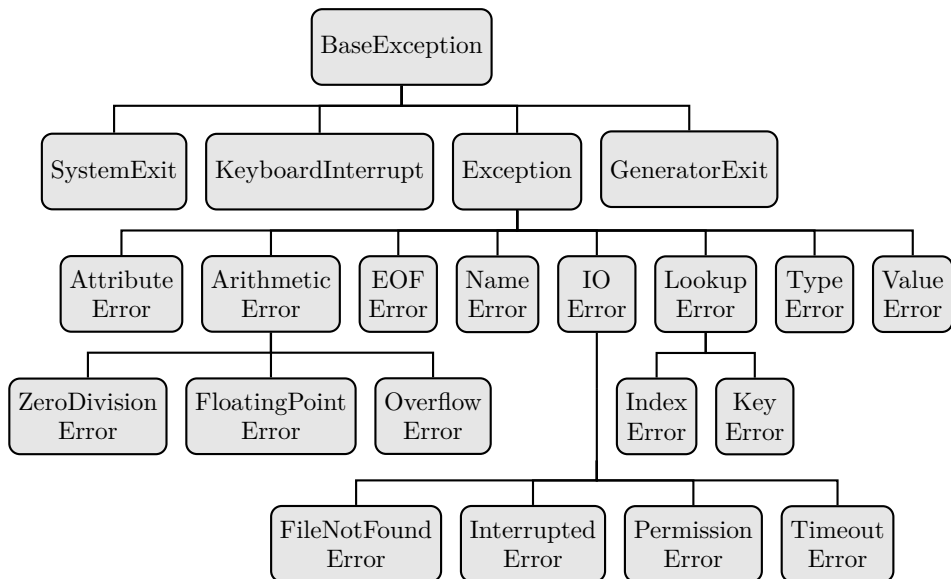
Exception ဆိုတာ ပရိုဂရမ်တစ်ခု ဖြစ်ရိုးဖြစ်စဉ် စီးဆင်းရာ လမ်းကြောင်းကို အနှောက်အယှက် အဟန့်အတား ဖြစ်စေတဲ့ ‘ပုံမှန်မဟုတ်တဲ့ အဖြစ်အပျက်’ လို့ အကြမ်းဖျဉ်း ပြောနိုင်ပါတယ်။ ဒီလို အဖြစ်ပျက်မျိုးတွေက တစ်မျိုးတည်း မဟုတ်တာ သေချာပါတယ်။ ဖိုင်တစ်ခု ဖွင့်တဲ့အခါ အဲ့ဒီဖိုင်က မရှိတာဖြစ်နိုင်သလို၊

ဖိုင်ကရှိပေမဲ့ ခွင့်ပြုချက် (permission) မပေးထားလို့ ဖွင့် မရတာလည်း ဖြစ်နိုင်တယ်။ ကွန်ပျူတာ မမ်မိုရီ မလုံလောက်တာ၊ နက်ဝပ်ချိတ်မရတာ၊ ဆာဗာပြဿနာဖြစ်ပြီး ဒေါင်းနေတာ၊ array index ဘောင်ကျော်သွားတာ၊ သုညနဲ့ စားမိတာ (division by zero)၊ ပုံမှန်မဟုတ်တဲ့ operating system signal စတဲ့ ပြဿနာ တစ်ခုမဟုတ် တစ်ခု ပရိုဂရမ်အလုပ်လုပ်နေစဉ် ရံဖန်ရံခါ ကြုံတွေ့ရတတ်ပါတယ်။

ဒီလို အကြောင်း အမျိုးမျိုးကြောင့် ဖြစ်ပေါ်နိုင်တဲ့ exception အမျိုးမျိုးအတွက် Python မှာ သက်ဆိုင်ရာ built-in exception ကလပ်စ် အသီးသီး ပါရှိပါတယ်။ [ပုံ (၁.၂)]။ Exception အားလုံးဟာ BaseException ရဲ့ subclass တွေပါ။ Built-in exception တွေ အပြင် လိုအပ်ရင် ကိုယ်တိုင် သတ်မှတ်ချင်လည်း ရတယ်။ User-defined exception တွေက BaseException ကနေ တိုက်ရိုက် inherit မလုပ်ရဘူး။ Exception ကလပ်စ်ကို inherit လုပ်ရပါမယ်။ ဥပမာ

```
class InvalidAgeException(Exception):
    """Raised when age is not valid"""
    pass

class BalanceNotEnoughException(Exception):
    """Raised when account balance is not enough"""
    pass
```



ပုံ ၁.၂ Exception Class Hierarchy (အပြည့်အစုံ မဟုတ်ပါ။ ချန်ခဲ့တာတွေရှိသေးတယ်)

## ၁.၆ Handling Multiple Exception

ရှေ့ပိုင်း exception handling ဥပမာတွေမှာ exception တစ်မျိုးတည်းကိုပဲ handle လုပ်တာတွေရမှာပါ။ except အပိုင်းမှာ သတ်မှတ်ထားတဲ့ exception ကလပ်စ် အမျိုးအစားကိုပဲ အဲ့ဒီဘာလောက်က handle လုပ်ပေးမှာ ဖြစ်တယ်။

```
try:
    ...
```



```
except IOError as e:
    ...
```

ဒီတိုင်းဆိုရင် IOError ကိုပဲ handle လုပ်မှာ ဖြစ်ပြီး ValueError (သို့) အခြား IOError instance မဟုတ်တဲ့ exception တွေကို handle မလုပ်ပါဘူး။ ValueError ကိုလည်း handle လုပ်မယ်ဆိုရင် အခုလို except ဘလောက်တစ်ခု ထပ်ဖြည့်နိုင်ပါတယ်။

```
try:
    ...
except IOError as e:
    # handle IOError here
except ValueError as e:
    # handle ValueError here
```

ဒီနေရာမှာ IOError နဲ့ ValueError တို့ဟာ is-a relationship မရှိတဲ့အတွက် except ဘလောက် နှစ်ခု အထက်အောက် ဖလှယ်လို့ရတယ်။ (IOError နဲ့ ValueError ကြားမှာ superclass/subclass အပြန်အလှန် ဆက်စပ်မှု မရှိတာကို သတိပြုပါ။)

```
try:
    ...
except ValueError as e:
    # handle ValueError here
except IOError as e:
    # handle IOError here
```

စောစောကနဲ့ အစီအစဉ်က ပြောင်းပြန်ဆိုပေမဲ့ ဖြစ်ပေါ်တဲ့ exception နဲ့ ကိုက်ညီတဲ့ except အပိုင်း က အလုပ်လုပ်မှာပါ။ ဘယ်ဟာ အရင်လာလာ အရေးမကြီးဘူး။

အောက်ပါ fun\_c က FileNotFoundError (သို့) PermissionError တက်နိုင်ပါတယ်။ ဒီ exception နှစ်ခုလုံးက IOError ရဲ့ subclass ဖြစ်တာကိုလည်း သတိပြုပါ [ပုံ (၁.၂)]။

```
def fun_c():
    print('Starting fun_c...')
    if random.uniform(0.0, 1.0) <= 0.25:
        raise FileNotFoundError("File not found!")
    if random.uniform(0.0, 1.0) > 0.75:
        raise PermissionError("No permission!")
    print('fun_c ends!')
```

ပုံတစ်မျိုးစီနဲ့ သီးခြား handle လုပ်မယ်ဆိုရင် အခုလို

```
def fun_b():
    try:
        fun_c()
    except FileNotFoundError as e:
        print("Handle FileNotFoundError")
    except PermissionError as e:
        print("Handle PermissionError")
```

အကယ်၍ exception နှစ်ခုလုံးကို ပုံစံတစ်မျိုးတည်း handle လုပ်မယ်ဆိုရင်တော့ အခုလို

```
def fun_b1():
    try:
        fun_c()
    except (FileNotFoundError, PermissionError) as e:
        print("Handle both FileNotFoundError and PermissionError")
```

ရေးရပါမယ်။ ဝိုက်ကွင်းထဲမှာ handle လုပ်မဲ့ exception တွေကို ထည့်ပေးပါတယ်။ နောက်တစ်နည်းက ဒီ exception နှစ်ခုရဲ့ superclass ဖြစ်တဲ့ IOError ကို handle လုပ်တာပါ။

```
def fun_b1():
    try:
        fun_c()
    except IOError as e:
        print("Handle both FileNotFoundError and PermissionError")
```

ဒီလိုဆိုရင်တော့ IOError အပါအဝင် သူ့ subclass exception အားလုံး handle လုပ်မှာဖြစ်တယ်။ ပုံ (၁.၂) class hierarchy အရ InterruptedError နဲ့ TimeoutError တို့ကိုပါ handle လုပ်မှာပါ။ ဖော်ပြခဲ့တဲ့ handling နည်းတွေကို ပေါင်းစပ်ပြီး လိုအပ်သလို အသုံးပြုနိုင်ပါတယ်။ ဆက်စပ်မှု ရှိတဲ့ exception တွေကို တစ်ပေါင်းတည်း handle လုပ်တာ၊ အမျိုးအစား တစ်ခုချင်းအလိုက် သီးခြား handle လုပ်တာ၊ ဒါမှမဟုတ် တချို့ကိုတော့ ရွေးထုတ်ပြီး ကျန်တာတွေကို ယေဘုယျပုံစံတစ်မျိုးနဲ့ handle လုပ်တာ စသည့်ဖြင့် လိုချင်တဲ့အတိုင်း ရအောင် အသေးစိတ် ချိန်ညှိပေးလို့ ရပါတယ်။ စမ်းကြည့်ရုံ သက်သက် တမင်ဖန်တီးထားတဲ့ အောက်ပါ ဥပမာကို လေ့လာကြည့်ပါ။

```
try:
    fun_d()
except InterruptedError as e:
    print(e)
except TimeoutError as e:
    print(e)
except (FileNotFoundError, PermissionError) as e:
    print(e)
except IOError as e:
    print(e)
except ArithmeticError as e:
    print(e)
```

InterruptedError နဲ့ TimeoutError ကို သီးခြား handle လုပ်ထားတယ်။ FileNotFoundError နဲ့ PermissionError က ပေါင်းထားတယ်။ ဒီ လေးခု မဟုတ်တဲ့ ကျန်တဲ့ အခြား IOError အားလုံး အတွက် အောက်ဆုံး မတိုင်ခင် except က တာဝန်ယူမယ်။ နောက်ဆုံးမှာ ZeroDivisionError, OverflowError စတဲ့ ArithmeticError အားလုံးကို ခြုံပြီး handle လုပ်တယ်။ fun\_d ကို အောက်မှာ ကြည့်ပါ။ စမ်းကြည့်ချင်တဲ့ exception တက်အောင် သက်ဆိုင်ရာ ဂဏန်းရိုက်ထည့်ရုံပဲ။

```
def fun_d():
    val = int(input("Enter an int: "))
    if val == 1:
```

```

        raise TimeoutError("Timeout")
    if val == 2:
        raise InterruptedError("Interrupted")
    if val == 3:
        raise FileNotFoundError("File not found")
    if val == 4:
        raise PermissionError("Not permitted")
    if val == 5:
        raise FileExistsError("File already exists")
    if val == 6:
        raise FloatingPointError("Floating point error")
    if val == 7:
        x = 10/0          # will cause ZeroDivisionError
    if val == 8:
        x = 2.0 ** 5000   # will cause OverflowError

```

## except ဘလောက် အစီအစဉ်

Subclass တွေထဲက တချို့ကိုပဲ ရွေး handle လုပ်ပြီး ကျန်တာတွေကို superclass instance အနေနဲ့ ခြုံပြီး handle လုပ်တဲ့အခါ except ဘလောက်တွေ အထက်အောက် စီစဉ်ထားတာကို သတိထားရပါမယ်။ Subclass ကို အရင် handle လုပ်ပြီး superclass ကို subclass အောက်မှာ လုပ်ရပါမယ်။ အခုလို handle နည်းလမ်းမမှန်ပါဘူး။

```

try:
    fun_c()
    fun_c1()
except IOError as e:
    print("Handle IOError")
except (FileNotFoundError) as e:
    print("Handle FileNotFoundError")

```

FileNotFoundError က IOError ရဲ့ subclass ။ ဒါကြောင့် FileNotFoundError တက်ခဲ့ရင် အပေါ် IOError အတွက် except ဘလောက်က အရင် handle လုပ်ပါလိမ့်မယ်။ သူ့ကို သီးခြား handle လုပ်ဖို့ ရည်ရွယ်တဲ့ အောက်က except ကို ဘယ်တော့မှ လုပ်ဆောင်မှာ မဟုတ်တော့ဘူး။ ဒီလိုမှပဲ လိုချင်တဲ့အတိုင်း အမှန်ဖြစ်မှာပါ။

```

try:
    fun_d()
except (FileNotFoundError) as e:
    print("Handle FileNotFoundError")
print("Handle IOError and all its subclasses "
      "except FileNotFoundError")

```

```

try:
    fun_d()
except Exception as e:

```

```

    print("Handle all Exception")
except (FileNotFoundError) as e:
    print("Handle FileNotFoundError")
except IOError as e:
    print("Handle all IOError")

```

ဒါလည်း သိပ်တော့ မဟုတ်သေးဘူး။ Exception က IOError ရဲ့ superclass ဆိုတော့ အောက်ဆုံးမှာ ထားသင့်တယ်။

### နိဂုံး

ဒီအခန်းမှာ exception handling နဲ့ ပတ်သက်ပြီး အခြေခံအဆင့် သိသင့်သိထိုက်တဲ့ သဘောတရားတွေကို ထည့်သွင်းဖော်ပြပေးထားပါတယ်။ အတွေ့အကြုံ သိပ်မရှိသေးတဲ့ သူတွေအတွက် အပြည့်အဝနားလည်ဖို့ အခက်အခဲ ရှိနိုင်ပါတယ်။ သဘောတရား နားလည်ရုံနဲ့လည်း မရသေးဘူး။ လက်တွေ့ အခြေအနေတွေမှာ အသုံးချတတ်ဖို့လည်း အရေးကြီးတယ်။ လုပ်သက်အတွေ့အကြုံ ရင့်လာတာနဲ့အမျှ ပိုပြီးနားလည်လာမှာပါ။ အခြေအနေ၊ အချိန်အခါ၊ လိုအပ်ချက်ပေါ် မူတည်ပြီးတော့လည်း သင့်တော်တဲ့ နည်းလမ်းရွေးချယ် အသုံးချတတ်လာမှာပါ။ ပြီးစလွယ် ဖြစ်ကတတ်ဆန်း မလုပ်ဘဲ တတ်နိုင်သမျှ အကောင်းဆုံးဖြစ်အောင် စဉ်ဆက်မပြတ် ဆင်ခြင်စဉ်းစား သုံးသပ်ဖို့၊ ပိုကောင်းသည်ထက် ကောင်းမဲ့ နည်းလမ်းကို အမြဲရှာဖွေနေဖို့တော့ လိုပါလိမ့်မယ်။



# အခန်း ၂

## ဒေတာဘေ့စ်များနှင့် ဆက်သွယ်ဆောင်ရွက်ခြင်း

ဒေတာဘေ့စ်တွေဟာ ကနေ့ခေတ် information system အားလုံးရဲ့ အဓိကကျောရိုးလို့ ဆိုနိုင်ပါတယ်။ ၎င်းတို့ဟာ web application တွေမှာ အသုံးပြုသူ ကိုယ်ရေးအချက်အလက်ကနေ ဘဏ္ဍာရေးဆိုင်ရာ အဖွဲ့အစည်းကြီးတွေ ငွေဝင်ငွေထွက် စာရင်းအထိ အရာအားလုံး သိမ်းဆည်းပေးတဲ့ စနစ်တွေ ဖြစ်တယ်။ ကျန်းမာရေး၊ စီးပွားရေး၊ ပညာရေး၊ ဘဏ္ဍာရေး စတဲ့ ကဏ္ဍ အားလုံးမှာ အချက်အလက်တွေ ထိထိရောက်ရောက် သိမ်းဆည်း စီမံနိုင်ဖို့အတွက် ဒေတာဘေ့စ်တွေက မရှိမဖြစ်ပါပဲ။ အရေးပါတဲ့ ဒီလို ကဏ္ဍတွေမှာ လုပ်ငန်းအသီးသီး မှန်ကန်တိကျ၊ နောက်ဆုံးရ အချက်အလက်တွေနဲ့ informed decision ချနိုင်ဖို့ အဓိကဆောင်ရွက်ပေးတဲ့ စနစ်တွေလည်း ဖြစ်တယ်။

ဒီအခန်းမှာ Python နဲ့ ဒေတာဘေ့စ် ချိတ်ဆက်အသုံးပြုပုံကို လေ့လာကြမှာပါ။ အခြေခံ ဒေတာဘေ့စ် concept တွေကိုတော့ ဒီစာအုပ်မှာ အကျဉ်းချုံးလောက်ပဲ ဖော်ပြပေးနိုင်မယ်။ ပရော်ဖက်ရှင်နယ် အဆင့် ဒေတာဘေ့စ် ပရိုဂရမ်မင်း အတွက်ဆိုရင် တချို့အပိုင်းတွေကို ထဲထဲဝင်ဝင် ဆက်လက် လေ့လာရပါလိမ့်မယ်။ ကိုးကားစာအုပ်တွေ နောက်ဆုံးမှာ ကြည့်နိုင်ပါတယ်။

### ၂.၁ Database Management Systems

‘ဒေတာဘေ့စ်’ ဆိုတာ အချက်အလက် အမြောက်အများ ရေရှည်သိမ်းဆည်းပေးတဲ့ စနစ်လို့ အကြမ်းဖျဉ်း ပြောနိုင်ပါတယ်။ သာမန်အားဖြင့် ရေရှည်သိမ်းထားချင်ရင် ဖိုင်စနစ် သုံးလို့ရပေမဲ့ ဒေတာ များလာတဲ့အခါ အဆင်မပြေနိုင်တော့ဘူး။ ပြန်လည်ရှာဖွေရတာ၊ ထုတ်ယူရတာ၊ အမြဲတမ်း မှန်ကန်ကိုက်ညီနေအောင် ထိန်းသိမ်းရတဲ့ ကိစ္စတွေအတွက် ပြဿနာရှိလာတယ်။ Database Management Systems (DBMS) တွေကို ဒီအခက်အခဲတွေ ဖြေရှင်းပေးဖို့ တီထွင်ခဲ့ကြတာပါ။ အချက်အလက် မှန်ကန်တိကျခြင်း၊ လုံခြုံမှုရှိခြင်းနှင့် အလွယ်တကူ access လုပ်နိုင်ခြင်းအတွက် DBMS တွေမှာ ဦးစားပေး ထည့်သွင်း စဉ်းစားထားတယ်။ ဒေတာပမာဏ အများအပြား စနစ်တကျ ထိထိရောက်ရောက် စုဆောင်း၊ သိမ်းဆည်း၊ စီမံဖို့အတွက် အားကိုးအားထားပြုရတဲ့ စနစ်တွေလို့ ဆိုရမယ်။

### သမိုင်းအကျဉ်း

ဒေတာဘေ့စ်တွေရဲ့ မူလအစ concept ဟာ IBM ကုမ္ပဏီက IMS (Information Management System) လို့ စနစ်တွေ တည်ဆောက်ခဲ့တဲ့ ၁၉၆၀ ခုနှစ်တွေလောက်ကို ပြန်သွားနိုင်တယ်။ အဲ့ဒီစနစ်တွေက hierarchical ဖြစ်တယ်။ ဆိုလိုတာက ဒေတာသိမ်းတဲ့ စထရက်ချာက သစ်ပင်လိုပဲ၊ အပင်ရဲ့ အမြစ်၊ အရွက်၊ အကိုင်းအခက်တွေ ဆက်စပ်နေသလိုပုံစံနဲ့ အချက်အလက်တွေကို သိမ်းတယ်။ Parent-child

relationship နဲ့ သိမ်းတာလိုလည်း ဆိုနိုင်တယ်။ ၁၉၇၀ ခုနှစ်တွေမှာတော့ Edgar F. Codd က ယနေ့ခေတ် Relational Database Management System (RDBMS) ရဲ့ အခြေခံအုတ်မြစ် ဖြစ်လာတဲ့ Relational Data Model ကို စတင်မိတ်ဆက်ခဲ့တယ်။ Relational model မှာက ဒေတာသိုလှောင်သိမ်းဆည်းဖို့ table တွေကို အသုံးပြုပြီး SQL (Structured Query Language) လို့ ခေါ်တဲ့ programming language ကို ထောက်ပံ့ပေးပါတယ်။

## SQL Language

SQL ဟာ table ဒေတာ အမြောက်အများကနေ မိမိစူးစမ်းလိုတဲ့ အချက်အလက်ကို အလွယ်တကူ ထုတ်ယူ (သို့) မေးမြန်းလိုရအောင် ကူညီထောက်ပံ့ပေးဖို့ အဓိကရည်ရွယ်တယ်။ “ကေသီ ဒီနှစ် ဇွန်လ စာမေးပွဲမှာ ဘာသာရပ်အသီးသီး ရမှတ်ဘယ်လောက်လဲ” လို့ ခပ်ရိုးရိုး မေးခွန်းကနေ “ဘယ်ကျောင်းသူ ကျောင်းသား တွေ စာမေးပွဲအားလုံးမှာ သင်္ချာရမှတ် ၉၅ မှတ်အထက် သုံးနှစ်ဆက်တိုက် ရကြလဲ” ဆိုတဲ့ အတော်လေး ရှုပ်ထွေးတဲ့ query မျိုးတွေထိ မြန်ဆန်ထိရောက်စွာ လုပ်ဆောင်ပေးနိုင်ပါတယ်။

အချက်အလက် ထုတ်ယူတာအပြင် ဒေတာဘေ့စ် အသစ်ဆောက်တာ၊ table ဆောက်တာ၊ ပြန်ဖျက် တာ စတဲ့ကိစ္စတွေကိုလည်း SQL နဲ့ပဲ လုပ်ရပါတယ်။ Table မှာ record အသစ်ထည့်တာ၊ ရှိပြီးသား record ကို update လုပ်တာ၊ ဖျက်ပစ်တာ စတာတွေအတွက်လည်း SQL ကိုပဲ သုံးရတာပါ။

SQL ဟာ RDBMS အားလုံးမှာ အသုံးပြုနိုင်တဲ့ standard language တစ်ခုလည်းဖြစ်တယ်။ ဆိုလိုတာက ဘယ် RDBMS ကိုပဲ သုံးသုံး၊ SQL တစ်မျိုးတည်းကိုပဲ သုံးရမှာပါ။ RDBMS တစ်ခု မှာ သူ့ကိုယ်ပိုင် ချဲ့ထွင်ထားတဲ့ အပိုင်းတွေ အနည်းအကျဉ်း ရှိကြပေမဲ့ SQL standard ကို ရာနှုန်းပြည့် မဟုတ်တောင် အဲ့လောက်နီးနီး လိုက်နာထားကြတဲ့အတွက် ပြောပလောက်အောင် မကွာကြဘူး။ SQL သာနားလည်ပါစေ၊ ဘယ်ဒေတာဘေ့စ်နဲ့မဆို အလုပ်ဖြစ်တယ်လို့ ပြောရင်လည်း မမှားဘူး။

## Client-Server Applications

တစ်ချိန်မှာ တစ်ယောက်ပဲ သုံးလိုရတဲ့ ဆော့ဖ်ဝဲတွေနဲ့ မတူတာက DBMS တွေဟာ တစ်ယောက်မက တပြိုင်နက် သုံးလိုရတဲ့ ဆာဗာ (server) ဆော့ဖ်ဝဲတွေ ဖြစ်ပါတယ်။ တပြိုင်နက် ဝင်ရောက်လာတဲ့ အသုံးပြု သူတွေရဲ့ တောင်းဆိုချက်တွေကို ဖြည့်ဆည်းဖို့အတွက် ကိုင်တွယ်ဆောင်ရွက် ပေးနိုင်စွမ်း ရှိတယ်။

တစ်ခုထက်ပိုတဲ့ client တွေက ဆာဗာတစ်ခုနဲ့ ချိတ်ဆက်အသုံးပြုတဲ့ ဆော့ဖ်ဝဲစနစ်မျိုးကို Client-Server Application လို့ ခေါ်တယ်။ Client ဆိုတာ အသုံးပြုသူ user (သို့) ၎င်းအသုံးပြုတဲ့ ပရိုဂရမ် ကို ဆိုလိုတာ။ ဒေတာဘေ့စ် application အများစုဟာ Client-Server Application တွေပါ။ Web Application တွေဟာ Client-Server Application ဖြစ်ပြီး ဒေတာသိမ်းဖို့အတွက် နောက်ကွယ်က RDBMS တစ်ခုနဲ့ ချိတ်ဆက်ထားရလေ့ရှိတယ်။

## ၂.၂ PostgreSQL

ဒီစာအုပ်မှာ PostgreSQL RDBMS အသုံးပြုပါမယ်။ PostgreSQL ဘယ်လို အင်စတောလ်လုပ်မလဲ စာမျက်နှာ ၂၉ နောက်ဆက်တွဲ (က) မှာ ကြည့်ပါ။ psql ဖွင့်ပြီး root user (postgres) အနေနဲ့ PostgreSQL ဒေတာဘေ့စ်ကို ချိတ်ဆက်ထားပါ။

### ဒေတာဘေ့စ် အသစ်ဆောက်ခြင်း

ဒေတာဘေ့စ် အသစ်တစ်ခု ဆောက်မယ်ဆိုရင် CREATE DATABASE SQL ကွန်မန်း သုံးပါတယ်။

```
CREATE DATABASE database_name;
```

၂၁

```
SQL Shell (psql)

postgres=# CREATE DATABASE students;
CREATE DATABASE
postgres=#
```

ပုံ ၂.၁

```
Select SQL Shell (psql)

postgres=# \l

              List of databases
  Name | Owner | Encoding | Collate | Ctype | ICU Locale | Locale Provider | Access privileges
-----+-----+-----+-----+-----+-----+-----+-----
 postgres | postgres | UTF8 | English_United States.1252 | English_United States.1252 | English_United States.1252 | English_United States.1252 | 
 students | postgres | UTF8 | English_United States.1252 | English_United States.1252 | English_United States.1252 | English_United States.1252 | 
 template0 | postgres | UTF8 | English_United States.1252 | English_United States.1252 | English_United States.1252 | English_United States.1252 | 
 template1 | postgres | UTF8 | English_United States.1252 | English_United States.1252 | English_United States.1252 | English_United States.1252 | 
(4 rows)

postgres=#
```

ပုံ ၂.၂

SQL language ဟာ စာလုံး အကြီးအသေး မခွဲဘူး။ ဒီစာအုပ်မှာ SQL keyword တွေဆိုရင် အကွရာ အကြီးနဲ့ ရေးပါမယ်။ Database, table, column, function စတာတွေရဲ့ နံမည်တွေက အကွရာ အသေးနဲ့ ဖြစ်မယ်။ student ဒေတာဘေ့စ် အတွက် ဒီ SQL ကို

```
CREATE DATABASE students;
```

psql ကနေ run ပေးပါ ပုံ (၂.၁)။ SQL စတိတ်မန့် တစ်ကြောင်း အဆုံးမှာ ဆီမီးကော်လံ (;) ထည့် ပေးရပါမယ်။

psql ကနေ \l (သို့) \list ကွန်မန်းနဲ့ PostgreSQL မှာ ရှိတဲ့ ဒေတာဘေ့စ်တွေကို ထုတ် ကြည့်နိုင်ပါတယ်။ \l က SQL မဟုတ်ဘူး။ psql သီးသန့် ကွန်းမန်းတစ်ခုဖြစ်တာကြောင့် ဒီကွန်းမန်းကို ; မထည့်ဘဲ run ရပါမယ်။ \l run လိုက်ရင် စာရင်းထဲမှာ students ဒေတာဘေ့စ် တွေ့ရမှာပါ ပုံ (၂.၂)။

psql မှာ ဒေတာဘေ့စ် ပြောင်းချိတ်မယ်ဆိုရင် \c (သို့) \connect နဲ့ ပြောင်းရပါမယ်။ psql သီးသန့် ကွန်းမန်းပါ။ Backslash (\) စရင် psql ကွန်းမန်းလို့ မှတ်နိုင်တယ်။ students ဒေတာဘေ့စ် ကို အခုလို connect လုပ်ပါ

```
\c students
```

psql က ဒီလို ပြပါလိမ့်မယ်

```
postgres=# \c students
You are now connected to database "students" as user "postgres".
students=#
```



## Table ဆောက်ခြင်း

CREATE TABLE က table ဆောက်တဲ့ SQL ကွန်မန်းပါ။ students ဒေတာဘေ့စ်ထဲမှာ student table အောက်ပါအတိုင်း ဆောက်ပါမယ်

```
CREATE TABLE student (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    age INT,
    grade VARCHAR(2)
);
```

psql မှာ SQL run ရင် လက်ရှိချိတ်ထားတဲ့ ဒေတာဘေ့စ်ကို အဲဒီ SQL ပေးပို့ လုပ်ဆောင်ခိုင်းပါတယ်။ အခုချိတ်ထားတာ students ဒေတာဘေ့စ်ဆိုတော့ table ကို အဲဒီ ဒေတာဘေ့စ်ထဲ ဆောက်ပေးသွားမှာပါ။ ဒီ table မှာ id, name, age နဲ့ grade column လေးခုရှိမယ်။

Column တစ်ခုစီမှာ data type ရှိရပါမယ်။ VARCHAR(100) က အများဆုံး ကာရက်တာ အလုံးတစ်ရာ သိမ်းဆည်းနိုင်တယ်။ သိမ်းတဲ့ ကာရက်တာ အရေအတွက်ပေါ် မူတည်ပြီး နေရာယူတာ အနည်းအများ ကွာတယ်။ ငါ့သိမ်းရင် ငါးခုစာ၊ ဆယ်ခုသိမ်းရင် ဆယ်ခုစာပဲ နေရာကုန်မှာပါ။ အမြဲ အလုံး တစ်ရာစာ နေရာကုန်တာ မဟုတ်ဘူး။ VARCHAR(2) ဆိုရင် အများဆုံး ကာရက်တာ နှစ်လုံး သိမ်းလို့ရမယ်။ SQL VARCHAR က Python str နဲ့ အလားတူတယ်။ INT ကတော့ integer ပါ။

id column က ထူးခြားပြီး နည်းနည်းပိုရှင်းပြဖို့ လိုတယ်။ SERIAL က data type အနေနဲ့ INT နဲ့ တူတူပဲ။ သူ့ရဲ့ ထူးခြားချက်က ဂဏန်းတွေကို အစဉ်အတိုင်း တစ်ခုပြီးတစ်ခု ထုတ်ပေးနိုင်တာပါ။ 1, 2, 3, ... စသည်ဖြင့် နောက်ဆုံးတန်ဖိုးကို အလိုအလျောက် တစ်တိုးတိုးပြီး ထုတ်ပေးသွားမှာ ဖြစ်တယ်။ id column က Primary Key လည်းဖြစ်တယ်။ Column တစ်ခုကို Primary Key အဖြစ် ထားချင်ရင် PRIMARY KEY လို့ သတ်မှတ်ရပါမယ်။ Primary Key ဆိုရင် column တန်ဖိုး ထပ် (duplicate) လို့မရဘူး၊ unique ဖြစ်ရပါမယ်။ အခု သိပ်နားမလည်သေးရင်လည်း table မှာ ကျောင်းသား record တွေထည့်တာ ဆက်ကြည့်ရင် ကောင်းကောင်း နားလည်သွားမှာပါ။

## INSERT

Relational Data Model အခြေခံတဲ့ RDBMS တွေဟာ ဒေတာတွေကို table ပုံစံနဲ့ သိမ်းဆည်းတယ်။ ကျောင်းသူ/သား တစ်ယောက်ချင်းစီအတွက် အချက်အလက်ကို student table မှာ row တစ်ခုစီနဲ့ ထည့်သွင်း သိမ်းဆည်းပါမယ်။ Row ကို record လို့လည်း သုံးနှုန်းလေ့ရှိတယ်။ Record အသစ် ထည့်သွင်းမယ်ဆိုရင် SQL INSERT ကို သုံးရပါတယ်။

```
INSERT INTO student (name, age, grade) VALUES ('Amy', 20, 'A');
INSERT INTO student (name, age, grade) VALUES ('Kathy', 22, 'B');
INSERT INTO student (name, age, grade) VALUES ('Waiyan', 21, 'C');
```

အေမီ၊ ကေသီ နဲ့ ဝေယံ ကျောင်းသား သုံးယောက်အတွက် record သုံးခု ထည့်သွင်းတာပါ။ Column နံမည်တွေ ဝိုက်ကွင်းထဲမှာ ထည့်ပြီး အဲဒီ column တွေအတွက် တန်ဖိုးအသီးသီးကို အစဉ်အတိုင်း ထည့်ပေးရပါတယ်။ age နဲ့ grade ရှေ့နောက် ဖလှယ်လိုက်မယ်ဆိုရင် အခုလို

```
INSERT INTO student (name, grade, age) VALUES ('Amy', 'A', 20);
```

ဖြစ်ရမှာပါ။

student table မှာ column က လေးခု ရှိတယ်။ အခု INSERT တွေမှာကျတော့ သုံးခုပဲတွေ့ရပြီး id မပါဘူး။ ဘာကြောင့်လဲ။ INSERT လုပ်တဲ့အခါ SERIAL column အတွက် တန်ဖိုးကို ဒေတာဘေ့စ်က အလိုအလျောက် ထည့်ပေးသွားတာ။ ကိုယ်တိုင်ထည့်ဖို့ မလိုဘူး။ ဒါကြောင့် id column ကို *auto-incrementing* primary key column လို့ ခေါ်တယ်။ Auto-increment ဖြစ်ဖို့ အခြားနည်းလမ်းတွေလည်း ရှိပါတယ်။ SERIAL ကတော့ ဒီကိစ္စအတွက် လွယ်အောင် လုပ်ပေးထားတာပါ။ စောစောက INSERT သုံးကြောင်းကို psql မှာ run ပါ။ အဲဒါ၊ ကေသီ နဲ့ ဝေယံတို့အတွက် record အသီးသီးကို id နံပါတ် 1, 2, 3 အစဉ်နဲ့ student table ထဲ ထည့်သွင်းသွားမှာဖြစ်တယ်။ နောက်ထပ် record တစ်ခု ထပ်ထည့်ရင် id နံပါတ် 4 ဖြစ်မှာပါ။

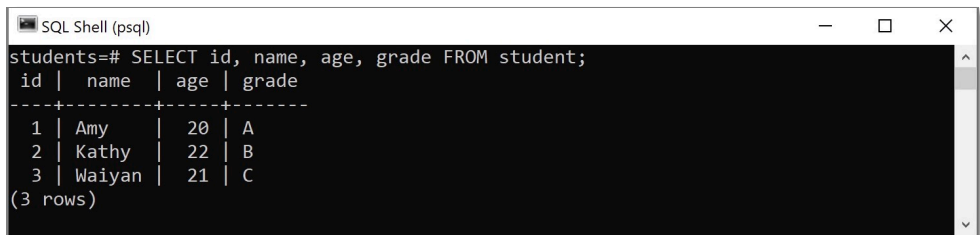
## SELECT

Table ဒေတာတွေ ထုတ်ယူကြည့်ဖို့ အသုံးပြုတဲ့ SQL ဖြစ်ပါတယ်။ Student table ထဲက record အားလုံးကို ကြည့်မယ်ဆို အခုလို

```
SELECT id, name, age, grade FROM student;
```

Table မှာ ရှိသမျှ column အကုန်လုံး ပါချင်ရင် SELECT \* သုံးလို့လည်းရတယ်။ SELECT \* ကို 'Select All' လို့ ဖတ်တယ်။

```
SELECT * FROM student;
```



```
SQL Shell (psql)
students=# SELECT id, name, age, grade FROM student;
 id | name | age | grade 
----+-----+----+-----
  1 | Amy  |  20 | A
  2 | Kathy |  22 | B
  3 | Waiyan | 21 | C
(3 rows)
```

## ပုံ ၂-၃

Column အကုန်မထုတ်ဘဲ ကိုယ်လိုချင်တာပဲ ရွေးပြီး select လုပ်ချင်လည်း ရတယ်။ အောက်ပါတို့ကို psql မှာ စမ်းကြည့်ပါ။

```
SELECT name, grade FROM student;
SELECT name, age, id FROM student;
```

## WHERE

Grade A ရတဲ့ ကျောင်းသားတွေကိုပဲ ရွေးထုတ်ကြည့်မယ် ဆိုပါစို့။ ဒီအတွက် SQL မှာ WHERE ရှိပါတယ်။ ဥပမာ

```
SELECT * FROM student WHERE grade = 'A';
```

WHERE မှာ ဘူလီယန် အိပ်စ်ပရက်ရှင် ပါရပါမယ်။ ရေးပုံရေးနည်း နည်းနည်းကွာပေမဲ့ SQL ဘူလီယန် အိပ်စ်ပရက်ရှင်က Python နဲ့ သဘောတရားအားဖြင့် တူပါတယ်။ grade = 'A' က grade column တန်ဖိုး 'A' နဲ့ ညီလား စစ်တာ။ ညီတဲ့ record တွေကိုပဲ WHERE က စစ်ထုတ်ပေးမှာပါ။ ဒါကြောင့်

အပေါ်က select က record တစ်ကြောင်းပဲ ထွက်မှာပါ။ အဲဒီတစ်ယောက်ပဲ A ရပါတယ်။ အခြား ဘူလီယန် အိပ်စ်ပရက်ရှင်တွေ လက်တွေ့စမ်းကြည့်ဖို့အတွက် အောက်ပါအတိုင်း ကျောင်းသား record သုံးခု ထပ်ထည့်ထားပါ။

```
INSERT INTO student (name, age, grade) VALUES ('Sandy', 19, 'A');
INSERT INTO student (name, age, grade) VALUES ('Thida', 21, 'B');
INSERT INTO student (name, age, grade) VALUES ('Peter', 21, 'B');
INSERT INTO student (name, age, grade) VALUES ('Haymar', 18, NULL);
```

```
--
SELECT * FROM student WHERE grade = 'A' OR grade = 'B';
--
SELECT * FROM student WHERE grade = 'B' AND id <= 5;
--
SELECT * FROM student WHERE grade <> 'C';
--
SELECT * FROM student WHERE grade IS NULL;
--
SELECT * FROM student WHERE grade IS NOT NULL;
```

## UPDATE

## DELETE

## ၂.၃ Database API

### psycopg2

Python programming language အတွက် Psycopg ဟာ popular အဖြစ်ဆုံး PostgreSQL ဒေတာဘေ့စ် adapter ဖြစ်ပါတယ်။ ဒေတာဘေ့စ် adapter ဆိုတာ DBMS နဲ့ programming language ကြား ပေါင်းကူးတံတားအဖြစ် ဆောင်ရွက်ပေးတဲ့ လိုက်ဘရီပဲ ဖြစ်တယ်။

Adapter သုံးတဲ့အခါ DBMS နဲ့ programming language အလိုက် သက်ဆိုင်ရာ adapter ကို ရွေးချယ်ရမှာပါ။ PostgreSQL အတွက် Python မှာ Psycopg 2 နဲ့ Psycopg 3 ရှိမယ်။ အခြားဟာတွေလည်း ရှိပါသေးတယ်။ MySQL အတွက် MySQL Connector/Python သုံးကြတယ်။ Microsoft SQL အတွက်ဆိုရင် pyodbc။

### Connecting to PostgreSQL

Python ကနေတစ်ဆင့် ဒေတာဘေ့စ် အသစ်တစ်ခု ဘယ်လိုဆောက်မလဲ ကြည့်ရအောင်။ စောစောက ပြောတဲ့ psycopg2 adaptor install လုပ်ထားပြီး ဖြစ်ရပါမယ်။

```
import psycopg2

# Connect to the default PostgreSQL database to create
# the new "students" database
conn = psycopg2.connect(
    dbname="postgres",
```

```

    user="postgres",
    password="asdfgh",
    host="localhost",
    port="5432"
)

conn.autocommit = True
cur = conn.cursor()

# Create the "students" database
cur.execute("CREATE DATABASE students")

# Close the initial connection
cur.close()
conn.close()

```

ဒီပရိုဂရမ်ကို ခွဲခြမ်းစိတ်ဖြာ ကြည့်ရအောင်။ psycopg2 မော်ဒျူး အင်ပို့ လုပ်ပါတယ်။ ပြီးတော့ connect ဖန်ရှင်နဲ့ connection ယူတယ်။ ပါရာမီတာ တစ်ခုစီရဲ့ အဓိပ္ပါယ်က

- dbname မှာ ချိတ်ဆက်မဲ့ ဒေတာဘေ့စ်နံမည် ထည့်ပေးရမယ်။ default ဒေတာဘေ့စ်နဲ့ ချိတ်ဆက် မှာဆိုတော့ postgres ပဲ
- user က ဘယ်သူ့အနေနဲ့ ချိတ်ဆက်အသုံးပြုမှာလဲ။ root user ဖြစ်တဲ့ postgres အနေနဲ့ပဲ ချိတ်ဆက်မှာ
- password က ချိတ်ဆက် အသုံးပြုမဲ့သူရဲ့ password ၊ root user postgres ရဲ့ password ထည့်မယ်
- host ကတော့ ဒေတာဘေ့စ် ဆာဗာရဲ့ IP address (သို့) host နံမည်၊ ဒေတာဘေ့စ် ဆာဗာက ကိုယ့်ကွန်ပျူတာမှာပဲဆိုရင် 127.0.0.1 (သို့) localhost ထည့်နိုင်တယ် (အခြားကွန်ပျူတာမှာ run ထားတဲ့ ဒေတာဘေ့စ် ဆိုရင် အဲဒီ ကွန်ပျူတာရဲ့ IP address သို့ domain name ရှိရင် domain name ထည့်ပေးလို့ရတယ်)
- port PostgreSQL ဒေတာဘေ့စ် port နံပါတ်၊ အင်စတောလ်လုပ်တုန်းက ပေးခဲ့တဲ့ port နံပါတ် ပြန်ထည့်ပေးရမယ်

Connect လုပ်တာ အောင်မြင်ရင် connect ဖန်ရှင်က Connection အော့ဘ်ဂျက်တစ်ခု ပြန်ရပါတယ်။ တကယ်လို့ ပြဿနာ တစ်ခုခုကြောင့် connect လုပ်လို့ မရရင်တော့ ဖြစ်ရတဲ့ အကြောင်းအရင်းပေါ် မူတည်ပြီး OperationalError, InternalError စတဲ့ exception တွေ တက်နိုင်တယ်။ ဒီ exception တွေက psycopg2.Error ရဲ့ subclass တွေပါ။ built-in တွေ မဟုတ်ပါဘူး။ psycopg2 သီးသန့် exception တွေပါ။

```
conn.autocommit = True
```

ဒါက ဒေတာဘေ့စ် *auto-commit* mode ကို on လုပ်ပေးဖို့။ ဒေတာဘေ့စ် transaction နဲ့ ဆိုင်တဲ့ setting တစ်ခုဖြစ်ပြီး နောက်ပိုင်းမှာ အသေးစိတ် ရှင်းပြမှာပါ။ PostgreSQL က နဂိုအတိုင်း auto commit mode ကို off လုပ်ထားတယ်။ ဒေတာဘေ့စ် အသစ်ဆောက်တဲ့ CREATE DATABASE လို့ တချို့ SQL စတိတ်မန်တွေက auto commit ကို on လုပ်ပေးရတယ်လို့ လောလောဆယ် သိထားရင် လုံလောက်ပါပြီ။ On ထားရင် SQL စတိတ်မန်တွေ execute လုပ်ပြီးရင် commit ဖန်ရှင် ခေါ်ဖို့ မလိုဘူး။ (နောက် ဥပမာမှာ commit ဖန်ရှင် သုံးထားတာ တွေ့ရမှာပါ)။

SQL Shell (psql)

```
postgres=# \l
```

| Name      | Owner    | Encoding | Collate                    | List of databases<br>Ctype |
|-----------|----------|----------|----------------------------|----------------------------|
| postgres  | postgres | UTF8     | English_United_States.1252 | English_United_States.     |
| students  | postgres | UTF8     | English_United_States.1252 | English_United_States.     |
| template0 | postgres | UTF8     | English_United_States.1252 | English_United_States.     |
| template1 | postgres | UTF8     | English_United_States.1252 | English_United_States.     |

### ပုံ ၂-၄

SQL စတိတ်မန့်တွေ ဒေတာဘေ့ခ်ဆီ ပေးပို့လုပ်ဆောင်စေခြင်း၊ ဒေတာဘေ့ခ်ဆီက ပြန်ရလာတဲ့ ရလဒ်တွေကို အသုံးပြုခြင်း၊ ဒေတာဘေ့ခ် transaction စီမံခြင်း စတဲ့ ကိစ္စတွေအတွက် cursor က အဓိကကျတယ်။ Cursor အော့ဘ်ဂျက်ကို connection ကနေ တစ်ဆင့် အခုလို ယူရပါတယ်

```
cur = conn.cursor()
```

students ဒေတာဘေ့ခ် ဆောက်တဲ့ SQL ကို ဒေတာဘေ့ခ် ဆာဗာဆီ cursor နဲ့ ပေးပို့ လုပ်ဆောင် ခိုင်းရပါမယ်။

```
cur.execute("CREATE DATABASE students")
```

Cursor နဲ့ connection ကို အသုံးပြုပြီးသွားရင် ပိတ်ပေးသင့်တယ်။ Exception handle လုပ် မယ်ဆိုရင် finally ထဲမှာ ပိတ်ရမယ်။ ဒီဥပမာမှာတော့ ရိုးရိုးပဲ ပိတ်ထားပါတယ်။

```
cur.close()
conn.close()
```

ဒီပရိုဂရမ် run ပြီးလို့ ဘာပြဿနာမှမရှိဘဲ အောင်မြင်တယ်ဆိုရင် students ဒေတာဘေ့ခ် ဆောက် ပြီးသွားပါပြီ။ psql မှာ \l ကွန်မန်း run ကြည့်ရင် အခုလို တွေ့ရမှာပါ။ (သို့) pgAdmin

ဒေတာဘေ့ခ်ထဲမှာ table တွေ ဆောက်ပါမယ်။ လက်ရှိအသုံးပြုမဲ့ ဒေတာဘေ့ခ်နဲ့ connection အရင်ယူရမယ်။ students ဒေတာဘေ့ခ်မှာ table ဆောက်မှာ။ ဒီတော့ ချိတ်ဆက်မဲ့ dbname က students ဖြစ်တယ်။ ကျန်တာတွေက ရှေ့ကနဲ့ တူတူပဲ။

```
import psycopg2
```

```
# Now, connect to the "students" database to create the "student" table
conn = psycopg2.connect(
    dbname="students",
    user="postgres",
    password="asdfgh",
    host="localhost",
    port="5432"
)
cur = conn.cursor()
```

```
# Create the "student" table
cur.execute("""
    CREATE TABLE student (
        id SERIAL PRIMARY KEY,
        name VARCHAR(100),
        age INT,
        grade VARCHAR(2)
    )
""")

# Commit changes and close the connection
conn.commit()
cur.close()
conn.close()
```

Table ဆောက်တဲ့ CREATE TABLE စတိတ်မန့်က auto-commit mode ON ဖြစ်ဖြစ်၊ OFF ဖြစ်ဖြစ် run လို့ရတယ်။ PostgreSQL မှာ default က OFF ဖြစ်တယ်။ Connection ယူပြီးတာနဲ့ အခုလို တကူးတက ထည့်ပေးမယ်ဆိုရင်လည်း ပြဿနာတော့ မရှိပါဘူး။

```
conn.autocommit = False
```

Default က OFF ဖြစ်ပြီးသားမို့လို့ လိုတော့မလိုအပ်ဘူး။

Auto-commit OFF ထားရင် SQL စတိတ်မန့် execute လုပ်ပြီး commit လုပ်ပေးဖို့ လိုတယ်။ ဒီအတွက်

```
conn.commit()
```

လုပ်ရပါမယ်။ ဒါ မေ့ကျန်ခဲ့ရင် execute လုပ်ထားတဲ့ SQL က သက်ရောက်မှု ရှိမှာမဟုတ်ပါဘူး။ တစ်နည်းအားဖြင့် student table အမှန်တကယ် မဆောက်ပေးတော့ဘူး။ အကယ်၍ auto-commit OFF ထားရင်တော့ ကိုယ်တိုင် conn.commit() လုပ်မပေးရဘူး၊ ဒေတာဘေ့ခ်က SQL ကွန်မန်း တစ်ခု execute လုပ်ပြီးတိုင်း အလိုအလျောက် commit လုပ်ပေးမှာပါ။ OFF ထားရင် conn.commit() လုပ်ပေးရမယ်၊ ON ဆိုရင် မလုပ်ပေးရဘူး၊ ဒီလို မှတ်ထားလို့ရတယ်။



# နောက်ဆက်တွဲ က

*PostgreSQL ဒေတာဘေ့စ် ဆာဗာဆော့ဖ်ဝဲ ထည့်သွင်းခြင်း*