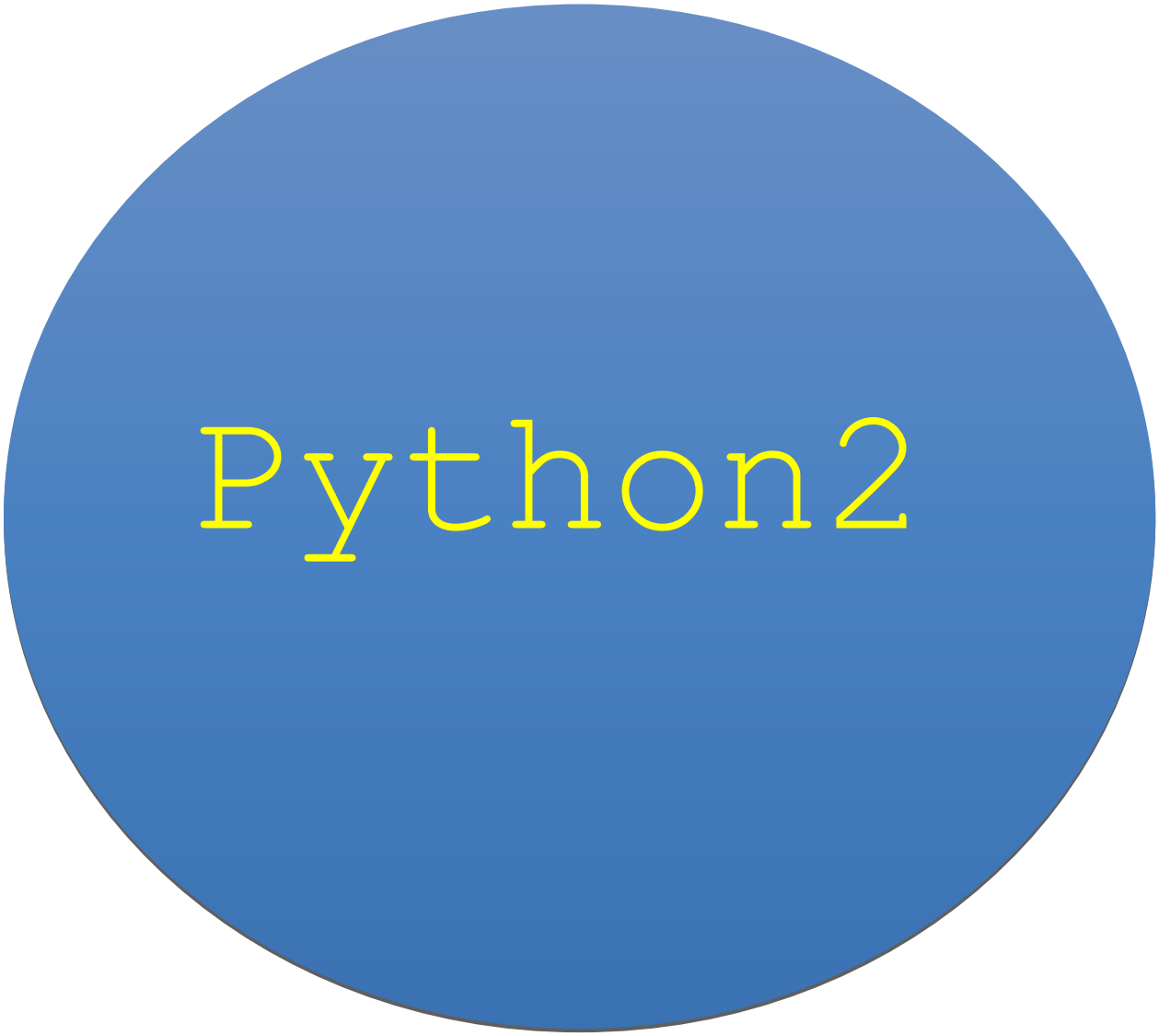



- Python 2 အကြောင်း ဘာသာပြန် note တွေ
- အကြောင်းအရာ ၄၀ ကျော်
- စာမျက်နှာ ၁၆၀ ကျော်ခန့်

Raw Version translated by Han Phyo Oo

# Contents

# Chapter 1

## Introduction

Python ကိုပြောရင် ABC programming language ကနေစပြောရပါမယ်။ ဘာလို့လဲဆိုတော့ python design က ABC ရဲ့လွှမ်းမိုးမှုတွေပါနေလို့ဖြစ်ပါတယ်။

Python ကို 1980 လောက်မှာ concept တစ်ခုအနေနဲ့တော့ရှိနေပါပြီ။ Guido van Rossum ကအဲ့ဒီအချိန်တုန်းက CWI မှာရှိတဲ့ Amoeba လို့ခေါ်တဲ့ distributed operation system Project တစ်ခုမှာ လုပ်ကိုင်နေပါတယ်။ ABC နဲ့ရေးတာပါ။ Bill Venners (January 2003) နဲ့ အင်တာဗျူးမှာ Guido van Rossum က " **ABC နဲ့ပတ်သက်ပြီး အတွေ့အကြုံတွေ မရှင်းလင်းတာတွေရှိတယ်ဗျ။ ဒါကြောင့် ABC ရဲ့ ကောင်းတဲ့အချက်တွေကို ABC ရဲ့ ပြဿနာတွေ မရှိပဲ အသုံးပြုနိုင်မယ့် scripting language တစ်ခု ဒီဇိုင်း လုပ်ဖို့ဆုံးဖြတ်ထားတယ်။ ခုစရေးနေပြီ။ Virtual Machine ရိုးရိုးလေးတစ်ခုရယ် parser တစ်ခုရယ် run time တစ်ခုရယ် လုပ်ထားတယ်။ ABC ထဲက ကျွန်တော်ကြိုက်တဲ့ အချက်တွေကို ကျွန်တော့ နည်း ကျွန်တော့ ဟန်ရေးထားတယ်။ syntax ရိုးရိုးလေးတွေသုံးတယ်။ statement တွေ group ဖွဲ့ဖို့ တွန့်ကွင်းတွေ begin-end block တွေအစား identation သုံးထားတယ် ပြီးတော့ data type ကောင်းတာလေးတွေလုပ်ထားတယ် a hash table ( or dictionary, as we call it), a list, strings, and numbers.”**

ဒါဆို pythonဆိုတဲ့နာမည်ကရော....? မြွေလို့ထင်ကြပေမယ့် အမှန်တော့ British ဟာသနဲ့ဆိုင်ပါတယ်။ "Guido van Rossum က ၁၉၉၆ မှာ ပြောပြထားတယ် " လွန်ခဲ့တဲ့ခြောက်နှစ်လောက် ဒီဇင်ဘာ 1989မှာ ကျွန်တော် က ခရစ်စမတ် သတင်းပတ်မှာ စွဲစွဲမြဲမြဲလေးလုပ်မယ့် hobby programming project လေးရှာနေတာ ။ ရုံးကလည်းပိတ် ...အိမ်မှာလည်း ကွန်ပျူတာရှိတော့ လုပ်စရာလည်းသိပ်မရှိဘူးလေ ။ ဒါကြောင့် အရင်တုန်းကစဉ်းစားထားတဲ့ scripting language အသစ်အတွက် Interpreter လေးရေးမယ် လို့ဆုံးဖြတ်လိုက်တယ်။ UNIX/ C hacker တွေအကြိုက် ABC နောက်မျိုးဆက်ပေါ့။ project ကို python လို့နာမည် ပေးလိုက်တယ်။ လူကလည်း နည်းနည်းပေါ့ ပြီးတော့ Monty Python's Flying Circus အဖွဲ့ရဲ့ fan တစ်ယောက်ဖြစ်နေတာလည်းပါပါတယ်။

Chapter 2

## Why Python?

ဘာလို့လဲဆိုတော့အများကြီးပါ ။သင်ရတာလွယ်တယ် powerfull ဖြစ်တယ်။ ပြီးတော့ ကောကောင်းမွန်တဲ့ hight-level datastructure တွေရှိတယ်။ဒါတွေသုံးပြီး ရှုပ်ထွေးတဲ့ လုပ်ငန်းတွေကို c,c++,java တို့ထက်နည်းတဲ့statement တွေနဲ့ရေးနိုင်တယ်။ OOP သုံးရာမှာလည်း java ထက်လွယ် တယ်။

ရှင်းလင်းပြီး ဖတ်ရတာလည်းလွယ်တယ်။ general-purpose high-level programming language တစ်ခုလည်းဖြစ်တယ်။ OOP / Imperative/ fuctional style တွေရေးလို့ရတယ်။ ဘယ် platform မှာပဲဖြစ်ဖြစ် run လို့ရတဲ့အတွက် portable ဖြစ်တယ်။

David Beazley က"ပြောစရာတွေအများကြီးရှိပေမဲ့ ရိုးရိုးလေးပြောရရင် Python က ပျော်စရာကောင်းပြီး productive လည်းဖြစ်ပါတယ်။"

Guido van Rossum က Python ကို Netherlands က Research Institute for Mathematics and Computer Science မှာ စတင်ခဲ့ပါတယ်။ Python ကိုဘာတွေသဘောကျလဲလို့ Linux jornal ကမေးမြန်းရာမှာ " ထင်ရှားတဲ့အချက်ကတော့ ကျွန်တော်programming style နဲ့အံကိုက်ပဲဗျ ။ interpreter နဲ့ interactive လုပ်ဆောင်နိုင်သလို စဆုံးရေးပြီးလည်း ပေါင်းစပေါင်းစပ်သုံးနိုင်လို့ ရေးရတာ မြန်တယ်။ တစ်ခြားလူတွေကလည်း ဒီအချက်က ပို productive ဖြစ်စေတယ်ပြောကြတယ်။

### Python ကို java perl အခြား language များနှင့် ယှဉ်ကြည့်ခြင်း

University of Karlsruhe မှ Prof. Lutz Prechelt က python ကို တစ်ခြား language တွေနဲ့ ယှဉ်ပြပါတယ်။ သူ့ပြောတဲ့ရလဒ်အကျဉ်းချုပ်ကတော့။ " run time ,memory consumption , source text length , comment density , program structure , reliability , လိုအပ်တဲ့ effort အားလုံး အမျိုး အစား ၈၀ လောက်တည်ဆောက်ကြည့်တဲ့အခါ. String တွေdictionary search တွေမှာ scripting language(perl,python,rexx,tcl) တွေက conventional language (c,c++,java)တွေထက်သာတယ်။ runtime နဲ့ memory consumption မှာ java ထက်သာပြီး c,c++ လောက်တော့မဆိုးပါဘူး။ ယေဘူယျဆိုရင် language တွေကြားကွာခြားချက်တော့ သိပ်မရှိပါဘူး ။

-တစ်ခြားကောင်းမွန်တဲ့အချက်များ

Python ကို embedding လုပ်ရတာလွယ်ပါတယ်။ c မှာ အကြာကြီးရေးရမယ့် ဟာကို Python သုံးပြီး ရေးလို့ရတာပေါ့။ အဲ့လိုပဲ Pythonကို c နဲ့ရေးထားတဲ့ Module တွေထည့်လို့ရပါတယ်။ လုပ်ရတဲ့ အကြောင်းကတော့ python နဲ့ရေးမယ့်ဟာ c library ရှိပြီးသားဆိုရင်ပေါ့ ။ နောက်တစ်ချက် ကတော့ ကို ကpythonမှာ manage လုပ်နိုင်တာထက် မြန်မြန် run ချင်ရင်လည်း သုံးပါတယ်။

Python standard library မှာ အသုံးဝင်တဲ့ module တွေရှိပြီးတော့ Install လုပ်တာနဲ့ပါပြီးဖြစ်ပါတယ်...။ Python အတွက် မဖြစ်မနေသိသင့်တာတွေသင်ပြီးရင်တော့ Python standard library တွေနဲ့ အကျွမ်းတဝင်ရှိဖို့လိုပါတယ်။ဒါမှ ဒီ library နဲ့လွယ်လွယ်ဖြေရှင်နိုင်တာ တွေ ပြုလုပ်နိုင်မှာဖြစ်ပါတယ်။

Chapter 3

## Python installation

Python ကို platform အတော်များများအတွက်ရနိုင်ပါတယ် Linux and Mac OS X အပါအဝင်ပေါ့။ Enviromental setup လုပ်ကြည့်ရအောင်။

Verision 2.7 ကိုသုံးပါမယ်။ window ကိုပဲအဓိကထားပြောပါမယ်။(window သုံးနေလို့ပါ)

Local Environment Setup

Python ရှိမရှိသိချင်ရင် ရှိရင်လည်း version သိရအောင် terminal/cmd မှာ python လို့ရိုက်ကြည့်ပါ။

Getting python

နောက်ဆုံးထွက် Python version တွေ document တွေဆိုင်ရာအားလုံးကို https://www.python.org/ မှာ ရနိုင်ပါတယ်။

Installating python

Platform တွေအများကြီးမို့ မိမိနဲ့ဆိုင်ရာ Platform binary code application ကိုရယူပြီး Install ပြုလုပ်ပေးရပါမယ်။

Unix and Linux Installation

1. Web browser ဖွင့်ပြီး https://www.python.org/downloads/ ကိုသွားပါ
2. Unix/Linux နဲ့ဆိုင်ရာ source code downyပါ။
3. Download and extract files.
4. Editing the Modules/Setup file if you want to customize some options.
5. run ./configure script
6. make
7. make install

ဒါဆိုရင် */usr/local/bin* မှာ python ကိုထည့်သွင်းမှာဖြစ်ပြီး */usr/local/lib/pythonXX* မှာ python library တွေထည့်သွင်းမှာဖြစ်ပါတယ်။ XX ကတော့ version no ကိုပြောတာပါ။

### Windows Installation

Window အတွက်ဆိုရင်တော့

1. Web browser ဖွင့်ပြီး https://www.python.org/downloads/ ကိုသွားပါ

2. Windows installer python-XYZ.msi file ကိုရွာပါ XYZ က version ကိုပြောတာပါ
3. python-XYZ.msi ကိုအသုံးပြု နိုင်အောင် Windows system က support Microsoft Installer 2.0 ကို support ပေးရပါမယ်။
4. Downlod လုပ်ထားတဲ့ file ကို run ပါ သူ့ default အတိုင်းပဲ next ok စတာတွေပဲ နှိပ်သွား ပါအဆင်ပြေပါတယ်။

Setting up PATH(optional)

Program တွေ executable file တွေက directory အများအပြားထဲမှာ ရှိတဲ့အတွက် OS က executable file တွေရှာဖို့ search path တွေထဲမှာ executable တွေရှာဖို့ directory list တွေထည့် ထားပါတယ်။

Path ကို Environmental variable ထဲမှာသိမ်းထားပါတယ်။ OS ထဲမှာသိမ်းထားတဲ့ string တစ်ခုပါ။ ဒီ variable ထဲမှာ commeand shell တွေအခြား program တွေကနေ ရယူအသုံးပြုနိုင်တဲ့ information တွေရှိပါတယ်။

Path variable ကို unix မှာ PATH window မှာ Path လို့ခေါ်ပါတယ်.(Unix က case sensitive window ကတော့ case insensitive)

Setting path at Windows(optional)

**At the command prompt** – type path %path%;C:\Python and press Enter.

Python Environment Variables

Python ကနေသိတဲ့ Environemental variable တွေရှပါတယ်။

**PYTHONPATH-** Path နဲ့တူပါတယ်။ python program ထဲကို import လုပ်လိုက်တဲ့ module တွေ ရှာဖို့နေရာပြောပြပေးပါတယ်။ Python source code directory နဲ့ python source code တွေပါတဲ့ directory တွေပါသင့်ပါတယ်။ PYTHONPATH ကို python installer ကနေသတ်မှတ်ပေးထားတာ လည်းဖြစ်နေနိုင်ပါတယ်။

**PYTHONSTARTUP-** Python source နဲ့ပတ်သတ်တဲ့ initialization file တွေရှိတဲ့နေရာပါပါတယ်။ interpreter စတိုင် run ပါတယ်။ .pythonrc.py လို Unix မှာနာမည်ပေးထားပြီး utilities loading လုပ်တဲ့ commandတွေ PYTHONPATH ကိုပြင်တဲ့ command တွေပါပါတယ်။

**PYTHONCASEOK-** window မှာ import statement တွေမှာ case-insensitve match တွေရှာဖို့ python ကိုညွှန်ကြားရင်သုံးပါတယ်။ activate လုပ်ဖို့နှစ်သက်ရာတန်ဖိုးထည့်နိုင်ပါတယ်။

**PYTHONHOME**-module path တွေရှာဖို့နောက်တစ်နေရာဖြစ်ပါတယ်။ ပုံမှန်တော့ PYTHONSTARTUP သို့ PYTHONPATH directoryထဲမှာထည့်ထားပြီး module library တွေပြောင်းရလွယ်အောင်လုပ်ထားပါတယ်။

Running Python

Python ကိုနည်းလမ်းသုံးမျိုးနဲ့စတင်နိုင် runနိုင်ပါတယ်။

Interactive Interpreter

Command line ထဲမှာ python လို့ရိုက်ပြီး enter နှိပ်ပါ။ ဒါဆို စရေးလို့ရပါပြီ။

| S.No. | Option & Description |
|---|---|
| 1 | **-d**<br><br>It provides debug output. |
| 2 | **-O**<br><br>It generates optimized bytecode (resulting in .pyo files). |
| 3 | **-S**<br><br>Do not run import site to look for Python paths on startup. |
| 4 | **-v**<br><br>verbose output (detailed trace on import statements). |
| 5 | **-X**<br><br>disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6. |
| 6 | **-c cmd**<br><br>run Python script sent in as cmd string |
| 7 | **file**<br><br>run Python script from given file |

## Script from the Command-line

Command line ကနေ python file ကိုခေါ်ပြီးလည်း run နိုင်ပါတယ်။

```
C: >python script.py
```

## Integrated Development Environment

Python ကို GUI နဲ့လည်းသုံးနိုင်ပါတယ်။

Windows – Window မှာ window key နှိပ် search မှာ IDLE လို့အလွယ်ရှာနိုင်ပါတယ်။

Unix – IDLE is the very first Unix IDE for Python.

Macintosh – The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files.

# Chapter 4

## Using the Python Interpreter

Python interpreter သုံးဖို့ဆိုရင် command prompt မှာ python လို့ enter နှိပ်ရင်ရပါပြီ။

```
C:\Users\minthargyi>PYTHON
```

ဒါဆိုရင် python ကနေအောက်ပါအတိုင်းပြန်ပြပါလိမ့်မယ်။

```
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Interpreter စပြီးရင် python command တွေကို ">>>" အနောက်မှာ စတင်ရေးလို့ရပါပြီ ပထမဆုံးထုံးစံအတိုင်း "Hello World" လေး print လုပ်ကြည့်ရအောင်

```
>>> print "Hello World"
Hello World
```

လွယ်ပါတယ်နော် ပိုလွယ်ချင်ရင် print မပါပဲရေးလည်းရပါတယ်။ interactive python interpretor မှာ print ရေးစရာမလိုပါဘူး

```
>>> "Hello World"
'Hello World'
>>> 3
3
>>>
```

ပေါင်းနှုတ်မြောက်စား စတာတွေ လုပ်ကြည့်ပါမယ်

```
>>> 12 / 7
1
```

အကြွင်း 1 ပါ

```
>>> 12.0 / 7
1.7142857142857142
```

Float ပါရင် float တန်ဖိုး result ထွက်ပါမယ်။

Casting လုပ်မယ်

```
>>> float(12) / 7
1.7142857142857142
>>>
```

Python မှာလည်း operation order ရှိပါတယ်။

1. exponents and roots
2. multiplication and division
3. addition and subtraction

ဥပမာ "3+(2*4)" မှာဆို () ထည့်စရာတောင်မလိုပါဘူး

```
>>> 3 + 2 * 4
11
>>>
```

နောက်ဆုံးတွက်ချက်ခဲ့တဲ့အဖြေကို "_" ဆိုတဲ့ variable name ထဲမှာ သိမ်းထားပါတယ်။သိချင်ရင် ခေါ်လိုက်ယုံပါပဲ

```
>>> _
11
>>>
```

Underscore ကို တစ်ခြား variable တွေလိုပဲ သုံးနိုင်ပါတယ်။

```
>>> _ * 3
33
>>>
```

Interpreter ကိုပိတ်ချင်ရင် CTRL+Z enter or exit() enter ပါ။

# Chapter 5

## Execute a Python script

Interavtive interpreter က coding သေးသေးလေးတွေ script လေးတွေအတွက် အဆင်ပြေပေမယ့် coding များလာရင် အဆင်မပြေပါဘူး အဲ့ဒါကြောင့် code တွေကို script file ထဲမှာ သိမ်းဖို့လိုပါတယ်။

ဒါဆို text editor တစ်ခုလိုပါမယ်။ အများကြီးရှိတဲ့အထဲမှ syntax highlighting နဲ့ identation support ပေးတာဆိုအဆင်ပြေပါတယ်။ Pycharm community edition ကို recommend ပေးပါတယ်။ https://www.jetbrains.com/pycharm/download/#section=windows

မိမိနှစ်သက်ရာ editor နဲ့ print "hello world" လေးရေး helloworld.py လို့ desktop ပေါ်မှာသိမ်း ပေးပါ အဓိကကတော့ .py extension ဖြစ်ရမှာဖြစ်ပြီး မိမိသိမ်းဆည်းထားရာနေရာကို သိရှိရမှာဖြစ်ပါတယ်။

အောက်မှာ cmd နဲ့ run ပြထားပါတယ်။

```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\minthargyi>cd Desktop

C:\Users\minthargyi\Desktop>python helloworld.py
hello world
C:\Users\minthargyi\Desktop>
```

## Python Internals

Python ကို interpreted programming သို့ scripting language လို့ပြောကြပါလိမ့်မယ်။ အမှန်တော့ Python က interpreted ဖြစ်သလို compiled language လည်းဖြစ်ပါတယ်။ ဒါပေမယ့် compiled language လိုပြောရင်နားလည်မှုလွဲနိုင်ပါသေးတယ်။ compiler က python code ကို machine language ပြောင်းလိုက်တယ်လို့မြင်ကြပါလိမ့်မယ်။ python code ကို intermediate code အရင်ပြောင်းပါတယ်။ အဲ့ဒါကိုမှ PVM လို့ခေါ်တဲ့ python virtual machine မှာ execute လုပ်ပါတယ်။ java နဲ့တူပါတယ်။ python ကို Jpython သုံးပြီး java byte code အဖြစ်ပြောင်းလို့ရတဲ့နည်းတွေတောင် ရှိပါတယ်။

မေးခွန်းက python code တွေကို မြန်ဆန်အောင် compile လုပ်ရမှာလား or ဘယ်လို compile လုပ်မလဲ စတာတွေပါ။ အဖြေကတော့ ဘာမှလုပ်စရာမလိုပါဘူး python ကလုပ်ပေးနေတာကြောင့်ပါ။

ကိုက manual လုပ်ချင်ပါတယ်ဆိုလည်းရပါတယ်။ interpreter ထဲမှာ py_compile module သုံးပြီးလုပ်နိုင်ပါတယ်။

```
>>> import py_compile
>>> py_compile.compile('easy_to_write.py')
>>>
```

Cmd မှာလုပ်ချင်ရင်

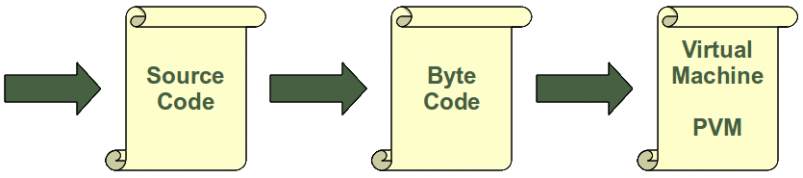
```
python -m py_compile helloworld.py
```

ဘယ်လိုပဲလုပ်လုပ် "helloworld.pyc" ဆိုတဲ့ file လေးလုပ်ပေးပါလိမ့်မယ်။

Directory တစ်ခုထဲမှာ ရှိနေတဲ့ python file အားလုံးကို compile လုပ်ချင်လည်းရပါတယ်။

```
python -m compileall .
```

ဒါပေမယ့်ပြောခဲ့သလိုပဲ user ကလုပ်စရာမလိုပါဘူး။ python program file ရှိတဲ့ directory မှာ write access ရှိရင် .pyc file တွေသိမ်းတတ်ပါတယ်။ write access မရှိလည်း အလုပ်လုပ်နေမှာပါပဲ ။ byte code ထုပ်ပြီး program ရပ်သွားရင် ဖျက်လိုက်မှာပါ။ python program ကို execute လုပ်တိုင်း .pyc file ရှိလားအရင်ကြည့်ပါတယ်။ ပိုမြန်တဲ့အတွက် အဲ့ဒီ file ကို load လုပ်ပေးမှာပါ။ မရှိဘူးဆိုရင်တော့ byte code ထုပ်ပြီး execute လုပ်ပါတယ်။ Execute လုပ်တယ်ဆိုတာ byte code တွေကို Python Virtual Machine (PVM) ပေါ်မှာ run တာကိုဆိုလိုတာဖြစ်ပါတယ်။




Python script execution လုပ်တိုင်း byte code ထုတ်တယ်။ module အနေနဲ့သုံးရင်လည်း သက်ဆိုင်ရာ .pyc file ထဲမှာသိမ်းထားတယ်။

အောက်မှာ desktop ထဲက file list ပါ။

```
C:\Users\minthargyi>cd Desktop

C:\Users\minthargyi\Desktop>dir

 Volume in drive C has no label.

 Volume Serial Number is 1E04-AD6F

 Directory of C:\Users\minthargyi\Desktop

09/08/2017  03:12 PM    <DIR>          .
```

```
09/08/2017  03:12 PM    <DIR>          ..
09/08/2017  01:58 PM            28,184 A Note On Pyhton 2.docx
09/07/2017  07:17 PM            37,093 A Note On Pyhton 2.pdf
09/03/2017  11:05 AM                11 hello.py
09/08/2017  02:41 PM                19 helloworld.py
07/23/2017  05:54 PM    <DIR>          ngtest
               4 File(s)         65,307 bytes
               3 Dir(s)  64,989,577,216 bytes free
C:\Users\minthargyi\Desktop>
```

Helloworld ကို run ကြည့်ပါမယ်။

```
C:\Users\minthargyi\Desktop>python helloworld.py
hello world
C:\Users\minthargyi\Desktop>dir
 Volume in drive C has no label.
 Volume Serial Number is 1E04-AD6F
 Directory of C:\Users\minthargyi\Desktop
09/08/2017  03:12 PM    <DIR>          .
09/08/2017  03:12 PM    <DIR>          ..
09/08/2017  01:58 PM            28,184 A Note On Pyhton 2.docx
09/07/2017  07:17 PM            37,093 A Note On Pyhton 2.pdf
09/03/2017  11:05 AM                11 hello.py
09/08/2017  02:41 PM                19 helloworld.py
07/23/2017  05:54 PM    <DIR>          ngtest
               4 File(s)         65,307 bytes
               3 Dir(s)  64,990,224,384 bytes free
```

```
C:\Users\minthargyi\Desktop>
```

.pyc file မဆောက်ပေးပါဘူး

နောက်တစ်နည်းသွားကြည့်မယ်။

```
C:\Users\minthargyi\Desktop>python
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import helloworld
hello world
>>> exit()
C:\Users\minthargyi\Desktop>dir
 Volume in drive C has no label.
 Volume Serial Number is 1E04-AD6F
 Directory of C:\Users\minthargyi\Desktop
09/08/2017  03:16 PM    <DIR>          .
09/08/2017  03:16 PM    <DIR>          ..
09/08/2017  01:58 PM            28,184 A Note On Pyhton 2.docx
09/07/2017  07:17 PM            37,093 A Note On Pyhton 2.pdf
09/03/2017  11:05 AM                11 hello.py
09/08/2017  02:41 PM                19 helloworld.py
09/08/2017  03:16 PM               121 helloworld.pyc
07/23/2017  05:54 PM    <DIR>          ngtest
               5 File(s)         65,428 bytes
               3 Dir(s)  64,998,924,288 bytes free

C:\Users\minthargyi\Desktop>
```

.pyc file ပါလာတာသတိပြုမိပါလိမ့်မယ့်

## Compiler

Compiler ကတော့ programmng language တစ်ခုကနေ အခြားတစ်ခုကိုပြောင်းလဲပေးပါတယ်။ source code ကနေ executable program အဖြစ်ပြောင်းတာ အဖြစ်သုံးကြပါတယ်။ hight level language(human readable ) to low level language(machine language)

### Interpreter

Programming language တစ်ခုခုနဲ့ရေးထားတဲ့ instruction တွေကို execute လုပ်ပါတယ်။

Source code ကို direct run တာဖြစ်နိုင်သလို]

ပိုကောင်းတဲ့ language တစ်ခုအဖြစ်ပြောင်းလဲပြီးမှ execute လုပ်တာလည်းဖြစ်နိုင်ပါတယ်။

Help

```
help("execfile")
```

# Chapter 6

## Structuring with Indentation

Programmng language တော်တော်များများမှာ statement တွေ group ဖွဲ့ဖို့ character တွေ keyword တွေသုံးကြပါတယ်။

- begin ... end
- do ... done
- { ... }
- if ... fi

python မှာ တစ်ခြားနည်းသုံးပါတယ်။ structre ကို identation (စကားလုံးအကွာအဝေး) သံသုံးပြီးဆောက်ပါတယ်။ identation တူရင် block တူတဲ့သဘောပါပဲ တစ်ခြား language တွေမှာ ဆိုရင် လှအောင် ဖ တ်ရလွယ်အောင် identation သုံးကြပေမယ့် python မှကတော့ language
လိုအပ်ချက်အနေနဲ့မဖြစ်မနေသုံးပေးရပါမယ်။ ဖတ်ရနားလည်ရလည်းပိုလွယ်စေပါတယ်။

ညာဘက်ကနေ အကွာအဝေးတူ တဲ့ statement တွေအားလုံး block တစ်ခုထဲမှာရှိပါတယ်။ ဆိုလိုတာကတော့ block တစ်ခုထဲမှာဆိုရင် အထက်အောက်စာလုံးအစတွေညီနေပါလိမ့်မယ်။ indent လုပ်တဲ့နောက်ဆုံးလိုင်း သို့ file အဆုံးမှာ block ဆုံးပါတယ်။ block ထဲမှာ နောက်ထပ် block တွေထပ်ထည့်မယ်ဆိုလည်း ညာဘက်ကို indent ထပ်လုပ်ပေးယုံပါပဲ။

တစ်ကြောင်းတည်းမလောက်လို့ ဆက်ရေးမယ်ဆိုအဆုံးမှာ "\" ထည့်ပေးရပါမယ်။

Eg

First block

    Inerblock statement 1

    Inerblock statement 2

    Inerblock statement 3

Second block

Third block

# Chapter 7

## Data Types and Variables

Python မှာ powerfull ဖြစ်တဲ့ data type တွေပါဝင်တဲ့အတွက်ကြောင့် ၎င်းတို့ကို သိရှိဖို့လိုပါတယ်။

## Variables

Variable ကတော့ နာမည်မှာပါသလိုပဲ ပြောင်းလဲလို့ရဲ့အရာတစ်ခုပါ။ computer memory ထဲမှာရှိတဲ့ memory location တစ်ခုကိုညွှန်းချင်ရင်သုံးတဲ့နည်းလမ်းတစ်ခုပါပဲ။ တကယ်ရှိတဲ့နေရာအတွက် အမှတ်အသားလေးတစ်ခုဖြစ်ပါတယ်။ memory location မှာတော့ တန်ဖိုးတွေဖြစ်တဲ့ number,text, နဲ့အခြားရှုပ်ထွေးတဲ့ datatype တွေပါရှိပါတယ်။

Variable ကို တန်ဖိုးတွေ သိမ်းထားဖို့ နေရာလွတ်လေးတွေ container လေးတွေလို့ မှတ်သားနိုင်ပါတယ်။ program run နေချိန်မှာ တန်ဖိုးတွေရယူသုံးစွဲနိုင်သလို တန်ဖိုးအသစ်တွေ ပြောင်းလဲထည့်သွင်းနိုင်ပါတယ်။

Python နဲ့အခြား strong type language တွေဖြစ်တဲ့ c ,c++, java တွေနဲ့မတူတဲ့အချက်ကတော့ types တွေနဲ့အလုပ်လုပ်ပုံကွာတာဖြစ်ပါတယ်။ strong typed language တွေမှာ variable တွေက တိကျတဲ့ data type တစ်ခုရှိရပါတယ်။ Eg. Integer type ဆိုရင် integer value တွေပဲသိမ်းနိုင် ပါတယ်။ java ရော c ရော variable တွေမသုံးခင် အရင်ကြေငြာ(declare)ရပါတယ်။ declare လုပ်တယ်ဆိုတာက variable နဲ့ datatype binding လုပ်တာဖြစ်ပါတယ်။

Python မှာ တော့ variable declaration လုပ်ဖို့မလိုပါဘူး ။ မိမိလိုတဲ့အချိန် သုံးရုံပါပဲ။

နောက်တစ်ချက်က variable value ပြောင်းနိုင်သလို variable data type ကိုလည်း program run နေချိန်မှာ ပြောင်းနိုင် ပါတယ်။ integer variable တစ်ခုလုပ်သုံးနောက်လိုလာရင် ၎င်းကို variable ကို string type အဖြစ်ပြောင်းလဲသုံးနိုင်ပါတယ်။ Eg.

```
>>> i=42
```

ဒီမှာဆိုရင် variable i ကို 42 လို့သတ်မှတ်လိုက်ပါတယ်။ 1 ပေါင်းကြည့်မယ်။

```
>>> i=i+1
>>> print i
43
>>>
```

## Variables vs. Identifiers

Variable နဲ့ identifier တူတယ်လို့မှားပြောကြပါတယ်။ ရှင်းရှင်းပြောရရင် variable ရဲ့နာမည် ကို identifier လို့ခေါ်ပါတယ်။ ဒါပေမယ့် variable ကအဲ့ဒီထက်ပိုပါတယ်။ variable မှာ နာမည်ရှိမယ်။ type ရှိမယ်။ scope ရှိမယ် value တွေရှိမယ်။ ပြီးတော့ identifier ဆိုတာ variable name အတွက် မှမဟုတ်ပါဘူး။ အခြား variables, types, labels, subroutines or functions, packages စတာ တွေအတွက်လည်းသုံးပါသေးတယ်။

Naming Identifiers of Variables

အခြားprogramming language တွေလိုပဲ python မှာလည်း rule တွေရှိပါတယ်။

Empty မဟုတ်တဲ့ character sequence တွေပြီးတော့

- "_" or a capital or lower case letter နဲ့စနိုင်တယ်။
- အစစကားလုံးပြီးရင် နောက်မှာ အစစကားလုံးမှာခွင့်ပြုထားတာတွေအပြင် digit တွေပါပါနိုင်ပါတယ်။ .
- Window မှာ တော့ case-sensitive ဖြစ်ပါတယ်။
- Python keywords တွေကို သုံးလို့မရပါဘူး

## Python Keywords

Identifier တွေမှာ Python keyword တွေပေးလို့မရပါဘူး။

and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield

Changing Data Types and Storage Locations

အထက်မှာပြောခဲ့သလိုပဲ data type တွေကို program execute လုပ်နေချိန်မှာ ပြောင်းလဲနိုင်ပါတယ်။

```
>>> i=42
>>> i=42+0.11
>>> i="forty"
>>>
```

Physically ဘယ်လို ထားမယ်ဆိုတာတော့ python ကလုပ်ပေးသွားပါတယ်။ interger float string စတာတွေကို နေရာသပ်သပ်ဆီသိမ်းပေးပါတယ်။

Assignment လုပ်တဲ့အခါဘာတွေဖြစ်လဲ? ကြည့်ရအောင်

```
>>> x = 3
>>> y = x
>>> y = 2
```

ပထမ assignemet မှာ ပြဿနာမရှိဘူး။ x အတွက် memory နေရာတစ်ခု python ကလုပ်ပေးမယ် ပြီးတော့ integer value 3 ကိုသိမ်းလိုက်မယ်။ ဒုတိယ မှာကျတော့ ထင်ရတာကို python က y အတွက် memory location လုပ်ပြီး 3 ကို ကူးပြီးထည့်လိုက်မယ် လို့ ထင်စရာရှိပါတယ်။ python မှာ အဲ့လိုမလုပ်ပါဘူး။ value တွေတူတဲ့အတွက်ကြောင့် Python က y ကို x ရဲ့ memory location ကိုပဲညွှန်းပေးပါတယ်။

တတိယ statement မှာ မေးစရာမေးခွန်းရှိလာပါပြီ y ကို 2 လို့ထားလိုက်ပါတယ်။ ဒါဆို x ကဘာဖြစ်သွားမလဲ C သမားတွေကတော့ x လည်းပြောင်းသွားမယ်ပြောပါလိမ့်မယ်။ ဘာလို့လဲ ဆိုတော့ y က x ရဲ့ memory location ကို ညွှန်းထားတယ်ပြောခဲ့တာကိုး။ ဒါပေမယ့် ဒါက c pointer မဟုတ်ပါဘူး ။ x နဲ့ y က value မတူတော့တဲ့အတွက် y လည်းသူ့ကိုယ်ပိုင် memory location ရသွားပါတယ်။ ဆိုတော့ y က 2 x ကလည်း သူ့တန်ဖိုး 3 ပါပဲ။

Identify function id() နဲ့သေချာအောင်လုပ်ကြည့်မယ်။ instance (Object /variable )အားလုံးမှာ identity ရှိပါတယ်။

```
>>> x = 3
>>> print id(x)
157379912
>>> y = x
>>> print id(y)
157379912
>>> y = 2
>>> print id(y)
157379924
>>> print id(x)
157379912
>>>
```

Numbers

Python ရဲ့ built in data type တွေကို object type လို့လည်းခေါ်ပါသေးတယ်။ number အတွက် 4 မျိုးရှိပါတယ်။

- Integer
  - Normal integers
    e.g. 4321
  - Octal literals (base 8)
    A number prefixed by a 0 (zero) will be interpreted as an octal number
    example:

```
>>> a = 010
>>> print a
```

8 Alternatively, an octal number can be defined with "0o" as a prefix:

```
>>> a = 0o10
>>> print a
8
```

- Hexadecimal literals (base 16)

  Hexadecimal literals have to be prefixed either by "0x" or "0X".

  example:

```
>>> hex_number = 0xA0F
>>> print hex_number
2575
```

- Long integers

  these numbers are of unlimited size

  e.g.42000000000000000000L

- Floating-point numbers

  for example: 42.11, 3.1415e-10

- Complex numbers

  Complex numbers are written as <real part> + <imaginary part>j

  examples:

```
>>> x = 3 + 4j
>>> y = 2 - 3j
>>> z = x + y
>>> print z
(5+1j)
```

Strings

String တွေကို quote "' လေးတွေနဲ့မှတ်ပါတယ်။

- Wrapped with the single-quote ( ' ) character:
  'This is a string with single quotes'
- Wrapped with the double-quote ( " ) character:
  "Obama's dog is called Bo"
- Wrapped with three characters, using either single-quote or double-quote:
  '''A String in triple quotes can extend
  over multiple lines like this one, and can contain
  'single' and "double" quotes.'''

Python string တွေမှာ characters , letters ,numbers, special characters တွေပါဝင်ပါတယ်။ string တွေကို index နဲ့သုံးလို့လည်းရပါတယ်။ c မှာ လိုပဲ ပထမစလုံးက index 0ပါ

```
>>> s = "A string consists of characters"
```

```
>>> s[0]
'A'
>>> s[3]
't'
```

နောက်ဆုံးစာလုံးကတော့

```
>>> s[len(s)-1]
's'
```

ပိုလွယ်တဲ့နည်းလမ်းက နောက်ဆုံး character ကို index -1 ဒုတိယကို -2 သုံးနိုင်ပါတယ်။

```
>>> s[-1]
's'
>>> s[-2]
'r'
```

Python မှာ character type မရှိပါဘူး ၊ string size one ဆို character ပါပဲ

Index တွေကို ညာဖက်ကရေတွက်ခြင်ရင် -1 ကစပါတယ်။

String operators တွေ function တွေ

- Concatenation
  Strings can be glued together (concatenated) with the + operator:
  "Hello" + "World" will result in "HelloWorld"
- Repetition
  String can be repeated or repeatedly concatenated with the asterisk operator "*":
  "*-*" * 3 -> "*-**-**-*"
- Indexing
  "Python"[0] will result in "P"
- Slicing
  Substrings can be created with the slice or slicing notation, i.e. two indices in square brackets separated by a colon:
  "Python"[2:4] will result in "th"
- Size
  len("Python") will result in 6

## Immutable Strings

Java နဲ့တူပြီး C C++ နဲ့မတူတာက python string တွေကို ပြောင်းလို့မရပါဘူး။ index postion တွေကို value ပြောင်းကြည့်ရင် error ပေးပါလိမ့်မယ်။

```
>>> s = "Some things are immutable!"
>>> s[-1] = "."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

## A String Peculiarity

String တွေမှာ ထူးခြားချက်တစ်ခုရှိတယ်။ eg/ "is" operator လိုပါမယ်။ a နဲ့ b က string တွေဖြစ်တယ် "a is b " ဆိုရင် identity တူမတူစစ်တာဖြစ်ပါတယ်(memory share သုံးကြတယ်ပေါ့ )။ "a is b" က True ဆိုရင် "a==b" လည်းတူရမှာဖြစ်ပါတယ်။ ဒါပေမယ့် "a==b" ဖြစ်ယုံနဲ့တော့ "a is b" က True မဖြစ်ပါဘူး။ eg

```
>>> a = "Linux"
>>> b = "Linux"
>>> a is b
True
```

String အရှည်ဆိုရင်ရော ?

```
>>> a = "Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch"
>>> b = "Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch"
>>> a is b
True
```

အတူတူပါပဲ ခုပြမယ့် eg မှာတော့ မတူတော့ပါဘူး

```
>>> a = "Baden-Württemberg"
>>> b = "Baden-Württemberg"
>>> a is b
False
>>> a == b
True
```

Special character hypen ပါနေလို့ဖြစ်ပါတယ်။

```
>>> a = "Baden!"
>>> b = "Baden!"
>>> a is b
False
>>> a = "Baden1"
>>> b = "Baden1"
>>> a is b
True
```

Escape Sequences

| Escape Sequence | Meaning Notes |
|---|---|
| \newline | Ignored |
| \\ | Backslash (\) |
| \' | Single quote (') |
| \" | Double quote (") |
| \a | ASCII Bell (BEL) |
| \b | ASCII Backspace (BS) |
| \f | ASCII Formfeed (FF) |
| \n | ASCII Linefeed (LF) |
| \N{name} | Character named name in the Unicode database (Unicode only) |
| \r | ASCII Carriage Return (CR) |
| \t | ASCII Horizontal Tab (TAB) |
| \uxxxx | Character with 16-bit hex value xxxx (Unicode only) |
| \Uxxxxxxxx | Character with 32-bit hex value xxxxxxxx (Unicode only) |
| \v | ASCII Vertical Tab (VT) |
| \ooo | Character with octal value ooo |
| \xhh | Character with hex value hh |

# Chapter 8

## Arithmetic and Comparison Operators

### Operators

| Operator | Description | Example |
|---|---|---|
| +, - | Addition, Subtraction | 10 -3 |
| *, /, % | Multiplication, Division, Modulo | 27 % 7<br>Result: 6 |
| // | Truncation Division (also known as floordivision or floor division)<br>The result of this division is the integral part of the result, i.e. the fractional part is truncated, if there is any. If both the divident and the divisor are integers, the result will be also an integer. If either the divident or the divisor is a float, the result will be the truncated result as a float. | >>> 10 // 3<br>3 >>> 10.0 // 3<br>3.0 >>> |
| +x, -x | Unary minus and Unary plus (Algebraic signs) | -3 |
| ~x | Bitwise negation | ~3 - 4<br>Result: -8 |
| ** | Exponentiation | 10 ** 3<br>Result: 1000 |
| or, and, not | Boolean Or, Boolean And, Boolean Not | (a or b) and c |
| in | "Element of" | 1 in [3, 2, 1] |
| <, <=, >, >=, !=, == | The usual comparison operators | 2 <= 3 |
| \|, &, ^ | Bitwise Or, Bitwise And, Bitwise XOR | 6 ^ 3 |
| <<, >> | Shift Operators | 6 << 3 |

# Chapter 9

Input from Keyboard

Input Function

Python မှာ keyboard ကနေ input ယူဖို့ input() function ပါပါတယ်။ optional parameter တစ်ခုပါပြီးတော့ prompt string လေးတစ်ခုပြဖို့သုံးနိုင်ပါတယ်။

Input function ကို ခေါ်တဲ့အခါ program flow က user input မပေးမချင်းရပ်နေမှာပါ။ optional parameter ထဲက string ကတော့ screen မှာပေါ်နေမှာပါ။

ပြီးတော့ user input လုပ်သမျှကို interpret လည်းလုပ်မှာပါ။ integer ဆိုရင် Integer ၊ list ထည့် ရင် list return ပြန်ပေးပါတယ်။

```
name = input("What's your name? ")
print("Nice to meet you " + name + "!")
age = input("Your age? ")
print("So, you are already " + str(age) + " years old, " + name + "!")
```

"input_test.py" လို့သိမ်းပြီး runကြည့်ပါ။

```
$ python input_test.py
What's your name? "Frank"
Nice to meet you Frank!
Your age? 30
So, you are already 30 years old, Frank!
```

တကယ်လို့ ကိုက name ထည့်ရာမှာ quotes တွေကျန်သွားရင်တော့ error ပြပါလိမ့်မယ်။

```
Traceback (most recent call last):
  File "input_test.py", line 1, in <module>
    name = input("What's your name? ")
  File "<string>", line 1, in <module>
NameError: name 'Frank' is not defined
```

အစမှာပြောခဲ့သလိုပဲ input ကို interpret လုပ်တဲ့အတွက် ကြောင့် quoutes မပါရင် variable အနေနဲ့သတ်မှတ်တဲ့အတွက်ကြောင့် error ပြတာဖြစ်ပါတယ်။

```
>>> name = input("What's your name? ")
What's your name? "John"
>>>
>>> age = input("Your age? ")
Your age? 38
>>> print(age, type(age))
(38, <type 'int'>)
>>> colours = input("Your favourite colours? ")
Your favourite colours? ["red","green","blue"]
>>> print(colours)
['red', 'green', 'blue']
>>> print(colours, type(colours))
(['red', 'green', 'blue'], <type 'list'>)
>>>
```

Input with raw_input()

Raw_input ကတော့ interpret လုပ်မပေးပါဘူး။ user ထည့်သမျှ အပြောင်းအလဲမရှိ return ပြန်ပေးပါတယ်။ raw inpyt ကိုလည်း လိုအပ်သလို casting တွေ eval function တွေသုံးပြီး data type ပြောင်းနိုင်ပါတယ်။

```
>>> age = raw_input("Your age? ")
Your age? 38
>>> print(age, type(age))
('38', <type 'str'>)
>>>
>>> age = int(raw_input("Your age? "))
Your age? 42
>>> print(age, type(age))
(42, <type 'int'>)
>>>
>>> programming_language = raw_input("Your favourite programming languages?
")
Your favourite programming languages? ["Python", "Lisp","C++"]
>>> print(programming_language, type(programming_language))
('["Python", "Lisp","C++"]', <type 'str'>)
>>>
>>> programming_language = eval(raw_input("Your favourite programming
languages? "))
Your favourite programming languages?  ["Python", "Lisp","C++"]
>>> print(programming_language, type(programming_language))
(['Python', 'Lisp', 'C++'], <type 'list'>)
>>>
```

ဒါပေမယ့် list ကို casting လုပ်ရင်တော့ ကျွန်တော်တို့ထင်သလိုလုပ်မှာမဟုတ်ပါဘူး

```
>>> programming_language = list(raw_input("Your favourite programming
languages? "))
Your favourite programming languages?  ["Python", "Lisp","C++"]
>>> print(programming_language, type(programming_language))
([' ', '[', '"', 'P', 'y', 't', 'h', 'o', 'n', '"', ',', ' ', '"', 'L',
'i', 's', 'p', '"', ',', '"', 'C', '+', '+', '"', ']'], <type 'list'>)
>>>
```

တစ်လုံးခြင်းစီဖြတ်ပြစ်တာတွေ့ရပါလိမ့်မယ်။

# Chapter 10

Conditional Statements

Programming language တွေမှာ conditional statement or conditional construct တွေကို အခြေအနေတစ်ခုပေါ်မူတည်ပြီး တွက်ချက်တာမျိုး တွေလုပ်ရင်သုံးပါတယ်။ python မှာ True နဲ့ False လို့ပဲသုံးပါတယ်။

Condition တွေစစ်တဲ့အခါ variable တွေ ကို comparison and arithmetic expression တွေနဲ့ တွဲသုံးကြပါတယ်။ ဒီ expression တွေကို Boolean value တွေဖြစ်တဲ့ True or False လားတွက်ချက် ပါတယ်။ decision လုပ်ဖို့ရေးသားတဲ့ statement တွေကို condition statement or conditional expressions or conditional constructs လို့ခေါ်ပါတယ်။

If-then ကို language တွေမှာသုံးကြပါတယ်။ syntax ရေးတဲ့ပုံစံတော့ကွာကြပါတယ်။

The if Statement

ယခုလိုရေးပါတယ်

```
if condition_1:
    statement_block_1
elif condition_2:
    statement_block_2
else:
    statement_block_3
```

condtion 1 မှန် ရင် statement_block_1 မမှန် ရင် condition 2 ထပ်စစ် မှန်ရင် statement_block_2 မှားရင် statement_block_3 အလုပ်လုပ်ပါမယ်။

Example Dog Years

ခွေးအသက် 1 နှစ်ဆို လူအသက် 14 နှစ်

ခွေးအသက် ၂ နှစ်ဆို လူအသက် 22 နှစ်

ကျန်တာက လူအသက် ၅ နှစ်

```
age = input("Age of the dog: ")
print
if age < 0:
        print "This can hardly be true!"
elif age == 1:
        print "about 14 human years"
elif age == 2:
        print "about 22 human years"
elif age > 2:
        human = 22 + (age -2)*5
        print "Human years: ", human
```

True or False

True or False ခွဲခြားရတာကို python မှာ ဒီလိုသတ်မှတ်ထားပါတယ်။

အောက်ပါ object တွေကို False လို့သတ်မှတ်ပါတယ်။

- numerical zero values (0, 0L, 0.0, 0.0+0.0j),
- the Boolean value False,
- empty strings,
- empty lists and empty tuples,
- empty dictionaries.
- plus the special value None.

ကျန်တာက True ပါ။

Abbreviated IF statement

If ကို အတိုရေးချင်လည်းရပါတယ်။

```
max = a if (a > b) else b;
```

# Chapter 11

## Loops

### General Structure of a Loop

ထပ်ခါထပ်ခါလုပ်ဆောင်ရမယ့် အလုပ်တွေအတွက် loop construcet တွေသုံးပါတယ်။ python မှာ loop နှစ်မျိုးပါပါတယ်။ while loop နဲ့ for loop တို့ဖြစ်ပါတယ်။

### A Simple Example with a While Loop

1 က နေ 100 ထိပေါင်းကြည့်ရအောင်

```
n = 100

sum = 0
i = 1
while i <= n:
    sum = sum + i
    i = i + 1

print "Sum of 1 until %d: %d" % (n,sum)
```

### Reading Standard Input

While loop ကိုမပြောခင် standard input/output တွေပြောကြည့်ရအောင်။ standard input ဆိုရင် keyboard ကိုဆိုလိုတာဖြစ်ပြီးတော့ standard output ကတော့ terminal သို့ console output တွေကိုပြောတာပါ။ error တွေကိုလည်း ဒီမှာပဲပြရမှာပါ။

Python မှာ အောက်ပါ channel ၃ ခုရှိပါတယ်။

- standard input
- standard output
- standard error

အားလုံးက sys ဆိုတဲ့ Module မှာပါပါတယ်။နာမည်တွေကတော့

- sys.stdin
- sys.stdout
- sys.stderror

အောက်ပါ example မှာ keyboard input ကို while loop သုံးဖတ်ပြထားပါတယ်။၏

```
import sys

text = ""
while 1:
   c = sys.stdin.read(1)
   text = text + c
   if c == '\n':
```

```
        break

print "Input: %s" % text
```

ပုံမှန်တော့ raw_input() ကိုပဲသုံးကြပါတယ်။

```
>>> name = raw_input("What's your name?\n")
What's your name?
Tux
>>> print name
Tux
>>>
```

The else part

If statement လိုပဲ while loop မှာ optional else တစ်ခုပါပါတယ်။ အခြား language တွေနဲ့ တော့ ကွဲနေပါလိမ့်မယ်။ else အပိုင်းကို while loop condition မမှန်တော့ရင် execution လုပ်ပါတယ်။ else ထပ်ပါတော့ ဘာပိုကောင်းသွားလဲသိဖို့တော့ break statement လည်းသိဖို့ လိုပါတယ်။

While loop ပုံစံကတော့

```
while                                                    condition:
statement_1
...
statement_n
else:
statement_1
...
statement_n
```

Premature Termination of a while Loop

While loop တစ်ခုဟာ သူသတ်မှတ်ထားတဲ့ အခြေအနေတစ်ခုရောက်မှ အဆုံးသတ်ပါတယ်။ break statement နဲ့ဆိုလျှင် သတ်မှတ်ချက်မပြည့်မီလည်း loop ကနေ ထွက်သွားအောင် လုပ်လို့ ရပါတယ်။ break နဲ့ continue ကလည်းမတူပါဘူး။ continue ကလက်ရှိ loop ကိုရပ်ပြီး loop နောက်တစ်ဆင့်ကိုကူးပါတယ်။

တကယ်လို့ loop က break သုံးပြီးထွက်သွားရင် else အပိုင်းအလုပ်မလုပ်တော့ပါဘူး။

နံပါတ်ခန့်မှန်းတဲ့ game လေးနဲ့ရှင်းပြထားပါတယ်။

လူတစ်ယောက်က 1 to n ကြား နံပါတ်တစ်ခုခန့်မှန်းရမယ်။ keyboard ကနေ input ပေးရမယ်။ ခန့်မှန်းတာက တကယ့် နံပါတ်ထက် ကြီးလား ငယ်လား တူလားဆိုတာကို program က အသိပေးရမယ်။ တကယ်လို့အရုံးပေးချင်ရင် ဂ or အနှုတ်ကိန်းတစ်ခုထည့်ပေးရမယ်။

Random no လုပ်ဖို့ "random" modul လိုပါတယ်။

```
import random
```

```
n = 20
to_be_guessed = int(n * random.random()) + 1
guess = 0
while guess != to_be_guessed:
    guess = input("New number: ")
    if guess > 0:
        if guess > to_be_guessed:
            print "Number too large"
        else:
            print "Number too small"
    else:
        print "Sorry that you're giving up!"
        break
else:
    print "Congratulation. You made it!"
```

# Chapter 12

For Loops

Introduction

For loop statement ကလည်း C C++ နဲ့ကွဲပါတယ်။ batsh shell loop နဲ့ဆင်ပါတယ်။

တစ်ခါတစ်ရံ list တွေ number အများကြီးတွေနဲ့ operation တွေလုပ်ရတာရှိပါတယ်။ python for statement က ဒီလို list တွေ range type တွေအများကြီးအတွက် အဆင်ပြေပါတယ်။

Syntax of the For Loop

```
for variable in sequence:
        Statement1
        Statement2
        ...
        Statementn
else:
        Else-Statement1
        Else-Statement2
        ...
        Else-Statementm
```

Example of a for loop in Python:

```
>>> languages = ["C", "C++", "Perl", "Python"]
>>> for x in languages:
...     print x
...
C
C++
Perl
Python
>>>
```

The range() Function

Number sequence တွေနဲ့ iteration တွေလုပ်မယ်ဆိုရင် range() function ကအဆင်ပြေပါတယ်။ သူက arithmetic progession list တွေထုပ်ပေးပါတယ်။

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

range(n) ဆိုရင် 0 ကနေ စပြီး (n-1) အထိထုပ်ပေးပါတယ်။

range() ကို argument နှစ်ခုနဲ့လည်းအသုံးပြုနိုင်ပါတယ်။

```
range(begin,end)
```

အစကနေ အဆုံးထက် 1 ခုလျော့ထိ ထုတ်ပေးမှာပါ။ default increment က ၁ ပါ။ အဲ့တာကိုပြင်ချင် ရင် တတိယ argument ထည့်ပေးရပါမယ်။ step လို့ခေါ်ပါတယ် non-zero ဖြစ်ရပါမယ်။

```
range(begin,end, step)
```
Example with step:
```
>>> range(4,10)
[4, 5, 6, 7, 8, 9]
>>> range(4,50,5)
[4, 9, 14, 19, 24, 29, 34, 39, 44, 49]
```
range() function က loop နဲ့သုံးရင် အလွန်အသုံးဝင်ပါတယ်။ အောက်ပါဥပမာမှာ range() function က number 1 to 100 ထိ for loop ကိုထုတ်ပေးပါတယ်။

```
n = 101
sum = 0
for i in range(1,n):
    sum = sum + i
print sum
```

Calculation of the Pythagorean Numbers

Pythagorean definition က $a^2+b^2=c^2$ ဖြစ်ပါတယ်။

```
from math import sqrt
n = raw_input("Maximal Number? ")
n = int(n)+1
for a in range(1,n):
    for b in range(a,n):
        c_square = a**2 + b**2
        c = int(sqrt(c_square))
        if ((c_square - c**2) == 0):
            print a, b, c
```

Iterating over Lists with range()

List index တွေကိုသိချင်ရင် for loop သုံးရတာအဆင်မပြေပါဘူး။ element တွေကိုသိနိုင်ပေမယ်။ ဆိုင် ရာ index တွေကိုမသိနိုင်ပါဘူး။ index ရော value ရော သိချင်ရင် range() နဲ့ Len() တွဲသုံးပေးရပါမယ်။

```
fibonacci = [0,1,1,2,3,5,8,13,21]
for i in range(len(fibonacci)):
    print i,fibonacci[i]
print
```
len() က list တွေ tuple တွေမှာ ပါတဲ့ element အရေအတွက်ကိုလိုချင်ရင်သုံးပါတယ်။

List iteration with Side Effects

List တွေကို interation လုပ်ရင် list ကို loop body မှာ ပြောင်းလဲတာတွေမလုပ်တာကောင်းပါတယ်။ eg.

```
colours = ["red"]
for i in colours:
    if i == "red":
        colours += ["black"]
    if i == "black":
        colours += ["white"]
print colours
```

ဘာ print ထုတ်မလဲဆိုတော့

```
['red', 'black', 'white']
```

အခဲ့လိုမဖြစ်အောင် slicing operator သုံးပြီး list ကို copy ပွားပြီးသုံးနိုင်ပါတယ်။

```
colours = ["red"]
for i in colours[:]:
    if i == "red":
        colours += ["black"]
    if i == "black":
        colours += ["white"]
print colours
```

ဒီတစ်ခါ print ထုတ်မှာက

```
['red', 'black']
```

"colours" list ကိုပြောင်းပေမယ့်လည်း loop က element ကမပြောင်းတဲ့အတွက် လိုချင်တဲ့ result ထွက်ပါတယ်။

# Chapter 13

Formatted Output

Many Ways for a Nicer Output

Python မှာ လှလှပပ print လုပ်ဖို့နည်းလမ်းတွေအများကြီးရှိပါတယ်။ string class ရဲ့ format method ကိုတော့သုံးသင့်ပါတယ်။

Value နှစ်ခုထက်ပိုတာတွေ print ထုပ်ချင်ရင် နှစ်မျိုးသုံးခွဲကြပါတယ်။

- အလွယ်ဆုံးကတော့ values တွေကြားမှာ , ခံပြီး print ထုတ်တာဖြစ်ပါတယ်။ default အနေနဲ့တော့ blank တွေကြားထဲမှာထည့်ပေးပါတယ် ။ sep keyword သုံးပြီး ကြိုက်တာသတ် မှတ်ချင်လည်းရပါတယ်။

```
>>> q = 459
>>> p = 0.098
>>> print(q, p, p * q)
459 0.098 44.982
>>> print(q, p, p * q, sep=",")
459,0.098,44.982
>>> print(q, p, p * q, sep=" :-) ")
459 :-) 0.098 :-) 44.982
>>>
```

- နောက်တစ်နည်းကတော့ string concatenation operator သုံးတာဖြစ်ပါတယ်။

```
>>> print(str(q) + " " + str(p) + " " + str(p * q))
459 0.098 44.982
>>>
```

ဒုတိယနည်းကပိုကောင်းပါတယ်။

The Old Way or the non-existing printf and sprintf

Python မှာ c လို printf မပါပေမယ့် printf function တွေပါပါတယ်။အဲ့ဒီအတွက် modulo operator % ကို string class က string formatting တွေလုပ်ဖို့ overload လုပ်ထားပါတယ်။ တစ်ခါတစ်ရံ string modulo operator လို့လည်းခေါ်ကြပါတယ်။ နောက်တစ်ခုက string interpolation လို့လည်းပြောကြပါတယ်။ ဘာလို့လည်းဆိုတော့ အမျိုးမျိုးသော data type တွေ( int float and so on) တွေကို formatted string တွေအဖြစ်ပြောင်းလဲပေါင်းစပ် ပေးလို့ပေးလို့ဖြစ်ပါတယ်။ string interpolation သုံးပြီး ပြုလုပ်လိုပ်တဲ့ string တွေကို ထူခြားတဲ့အသုံးပြုပုံနဲ့သုံးကြပါတယ်။ database ထဲ ကို format မှန်မှန် ကန်ကန် ဖန်တီးဖို့လည်းသုံးပါတယ်။ ဒါပေမယ့် python 2.6 ကနေစတင်ပြီးတော့ ဒီနည်းလမ်းအစား string mehto format ကိုသုံးသင့်ပါတယ်။ ဆိုးတာက string module % က python 3 မှာလည်းပါသလို ကျယ်ကျယ်ပြန့်ပြန့်လည်းသုံးနေတုံးပါပဲ။

```
>>>print("Art:%5d,price per Unit : %8.2f" %(453,59.058))
Art:   453,price per Unit :     59.06
```

Formatted place holder ရဲ့ရေးတဲ့ပုံစံက

%[flags][width][.precision]type

Eg ရေးပြထားတဲ့ဆီမှာ ဒုတိယ "%8.2f" က float number အတွက်ပါ။ တစ်ခြား place holder တွေလိုပဲ % နဲ့စပါတယ်။ နောက်မှာက string မှာပါမယ့် digit အရေအတွက် . နောက်မှာပါတာကတော့ float point နောက်က number အရေအတွက်ဖြစ်ပါတယ်။ f ကတော့ place holder သည် float ဖြစ်ကြောင်းပြောပါတယ်။

Output ကိုကြည့်မယ်ဆိုရင် digit 3 ခုကို round ယူထားတာတွေ့ပါမယ်။ ပြီးတော့ရှေ့မှာ blank 3 နေရာလည်းပါပါမယ်။

"%5d" ကတော့ content tuple ထဲက 453 အတွက်ပါ။ character 5 လုံးနဲ့ရိုက်ပါမယ်။ 453 က 3 လုံးပဲပါတော့ ရှေ့မှာ black 2 လုံးပါပါမယ်။

| Conversion | Meaning |
|---|---|
| d | Signed integer decimal. |
| i | Signed integer decimal. |
| o | Unsigned octal. |
| u | Obsolete and equivalent to 'd', i.e. signed integer decimal. |
| x | Unsigned hexadecimal (lowercase). |
| X | Unsigned hexadecimal (uppercase). |
| e | Floating point exponential format (lowercase). |
| E | Floating point exponential format (uppercase). |
| f | Floating point decimal format. |
| F | Floating point decimal format. |
| g | Same as "e" if exponent is greater than -4 or less than precision, "f" otherwise. |
| G | Same as "E" if exponent is greater than -4 or less than precision, "F" otherwise. |
| c | Single character (accepts integer or single character string). |
| r | String (converts any python object using repr(). |
| s | String (converts any python object using str(). |

| Conversion | Meaning |
|---|---|
| % | No argument is converted, results in a "%" character in the result. |

Examples

```
>>> print("%10.3e"% (356.08977))
 3.561e+02
>>> print("%10.3E"% (356.08977))
 3.561E+02
>>> print("%10o"% (25))
        31
>>> print("%10.3o"% (25))
       031
>>> print("%10.5o"% (25))
     00031
>>> print("%5x"% (47))
   2f
>>> print("%5.4x"% (47))
 002f
>>> print("%5.4X"% (47))
 002F
>>> print("Only one percentage sign: %% " % ())
Only one percentage sign: %
>>>
```

| Flag | Meaning |
|---|---|
| # | Used with o, x or X specifiers the value is preceded with 0, 0o, 0O, 0x or 0X respectively. |
| 0 | The conversion result will be zero padded for numeric values. |
| - | The converted value is left adjusted |
| | If no sign (minus sign e.g.) is going to be written, a blank space is inserted before the value. |
| + | A sign character ("+" or "-") will precede the conversion (overrides a "space" flag). |

Examples:

```
>>> print("%#5X"% (47))
 0X2F
>>> print("%5X"% (47))
   2F
>>> print("%#5.4X"% (47))
0X002F
>>> print("%#5o"% (25))
 0o31
>>> print("%+d"% (42))
+42
>>> print("% d"% (42))
 42
>>> print("%+2d"% (42))
+42
```

```
>>> print("% 2d"% (42))
 42
>>> print("%2d"% (42))
42
```

String modulo operator % ကို print နဲ့တွဲသုံးနေပေမယ့် မဆိုင်ကြပါဘူး ၊ string တစ်ခုကို string modulo operator နဲ့ apply လုပ်ရင် string ထုတ်ပေးမှာပဲဖြစ်ပါတယ်။

```
>>> s = "Price: $ %8.2f"% (356.08977)
>>> print(s)
Price: $   356.09
>>>
```

The Pythonic Way: The string method "format"

Format method ကို python 2.6 မှာ စထည့်ခဲ့ပါတယ်။ ပုံစံက

```
template.format(p0, p1, ..., k0=v0, k1=v1, ...)
```

template ဆိုတာက format codes တွေပါတဲ့ string ဖြစ်ပြီးတော့ text တွေထဲမှာထည့်ထားမှာ ဖြစ်ပါတယ်။ အစားထိုးမယ့် စာတွေကိုတော့ {} ထဲမှာ ထည့်ထားပါတယ်။ {} နဲ့ အတူ အထဲမှာ ရှိတဲ့ code တွေကို formatted value ရှိတဲ့ arugument နဲ့ အစားထိုးမှာဖြစ်ပါတယ်။ {} ထဲမှမပါတာတွေကတော့ ပုံမှန် စာကြောင်းလိုပဲ ပြောင်းလဲမှုမရှိ print လုပ်ပါမယ်။ {} တွေ print လုပ်ဖို့ လိုလာရင်တော့ 2 ခုရေးပြီး escape လုပ်ပေးရပါမယ် {{ နဲ့ }}

.format method အတွက် argument နှစ်မျိုးရှိပါတယ်။ 0 ကနေစတဲ့ positional arugment list (p0,p1,….) အဲ့နောက်မှာ keyword argument တွေက name=value ပုံစံနဲ့ရှိပါမယ်။

.format() method ရဲ့ Positional parameter တွေကို လိုချင်ရင် eg.{0} {1}စသဖြင့် အသုံးပြုနိုင်ပါတယ်။ string modulo operator % မှာလိုပဲ index နောက်မှာ : ခံပြီး format string တွေ ထည့်နိုင်ပါသေးတယ်။ eg {0:5d}

Format method ထဲ က positional parameter တွေကို အစဉ်အတိုင်းပဲဆိုရင် {} ထဲက positional argument specifier တွေကို မရေးလဲရပါတယ်။ ဆိုလိုတာက "{}{}{}" ဆိုတာ "{0}{1}{2}" ပါပဲ။ နေရာတလွဲတွေထားရင်တော့လိုပါတယ်။

```
>>>"Art:{0:5d}, Price per Unit:{1:8.2f}".format(453,59.058)
Art:  453,  Price per Unit:   59.06
```

Examples

```
>>> "First argument: {0}, second one: {1}".format(47,11)
'First argument: 47, second one: 11'
>>> "Second argument: {1}, first one: {0}".format(47,11)
'Second argument: 11, first one: 47'
>>> "Second argument: {1:3d}, first one: {0:7.2f}".format(47.42,11)
'Second argument:  11, first one:   47.42'
>>> "First argument: {}, second one: {}".format(47,11)
```

```
'First argument: 47, second one: 11'
>>> # arguments can be used more than once:
...
>>> "various precisions: {0:6.2f} or {0:6.3f}".format(1.4148)
'various precisions:   1.41 or  1.415'
>>>
```

အောက်ပါ ဥပမာမှာ keyword parameter သုံးတဲ့ပုံစံပြထားပါတယ်။

```
>>> "Art: {a:5d},  Price: {p:8.2f}".format(a=453, p=59.058)
'Art:   453,  Price:    59.06'
>>>
```

Format method နဲ့ left right justify လုပ်နိုင်ပါတယ်။ left justify အတွက် "<" right justify အတွက် ">" ကို : နောက်က format string အစမှာထည့်ပေးနိုင်ပါတယ်။

```
>>> "{0:<20s} {1:6.2f}".format('Spam & Eggs:', 6.99)
'Spam & Eggs:           6.99'
>>> "{0:>20s} {1:6.2f}".format('Spam & Eggs:', 6.99)
'        Spam & Eggs:   6.99'
```

| Option | Meaning |
|---|---|
| '<' | The field will be left-aligned within the available space. This is usually the default for strings. |
| '>' | The field will be right-aligned within the available space. This is the default for numbers. |
| '0' | If the width field is preceded by a zero ('0') character, sign-aware zero-padding for numeric types will be enabled.<br>`>>> x = 378`<br>`>>> print("The value is {:06d}".format(x))`<br>`The value is 000378`<br>`>>> x = -378`<br>`>>> print("The value is {:06d}".format(x))`<br>`The value is -00378` |
| ',' | This option signals the use of a comma for a thousands separator.<br>`>>> print("The value is {:,}".format(x))`<br>`The value is 78,962,324,245`<br>`>>> print("The value is {0:6,d}".format(x))`<br>`The value is 5,897,653,423`<br>`>>> x = 5897653423.89676`<br>`>>> print("The value is {0:12,.3f}".format(x))`<br>`The value is 5,897,653,423.897` |
| '=' | Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form "+000000120". This alignment option is only valid for numeric types. |
| '^' | Forces the field to be centered within the available space. |

Minimum field width မသတ်မှတ်ထားရင် field width အမြဲတူနေတဲ့အတွက်ကြောင့် alignment option က အသုံးမဝင်ပါဘူး။

Number type တွေအတွက်လည်း sign option ရှိပါသေးတယ်။

| Option | Meaning |
|---|---|
| '+' | indicates that a sign should be used for both positive as well as negative numbers. |
| '-' | indicates that a sign should be used only for negative numbers, which is the default behavior. |
| space | indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers. |

Using dictionaries in "format"

Format လုပ်မယ့် value ကိုနှစ်မျိုး access လုပ်နိုင်ကြောင်းပြောခဲ့ဖြပီးဖြစ်ပါတယ်။

- sing the position or the index:

```
>>> print("The capital of {0:s} is {1:s}".format("Ontario","Toronto"))
The capital of Ontario is Toronto
>>>
```

{}ချည်းပဲရေးချင်လည်းရပါတယ်။

- Using keyword parameters:

```
>>> print("The capital of {province} is
{capital}".format(province="Ontario",capital="Toronto"))
The capital of Ontario is Toronto
>>>
```

ဒုတိယနည်းကို dictionary သုံးပြီးလုပ်လို့ရပါသေးတယ်။

```
>>> data = dict(province="Ontario",capital="Toronto")
>>> data
{'province': 'Ontario', 'capital': 'Toronto'}
>>> print("The capital of {province} is {capital}".format(**data))
The capital of Ontario is Toronto
```

Asterisk * နှစ်ခုထည့်ထားရင် data ကို 'province="Ontario",capital="Toronto"' ပုံစံ ပြောပြောင်းပေးပါတယ်။

```
capital_country = {"United States" : "Washington",
                   "US" : "Washington",
                   "Canada" : "Ottawa",
                   "Germany": "Berlin",
                   "France" : "Paris",
                   "England" : "London",
                   "UK" : "London",
                   "Switzerland" : "Bern",
                   "Austria" : "Vienna",
```

```
                    "Netherlands" : "Amsterdam"}

print("Countries and their capitals:")
for c in capital_country:
    print("{country}: {capital}".format(country=c,
capital=capital_country[c]))
```

result က

```
Countries and their capitals:
United States: Washington
Canada: Ottawa
Austria: Vienna
Netherlands: Amsterdam
Germany: Berlin
UK: London
Switzerland: Bern
England: London
US: Washington
France: Paris
```

Dictionary နဲ့ရေးလည်းရပါတယ်။

```
capital_country = {"United States" : "Washington",
                   "US" : "Washington",
                   "Canada" : "Ottawa",
                   "Germany": "Berlin",
                   "France" : "Paris",
                   "England" : "London",
                   "UK" : "London",
                   "Switzerland" : "Bern",
                   "Austria" : "Vienna",
                   "Netherlands" : "Amsterdam"}

print("Countries and their capitals:")
for c in capital_country:
    format_string = c + ": {" + c + "}"
    print(format_string.format(**capital_country))
```

Using Local Variable Names in "format"

"local"ဆိုတာ function တစ်ခုပါ ။ local scope ထဲက variable တွေကို dictionary အနေနဲ့ return ပြန်ပေးပါတယ်။ local variable တွေက dictionary အတွက် key ဖြစ်ပြီး value ကတော့ ၎င်း variable value တွေဖြစ်ပါတယ်။

```
>>> a = 42
>>> b = 47
>>> def f(): return 42
...
>>> locals()
{'a': 42, 'b': 47, 'f': <function f at 0xb718ca6c>, '__builtins__': <module
'builtins' (built-in)>, '__package__': None, '__name__': '__main__',
'__doc__': None}
>>>
```

Locals() ကရလာတဲ့ dictionary ကို string format မှာသုံးနိုင်တယ်။

```
>>> print("a={a}, b={b} and f={f}".format(**locals()))
a=42, b=47 and f=<function f at 0xb718ca6c>
```

Other string methods for Formatting

String class မှာ အခြား method တွေရှိပါသေးတယ်။ ၎င်း တို့ကို format string လုပ်ဖို့သုံးနိုင်ပါတယ်။ ljust, rjust, center and zfill

S ဆိုတဲ့ string တစ်ခုနဲ့စမ်းကြည့်ပါမယ်။

- enter(...):

```
S.center(width[, fillchar]) -> str
```

Return S centred in a string of length width. Padding is done using the specified fill character. The default value is a space.

Examples:

```
>>> s = "Python"
>>> s.center(10)
'  Python  '
>>> s.center(10,"*")
'**Python**'
```

- ljust(...):

```
 S.ljust(width[, fillchar]) -> str
```

Return S left-justified in a string of length "width". Padding is done using the specified fill character. If none is given, a space will be used as default.

Examples:

```
>>> s = "Training"
>>> s.ljust(12)
'Training    '
>>> s.ljust(12,":")
'Training::::'
>>>
```

- rjust(...):

```
S.rjust(width[, fillchar]) -> str
```

Return S right-justified in a string of length width. Padding is done using the specified fill character. The default value is again a space.

Examples:

```
>>> s = "Programming"
>>> s.rjust(15)
'    Programming'
>>> s.rjust(15, "~")
'~~~~Programming'
>>>
```

- zfill(...):

```
S.zfill(width) -> str
```

Pad a string S with zeros on the left, to fill a field of the specified width. The string S is never truncated. This method can be easily emulated with rjust.

Examples:

```
>>> account_number = "43447879"
>>> account_number.zfill(12)
'000043447879'
>>> # can be emulated with rjust:
...
>>> account_number.rjust(12,"0")
'000043447879'
>>>
```

## Formatted String Literals

Python 3.6 မှာ formatted string leterals များပါလာပါတယ်။ 'f' နဲ့စပါတယ်။ str.format() ကလက်ခံ တဲ့ formatting syntax နဲ့တူပါတယ်။ format method ရဲ့ format string လိုပဲ {} နဲ့ အစားထိုးမယ့်နေရာတွေပါတယ်။ eg.

```
>>> price = 11.23
>>> f"Price in Euro: {price}"
'Price in Euro: 11.23'
>>> f"Price in Swiss Franks: {price * 1.086}"
'Price in Swiss Franks: 12.195780000000001'
>>> f"Price in Swiss Franks: {price * 1.086:5.2f}"
'Price in Swiss Franks: 12.20'
>>> for article in ["bread", "butter", "tea"]:
...     print(f"{article:>10}:")
...
     bread:
    butter:
       tea:
```

# Chapter 14

## Output With Print

Program တွေအားလုံးနီးပါး အပြင်လောက နဲ့ဆက်သွယ် ပြီး input>>process>>output ပုံစံ အလုပ်လုပ်ကြပါတယ်။ result ကိုတော့ ထုပ်ပြရမှာဖြစ်ပါတယ်။ python မှာ မြောက်များစွာထဲမှ တစ်ခုကတော့ print ဖြစ်ပါတယ်။

```
>>> print "Hello User"
Hello User
>>> answer = 42
>>> print "The answer is: " + str(answer)
The answer is: 42
>>>
```

() ထဲမှာ parameter တွေလည်းထည့်ပေးနိုင်ပါတယ်။

```
>>> print("Hallo")
Hallo
>>> print("Hallo","Python")
('Hallo', 'Python')
>>> print "Hallo","Python"
Hallo Python
>>>
```

Output ထုတ်ပေးတဲ့ပုံစံလည်းပြောင်းသွားပါတယ်။ အရေးကြီးတာကတော့ python version 2.x နဲ့ 3.x လည်းကွာပါတယ်။

```
$ python3
Python 3.2.3 (default, Apr 10 2013, 05:03:36)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello")
Hello
>>> print("Hello","Python")
Hello Python
>>>
```

Python 3 မှာလိုပဲဖြစ်ချင်ရင်တော့ "future" ဆီကနေ import လုပ်ပြီးသုံးရပါမယ်။

Import from future: print_function

Python program အချို့မှာ အောက်ပါline လေးပါပါလိမ့်မယ့်။

```
from __future__ import print_function
```

မရှင်းတာတွေရှိပါလိမ့်မယ်။ "print_function" ကို import လုပ်နေတယ်ထင်ပါလိမ့်မယ်။ အမှန်တော့ flag လေးတစ်ခုလုပ်လိုက်တာပါ။ ဒီလို flag လေးလုပ်လိုက်ရင် interpreter က print function ကိုသုံးလို့ရအောင်လုပ်ပေးပါတယ်။

Import လုပ်ပေးတာကို python 3 အတွက်လည်း အဆင်ပြေစေတာကြောင့် လုပ်သင့်ပါတယ်။

Note//python n 3 မှာ print statement မပါတော့ပဲ print function ပဲပါပါတော့တယ်။

# Chapter 15

Sequential Data Types

Strings

String တွေ character အတွဲလိုက်တွေအနေနဲ့မြင်နိုင်ပါတယ်။အမျိုးမျိုးဖော်ပြနိုင်ပါတယ်။

- single quotes (')
  'This a a string in single quotes'
- double quotes (") "Miller's dog bites"
- triple quotes(''') or (""") '''She said: "I don't mind, if Miller's dog bites"'''

Indexing strings

"Hello World" ဆိုတဲ့ string တစ်ခုမှာ character တွေကို ဘယ်မှညာသို့ 0 ကနေစပြီး enumeration လုပ်ပါတယ်။ ညာဘက်ကနေဆိုရင် -1 ကစပါတယ်။

String ထဲက character တစ်ခုခြင်းဆီကို string name နဲ့ [index no] သုံးပြီး အသံအသုံးပြု နိုင်ပါတယ်။

```
>>> txt = "Hello World"
>>> txt[0]
'H'
>>> txt[4]
'o'
```

Negative index value တွေလည်းအသုံးပြုနိုင်တယ်။ညာဘက်ကနေ -1 နဲ့စပါတယ်။

```
>>> txt[-1]
'd'
>>> txt[-5]
'W'
```

Python Lists

Pythonမှာ list ကတော့ versatile data type တစ်ဖြစ်ပါတယ။ versatile ဆိုတာကတော့ type အမျိုးပြောင်းလဲအသုံးပြုနိုင်တာဖြစ်ပါတယ်။eg. integer list , string list.  List တွေရေးတဲ့အခါ [ [] ထဲမှာ comma sepatater value(CSV) အနေနဲ့ရေးနိုင်ပါတယ်။ c,c++,java တို့ဆီက array နဲ့တူပေမဲ့ ပုံမှန် classical array တွေထက်ပိုပြီး flexible ဖြစ်ပါတယ်။ ဥပမာ list ထဲက item တွေက type မတူလည်းရပါတယ်။ ပြီးတော့ list size က ထပ်ချဲ့လို့ရပါတယ်။ array မှာလို အသေမဟုတ်ပါဘူး။

An example of a list:

```
languages = ["Python", "C", "C++", "Java", "Perl"]
```

list itemတွေကို သုံးစွဲဖို့(access)လုပ်ဖို့ နည်းအမျိုးမျိုးရှိပါတယ်။ အလွယ်ဆုံးကတော့ index သုံးတာပါ။

```
>>> languages = ["Python", "C", "C++", "Java", "Perl"]
>>> languages[0]
'Python'
>>> languages[1]
'C'
>>> languages[2]
'C++'
>>> languages[3]
'Java'
```

ခုပြထားတဲ့ဆီမှာ data type တူပါတယ်။ မတူလဲရပါတယ်။

```
group = ["Bob", 23, "George", 72, "Myriam", 29]
```

## Sublists

List ထဲမှာ sublist တွေရှိနိုင်ပါတယ်။ sublist ထဲမှာ sublist ရှိနိုင်ပါသေးတယ်။

```
>>> person = [["Marc","Mayer"],["17, Oxford Str", "12345","London"],"07876-
7876"]
>>> name = person[0]
>>> print name
['Marc', 'Mayer']
>>> first_name = person[0][0]
>>> print first_name
Marc
>>> last_name = person[0][1]
>>> print last_name
Mayer
>>> address = person[1]
>>> street = person[1][0]
>>> print street
17, Oxford Str
```

ပိုမိုရှုပ်ထွေးတဲ့ပုံစံလေးပြထားပါတယ်။

```
>>> complex_list = [["a",["b",["c","x"]]]]
>>> complex_list = [["a",["b",["c","x"]]],42]
>>> complex_list[0][1]
['b', ['c', 'x']]
>>> complex_list[0][1][1][0]
'c'
```

## Tuples

Tuples တွေကတော့ immutable list တွေဖြစ်ပါတယ်။ တည်ဆောက်ပြီးရင် ပြောပြောင်းလဲလို့မရပါဘူး tuples နဲ့ list နဲ့သဘောတရားအနေနဲ့တူပေမဲ့ tuple က () သုံးပြီး list က [] သုံးပါတယ်။ index rule တွေလည်းအတူတူပါပဲ ။ tuple တစ်ခုတည်ဆောက်ပြီးရင် item တွေထည့်လို့ဖြုတ်လို့ မရတော့ပါဘူး

Tuple အကျိုးကျေးဇူးများ

- Tuple က list ထက်မြန်ပါတယ်။
- မိမိရဲ့ data ကပြောင်းစရာမလိုတော့ဘူးဆိုတာသိရင် tuple သုံးသင့်ပါတယ်။ မတော်တဆ data ပြောင်းသွားတာမျိုးကနေကာကွယ်နိုင်ပါတယ်။
- Tuple ကို dictionary တွေမှာ key အနေနဲ့သုံးနိုင်ပါတယ်။ list ကတော့မရပါဘူး

Tuple examples/ data ပြောင်းလဲဖို့ကြိုးစားတဲ့အခါ error ပေးမှာပါ။

```
>>> t = ("tuples", "are", "immutable")
>>> t[0]
'tuples'
>>> t[0]="assignments to elements are not possible"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Generalization

List တွေ string တွေမှာ common properties တွေရှိပါတယ်။ eg.list item တွေ string character တွေကို အစဉ်လိုက်ရှိပြီးတော့ Index နဲ့အသုံးပြုနိုင်ပါတယ်။ tuple လိုတစ်ခြား data type တွေရှိပါသေးတယ်။(buffer and xrange) "sequence data types" or "sequential data types"လို့ခေါ်ပါတယ်။

"sequence data types" အတွက် operator တွေ method တွေကတော့ အတူတူပါပဲ။

Slicing

String ထဲကတစ်စိတ်တစ်ပိုင်း , list ထဲက တစ်စိတ်တစ်ပိုင်း လိုချင်ရင် slice operator သုံးပါတယ်။ range တစ်ခုနဲ့ ယူတဲ့သဘောပါပဲ ။ eg

```
>>> str = "Python is great"
>>> first_six = str[0:6]
>>> first_six
'Python'
>>> starting_at_five = str[5:]
>>> starting_at_five
'n is great'
>>> a_copy = str[:]
>>> without_last_five = str[0:-5]
>>> without_last_five
'Python is '
>>>
```

List မှာလဲဒီသဘောပါပဲ

```
>>> languages = ["Python", "C", "C++", "Java", "Perl"]
>>> some_languages = languages[2:4]
>>> some_languages
['C++', 'Java']
>>> without_perl = languages[0:-1]
>>> without_perl
['Python', 'C', 'C++', 'Java']
>>>
```

Slicing ကို argument သုံးခုနဲ့သုံးချင်ရင်လည်းရပါတယ်။ step ပါ

```
s[begin: end: step]
```

third parameter ပဲ ပါရင် အစကနေအဆုံးကို third parameter ပေးထားတဲ့အတိုင်း ယူသွားမှာပါ။

```
>>> str = "Python under Linux is great"
>>> str[::3]
'Ph d n  e'
```

ဒီမှာ 3 ခုခြား ယူတာတွေ့မှာပါ။

Length

Sequence ရဲ့ length ကိုပြောတာပါ။ list, string , tuple တွေရဲ့ length ကို len() သုံးပြီးသိနိုင်ပါတယ်။ string မှာဆိုရင် character တွေကိုရေတွက်ပြီးတော့ list,tuple တွေမှာတော့ element တွေကိုရေတွက်ပါတယ်။ sublist တွေကို 1 element အနေနဲ့ရေတွက်ပါတယ်။

```
>>> txt = "Hello World"
>>> len(txt)
11
>>> a = ["Swen", 45, 3.54, "Basel"]
>>> len(a)
4
```

Concatenation of Sequences

Sequences တွေကိုပေါင်းစပ်တာ ဟာ ဂဏန်း နှစ်လုံးပေါင်းသလိုပါပဲ။ operator sign လည်းတူပါတယ်။

```
>>> firstname = "Homer"
>>> surname = "Simpson"
>>> name = firstname + " " + surname
>>> print name
Homer Simpson
>>>
```

List အတွက်လည်းအတူတူပါပဲ

```
>>> colours1 = ["red", "green","blue"]
>>> colours2 = ["black", "white"]
>>> colours = colours1 + colours2
```

```
>>> print colours
['red', 'green', 'blue', 'black', 'white']
```

"+=" operator ကလည်း sequences မှာအလုပ်လုပ်ပါတယ်။

```
s += t
```

အောက်ပါပုံစံနဲ့အတူတူပါပဲ

```
s = s + t
```

စာအနေနဲ့တူပေမဲ့ နောက်ကွယ်ကအလုပ်လုပ်ပုံတော့ကွဲပါတယ်။ ပထမမှာတော့ ဘယ်ဖက်အတွက် တစ်ခါပဲကြည့်စရာ (evaluate)လိုပါတယ်။ argument assignment ကို mutable object တွေမှာ optimizing ကောင်းဖို့သုံးကြပါတယ်။

Checking if an Element is Contained in List

Item တစ်ခု sequence ထဲမှာပါမပါ စစ်ဖို့လွယ်ပါတယ်။ "in" သို့ "not in" operator တွေသုံးနိုင်ပါတယ်။

```
>>> abc = ["a","b","c","d","e"]
>>> "a" in abc
True
>>> "a" not in abc
False
>>> "e" not in abc
False
>>> "f" not in abc
True
>>> str = "Python is easy!"
>>> "y" in str
True
>>> "x" in str
False
>>>
```

Repetitions

Sequences အတွက် + operator တွေသုံးခဲ့တယ်။ "*" လည်းရှိပါတယ်။ မြောက်တာမဟုတ်ပါဘူး အကြိမ်အရေအတွက်တစ်ခု ထပ်ခါထပ်ခါပွားရင်သုံးပါတယ်။

```
str * 4
```

ဆိုတာက

```
str + str + str + str
```

Further examples:

```
>>> 3 * "xyz-"
'xyz-xyz-xyz-'
```

```
>>> "xyz-" * 3
'xyz-xyz-xyz-'
>>> 3 * ["a","b","c"]
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']
```

"*=" လည်းသုံးနိုင်ပါတယ်။

The Pitfalls of Repetitions

Repetition ကို ရိုးရိုး list မှာသုံးနိုင်သလို nested list အတွက်လည်းသုံးနိုင်ပါတယ်။

```
>>> x = ["a","b","c"]
>>> y = [x] * 4
>>> y
[['a', 'b', 'c'], ['a', 'b', 'c'], ['a', 'b', 'c'], ['a', 'b', 'c']]
>>> y[0][0] = "p"
>>> y
[['p', 'b', 'c'], ['p', 'b', 'c'], ['p', 'b', 'c'], ['p', 'b', 'c']]
```

ရလာတဲ့ result က စလေ့လာတဲ့သူဆိုရင် အံ့သြစရာပါ။ y ရဲ့ first sublist ရဲ့ first element ကို တန်ဖိုး အသစ်ထည့်လိုက်ပါတယ်။ ဆိုလိုတာက y[0][0] ကို ပြောင်းလိုက်တဲ့အခါ တစ်ခြား sublist ရဲ့ first element တွေ y[1][0], y[2][0], y[3][0] တွေပါလိုက်ပြောင်းသွားပါတယ်။

အကြောင်းပြချက်ကတော့ * operator က list အတွက် reference 4 ခုလုပ်လိုက်လို့ပါ။ ဒါကြောင့် ကို က တစ်ခု ပဲပြောင်းတယ်ထင်ပေမယ့် 4 ခုကreference လုပ်ထားတဲ့အတွက် 4 ခုလုံးပြောင်းသွား တာပါ။

# Chapter 16

Dictionaries

List တွေအကြောင်းပြီးတော့ dictionary တွေအကြောင်းပါ။ Dictionary က list လိုပဲ runtime မှာ ပြောင်းလဲတာတွေလုပ်နိုင်ပါတယ်။ list ထဲမှာ dictionary ရှိနိုင်သလို dictionary ထဲမှာ လည်း list တွေပါဝင်နိုင်ပါတယ်။ကွာခြားချက်က list တွေက ordered sets တွေဖြစ်ပြီးတော့ dictionary တွေကတော့ unordered sets တွေဖြစ်ပါတယ်။ ပြီးတော့ dictionary တွေမှာ item တွေကို position နဲ့မဟုတ်ပဲ key နဲ့ access လုပ်တာဖြစ်ပါတယ်။Associative array (hashes) လို့လည်းခေါ်ပါတယ်။ dictionary မှာ key နဲ့ value ကိုmapping ထားပါတယ်။ dicationay value က python data type ကြိုက်တာဖြစ်နိုင်ပါတယ်။ dictionary တွေက unordered key-value-pairs တွေဖြစ်ပါတယ်။

Dictionary တွေက string , list , tuple တွေလို sequence operation လုပ်မရပါဘူး ။

Examples of Dictionaries

Dictionary အလွတ်တစ်ခု

```
>>> empty = {}
>>> empty
{}
```

Eg continue

```
>>> food = {"ham" : "yes", "egg" : "yes", "spam" : "no" }
>>> food
{'egg': 'yes', 'ham': 'yes', 'spam': 'no'}
>>> food["spam"] = "yes"
>>> food
{'egg': 'yes', 'ham': 'yes', 'spam': 'yes'}
>>>
```

German-English dictionary:

```
# -*- coding: iso-8859-1 -*-

en_de = {"red" : "rot", "green" : "grün", "blue" : "blau", "yellow":"gelb"}
print en_de
print en_de["red"]
```

ပထမline က german စကားလုံးတွေအတွက်ပါ။ ပေးထားတဲ့ code type နဲ့ save ပေးရပါမယ်။

German-french

```
# -*- coding: iso-8859-1 -*-

en_de = {"red" : "rot", "green" : "grün", "blue" : "blau", "yellow":"gelb"}
print en_de
print en_de["red"]
de_fr = {"rot" : "rouge", "grün" : "vert", "blau" : "bleu", "gelb":"jaune"}
print "The French word for red is: " + de_fr[en_de["red"]]
```

value ကိုကြိုက်တဲ့ data type ထားနိုင်ပေမယ့် key အတွက်တော့ ကန့်သတ်ချက်တွေရှိပါတယ်။ Immutable data type တွေကိုပဲ key အနေနဲ့သုံးနိုင်ပါတယ်။ Mutable data type သုံးရင် error ပေးပါလိမ့်မယ်။

```
>>> dic = { [1,2,3]:"abc"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list objects are unhashable
```

Tuples တွေကိုလည်း key အနေနဲ့သုံးနိုင်ပါတယ်.

```
>>> dic = { (1,2,3):"abc", 3.1415:"abc"}
>>> dic
{3.1415: 'abc', (1, 2, 3): 'abc'}
```

More

```
# -*- coding: iso-8859-1 -*-

en_de = {"red" : "rot", "green" : "grün", "blue" : "blau", "yellow":"gelb"}
de_fr = {"rot" : "rouge", "grün" : "vert", "blau" : "bleu", "gelb":"jaune"}

dictionaries = {"en_de" : en_de, "de_fr" : de_fr }
print dictionaries["de_fr"]["blau"]
```

Operators on Dictionaries

| Operator | Explanation |
|---|---|
| len(d) | returns the number of stored entries, i.e. the number of (key,value) pairs. |
| del d[k] | deletes the key k together with his value |
| k in d | True, if a key k exists in the dictionary d |
| k not in d | True, if a key k doesn't exist in the dictionary d |

Accessing non Existing Keys

မရှိတဲ့ key ကို access လုပ်ရင် error ပြပါလိမ့်မယ်.

```
>>> words = {"house" : "Haus", "cat":"Katze"}
>>> words["car"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'car'
```

"in" operator နဲ့ error မတတ်အောင်လုပ်နိုင်ပါတယ်။

```
>>> if "car" in words: print words["car"]
...
>>> if "cat" in words: print words["cat"]
...
Katze
```

Important Methods

Dictionary တွေကို copy() method သုံးပြီး ကူးနိုင်ပါတယ်။

```
>>> w = words.copy()
>>> words["cat"]="chat"
>>> print w
{'house': 'Haus', 'cat': 'Katze'}
>>> print words
{'house': 'Haus', 'cat': 'chat'}
```

Clear() method နဲ့dictionary ထဲရှိတာတွေရှင်းနိုင်ပါတယ်။ dictionary ကိုဖျက်တာ မဟုတ်ပေမဲ့ empty dictionary ဖြစ်သွားပါတယ်။

```
>>> w.clear()
>>> print w
{}
```

Update: Merging Dictionaries

Dictionary တွေကိုပေါင်းစပ်နိုင်ပါတယ်။ key တူတာတွေပါရင်တော့ over write လုပ်မှာပါ။

```
>>> w={"house":"Haus","cat":"Katze","red":"rot"}
>>> w1 = {"red":"rouge","blau":"bleu"}
>>> w.update(w1)
>>> print w
{'house': 'Haus', 'blau': 'bleu', 'red': 'rouge', 'cat': 'Katze'}
```

Iterating over a Dictionary

Dictionary တွေကို iterate လုပ်ဖို့ သီးသန့် method တွေမလိုပါဘူး

```
for key in d:
        print key
```

iterkeys()ကိုတော့သုံးနိုင်ပါတယ်။

```
for key in d.iterkeys():
        print key
```

values တွေအတွက်ဆိုရင်တော့ itervalues() method ကအသုံးဝင်ပါတယ်။

```
for val in d.itervalues():
        print val
```

အတူတူပါပဲ

```
for key in d:
        print d[key]
```

Connection between Lists and Dictionaries

List ကနေ dictionary ၊ dictionary ကနေ list ပြောင်းတာတွေလုပ်နိုင်ပါတယ်။

dictionary

```
{"list":"Liste", "dictionary":"Wörterbuch", "function":"Funktion"}
```

List

```
[("list","Liste"),("dictionary","Wörterbuch"), ("function","Funktion")]
```

Lists from Dictionaries

Items(),keys(),values() methods တွေသုံးပြီး dictionary ကနေ list ပြောင်းလဲနိုင်ပါတယ်။

```
>>> w={"house":"Haus","cat":"Katze","red":"rot"}
>>> w.items()
[('house', 'Haus'), ('red', 'rot'), ('cat', 'Katze')]
>>> w.keys()
['house', 'red', 'cat']
>>> w.values()
['Haus', 'rot', 'Katze']
```

Dictionary from list

ဟင်းပွဲတွေနဲ့ သက်ဆိုင်ရာ နိုင်ငံ list နှစ်ခုရှိမယ်။

```
>>> dishes = ["pizza", "sauerkraut", "paella", "Hamburger"]
>>> countries = ["Italy", "Germany", "Spain", "USA"]
```

Zip() method သုံးပြီး list နှစ်ခုပေါင်း မယ်

```
>>> country_specialities = zip(countries, dishes)
>>> print country_specialities
[('Italy', 'pizza'), ('Germany', 'sauerkraut'), ('Spain', 'paella'),
('USA', 'Hamburger')]
>>>
```

အခု country_specialities အထဲမှာ 2-tuple list ပုံစံရှိနေမယ်။ဒီပုံစံကနေ dictionary အဖြစ် dict() function သုံးပြီးပြောင်းမယ်။

```
>>> country_specialities_dict = dict(country_specialities)
>>> print country_specialities_dict
{'Germany': 'sauerkraut', 'Spain': 'paella', 'Italy': 'pizza', 'USA':
'Hamburger'}
>>>
```

Zip() function နဲ့ပတ်သတ်ပြီး မေးစရာရှိတာက list တစ်ခုကတစ်ခြားတစ်ခုထက် element တွေပိုနေရင်ဘာဖြစ်မလဲ/ list size မတူရင်ဘာဖြစ်မလဲပေါ့။ လွယ်ပါတယ် ပိုနေတဲ့ဘက်ကဟာတွေ မသုံးပါဘူး။

```
>>> countries = ["Italy", "Germany", "Spain", "USA", "Switzerland"]
>>> dishes = ["pizza", "sauerkraut", "paella", "Hamburger"]
>>> country_specialities = zip(countries,dishes)
>>> print country_specialities
[('Italy', 'pizza'), ('Germany', 'sauerkraut'), ('Spain', 'paella'),
('USA', 'Hamburger')]
```

# Chapter 17

## Sets and Frozensets

### Sets in Python

"set" data type က collection type ဖြစ်ပြီးတော့ python 2.4 ကစပါလာပါတယ်။ set ထဲမှာ immutable object တွေပါပါတယ်။ Set data type က mathematic set ကို python အနေနဲ့ အသုံးပြုနိုင်အောင်ရေးသားထားတာပဲဖြစ်ပါတယ်။ ဒါကြောင့် set မှာ list တွေ tuple တွေလို element တွေ မထပ်ပါဘူး။

### Creating Sets

Set တစ်ခုဖန်တီးချင်ရင် sequence တစ်ခု သို့ Iterable obect တစ်ခုကို set function သုံးပြီး ဖန်တီးနိုင်ပါတယ်။

```
>>> x = set("A Python Tutorial")
>>> x
set(['A', ' ', 'i', 'h', 'l', 'o', 'n', 'P', 'r', 'u', 't', 'a', 'y', 'T'])
>>> type(x)
<type 'set'>
>>>
```

List တွေလည်း set function ထဲမှာသုံးနိုင်ပါတယ်။

```
>>> x = set(["Perl", "Python", "Java"])
>>> x
set(['Python', 'Java', 'Perl'])
>>>
```

တူညီတဲ့ item/element တွေပါရင်ဘာဖြစ်မလဲကြည့်ကြည့်ပါ။

```
>>> cities = set(("Paris", "Lyon", "London","Berlin","Paris","Birmingham"))
>>> cities
set(['Paris', 'Birmingham', 'Lyon', 'London', 'Berlin'])
>>>
```

ထပ်တဲ့ data တွေပျောက်သွားတာတွေ့မှာပါ။

### Immutable Sets

Set ထဲမှာ mutable object တွေပါလို့မရပါဘူး။ eg.

```
>>> cities = set((("Python","Perl"), ("Paris", "Berlin", "London")))
>>> cities = set((["Python","Perl"], ["Paris", "Berlin", "London"]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```

Frozensets

Set ထဲမှာ mutable Object ပါလို့မရပေမယ့်။ set ကတော့ mutable ဖြစ်ပါတယ်။

```
>>> cities = set(["Frankfurt", "Basel","Freiburg"])
>>> cities.add("Strasbourg")
>>> cities
set(['Freiburg', 'Basel', 'Frankfurt', 'Strasbourg'])
>>>
```

Frozen set ကတော့ ပြောင်းလဲလို့မရတာကလွဲလို့ set နဲ့အတူတူပါပဲ။

```
>>> cities = frozenset(["Frankfurt", "Basel","Freiburg"])
>>> cities.add("Strasbourg")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
>>>
```

Simplified Notation

Python 2.6 ကနေစပြီးတော့ set function မသုံးပဲ set တွေဖန်တီးနိုင်ပါတယ်။ {}သုံးပါတယ်။

```
>>> adjectives = {"cheap","expensive","inexpensive","economical"}
>>> adjectives
set(['inexpensive', 'cheap', 'expensive', 'economical'])
>>>
```

Set Operations

add(element)

element တွေထပ်ပေါင်းချင်ရင်သုံးပါတယ်။

```
>>> colours = {"red","green"}
>>> colours.add("yellow")
>>> colours
set(['green', 'yellow', 'red'])
>>> colours.add(["black","white"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
```

နဂိုရှိပြီးသား data နဲ့တူနေထာထပ်ထည့်ရင်တော့ ဘာမှဖြစ်မှာမဟုတ်ပါဘူး။

clear()

set ထဲက elements တွေရှင်းချင်ရင်သုံးပါတယ်။

```
>>> cities = {"Stuttgart", "Konstanz", "Freiburg"}
>>> cities.clear()
>>> cities
set([])
>>>
```

copy

shallow copy လုပ်ပေးပါတယ်.

```
>>> more_cities = {"Winterthur","Schaffhausen","St. Gallen"}
>>> cities_backup = more_cities.copy()
>>> more_cities.clear()
>>> cities_backup
set(['St. Gallen', 'Winterthur', 'Schaffhausen'])
>>>
```

Assignment operator လေးသုံးယုံနဲ့ရတယ်ထင်ပါလိမ့်မယ်။

```
>>> more_cities = {"Winterthur","Schaffhausen","St. Gallen"}
>>> cities_backup = more_cities
>>> more_cities.clear()
>>> cities_backup
set([])
```

"cities_backup = more_cities"ဆိုတဲ့ statement က data တစ်နေရာထည်းကိုပဲ ညွှန်းပေးသွား တာဖြစ်ပါတယ်။

difference()

set တွေကြားထဲက ခြားနားမှုကို set အသစ်တစ်ခုအနေနဲ့ ထုတ်ပေးပါတယ်။

```
>>> x = {"a","b","c","d","e"}
>>> y = {"b","c"}
>>> z = {"c","d"}
>>> x.difference(y)
set(['a', 'e', 'd'])
>>> x.difference(y).difference(z)
set(['a', 'e'])
>>>
```

"-" operator သုံးလည်းရပါတယ်။

```
>>> x - y
set(['a', 'e', 'd'])
>>> x - y - z
set(['a', 'e'])
>>>
```

difference_update()

Set တစ်ခုထဲက element ကို အခြား set တစ်ခုထဲက element တွေဖယ်ခြင်ရင်သုံးပါတယ် ။

x=x-y နဲ့တူပါတယ်။

```
>>> x = {"a","b","c","d","e"}
>>> y = {"b","c"}
>>> x.difference_update(y)
>>>
>>> x = {"a","b","c","d","e"}
>>> y = {"b","c"}
>>> x = x - y
>>> x
set(['a', 'e', 'd'])
>>>
```

discard(el)

element တစ်ခုကို set ထဲကနေဖယ်ရှားချင်ရင်သုံးပါတယ်။ မပါရင်တော့ ဘာမှမဖြစ်ပါဘူး။

```
>>> x = {"a","b","c","d","e"}
>>> x.discard("a")
>>> x
set(['c', 'b', 'e', 'd'])
>>> x.discard("z")
>>> x
set(['c', 'b', 'e', 'd'])
>>>
```

remove(el)

discard() နဲ့တူပါတယ်။ ဒါပေမယ့် သူက ဖယ်ချင်တဲ့ element မပါရင် KeyError ပေးပါတယ်။

```
>>> x = {"a","b","c","d","e"}
>>> x.remove("a")
>>> x
set(['c', 'b', 'e', 'd'])
>>> x.remove("z")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'z'
>>>
```

intersection(s)

Set တွေကြား တူညီမှုတွေကို set အသစ်လုပ်ပေးပါတယ်။

```
>>> x = {"a","b","c","d","e"}
>>> y = {"c","d","e","f","g"}
```

```
>>> x.intersection(y)
set(['c', 'e', 'd'])
>>>
```

"&" operator သုံးလည်းရပါတယ်။

```
>>> x = {"a","b","c","d","e"}
>>> y = {"c","d","e","f","g"}
>>> x.intersection(y)
set(['c', 'e', 'd'])
>>>
>>> x = {"a","b","c","d","e"}
>>> y = {"c","d","e","f","g"}
>>> x  & y
set(['c', 'e', 'd'])
>>>
```

isdisjoint()

Set နှစ်ခု intersection (တူညီတဲ့ element) မပါရင် True ထုတ်ပေးပါတယ်။

issubset()

x.issubset(y) က True ထုတ်ပေးပါလိမ့်မယ်။ x က y ရဲ့ subset ဆိုရင်ပေါ့ ။ "<=" ဆိုရင် subset of ">=" ဆိုရင် superset of "<" ကတော့ subset ဖြစ်မဖြစ်စစ်ချင်ရင်သုံးပါတယ်။ eg ကိုပိုရှင်းပါလိမ့်မယ်။

```
>>> x = {"a","b","c","d","e"}
>>> y = {"c","d"}
>>> x.issubset(y)
False
>>> y.issubset(x)
True
>>> x < y
False
>>> y < x # y is a proper subset of x
True
>>> x < x # a set can never be a proper subset of oneself.
False
>>> x <= x
True
>>>
```

issuperset()

သူကတော့ super set အတွက်ပါ။

```
>>> x = {"a","b","c","d","e"}
>>> y = {"c","d"}
>>> x.issuperset(y)
True
>>> x > y
True
>>> x >= y
True
>>> x >= x
True
>>> x > x
False
>>> x.issuperset(x)
True
>>>
```

pop()

Set element တွေကိုတစ်ခုချင်းထုတ်ပေး ဖယ်ပေးပါတယ်။ Empty ဖြစ်သွားရင်တော့ KeyError ထုတ်ပေးပါတယ်။

```
>>> x = {"a","b","c","d","e"}
>>> x.pop()
'a'
>>> x.pop()
'c'
```

# Chapter 18

Shallow and Deep Copy

Data Types and Variables အခန်းမှာပြောခဲ့သလို integer / string တွေလို data type ရိုးရိုးလေးတွေ assignment လုပ်တာ copy လုပ်တာမှာ တစ်ခြား programming language တွေနဲ့စာရင် python မှာ ထူခြားတဲ့ ပုံစံတွေရှိပါတယ်။ shallow နဲ့ deep copying ကတော့ list လို class instance(object) လို object ထဲမှာ object ပါတာတွေမှာ အသုံးပြုဖို့သင့်တော်ပါတယ်။

အောက်မှာပြထားသလို y က x နဲ့ memory location တစ်နေရာထဲကပြနေပါတယ်။ y ကိုတစ်ခြားတန်ဖိုး ပြောင်းလိုက်ရင်တော့ နေရာပြောင်းသွားပါတယ်။ "Data Types and Variables" ခန်းမှာပြောသလို Y ကmemory နောက်တစ်နေရာရသွားပါတယ်။

```
>>> x = 3
>>> y = x
```

အတွင်းပိုင်း(internal) လုပ်ဆောင်ချက်တွေ ကထူးဆန်းပေမဲ့ , ဘာ result ရလာမယ်ဆိုတာတော့ သိနိုင်ပါတယ်။list လို dictionary လို mutable object တွေ copy လုပ်ရင်တော့ ပြဿနာကတော့ရှိနိုင်ပါတယ်။

Programmer ကတောင်းဆိုမှသာ တကယ် copy ကူးပေးမှာဖြစ်ပါတယ်။

List တွေ dictionary လို က mutable object တွေ copy လုပ်ရင်ဘာဖြစ်မလဲကြည့်ရအောင်

Copying a list

```
>>> colours1 = ["red", "green"]
>>> colours2 = colours1
>>> colours2 = ["rouge", "vert"]
>>> print colours1
['red', 'green']
```

colours1 ထဲကို list တစ်ခုပြီး ထည့် colour2 ထဲကို colour1 assign လုပ်ထားပါတယ်။ ပြီးတော့ colour2 ထဲကို list အသစ်တစ်ခု assign လုပ်ပါတယ်။

မျှော်လင့်ထားသလိုပဲ colour1 တန်ဖိုးကမပြောင်းပါဘူး။ "Data types and variables" တုန်းက လိုပဲ colours2 အတွက် memory location အသစ်တစ်ခုပေးသွားပါတယ်။ ဘာလို့လဲဆိုတော့ ကျွန်တော်တို့ colours2 ကို assign ထပ်လုပ်လိုက်တာက list အသစ်တစ်ခုဖြစ်နေလို့ပါ။

```
>>> colours1 = ["red", "green"]
>>> colours2 = colours1
>>> colours2[1] = "blue"
>>> colours1
['red', 'blue']
```

ဒီမှာမေးစရာစလာပါပြီ။ colour2 သို့ colour1 ထဲက element တစ်ခုကိုပြောင်းလိုက်ရင်ရော?

ဒီအပေါ်က example မှာ colour2 ရဲ့ ဒုတိယ element ကို တန်ဖိုးအသစ်ပြောင်းလိုက်ပါတယ်။ colour1 ရဲ့ list လဲပြောင်းလဲသွားတာကိုလည်းတွေ့ပါလိမ့်မယ်။

List အသစ်တစ်ခု assign လုပ်တာမဟုတ်ပဲ။ list ထဲက element တစ်ခုကိုပဲ ပြောင်းလိုက်တာကြောင့် ဖြစ်ပါတယ်။

Copy with the Slice Operator

List တွေ copy ကူးဖို့ slice operator ကိုသုံးနိုင်ပါတယ်။

```
>>> list1 = ['a','b','c','d']
>>> list2 = list1[:]
>>> list2[1] = 'x'
>>> print list2
['a', 'x', 'c', 'd']
>>> print list1
['a', 'b', 'c', 'd']
>>>
```

ဒါပေမယ့် list မှာ sublist တွေပါလာရင်တော့ ပြဿနာရှိတုန်းပါပဲ ။ sublist တွေရဲ့ pointer တွေပဲပါလာတာမို့ပါ။

```
>>> lst1 = ['a','b',['ab','ba']]
>>> lst2 = lst1[:]
```

Lists တွေရဲ့ element တစ်ခုခြင်းကို ပြင်ရင် ပြဿနာမရှိပေမယ့် ။ sublist ထဲက element တစ်ခုကို ပြောင်းလဲမယ်ဆိုရင်တော့ ပြဿနာရှိပါတယ်။

```
>>> lst1 = ['a','b',['ab','ba']]
>>> lst2 = lst1[:]
>>> lst2[0] = 'c'
>>> lst2[2][1] = 'd'
>>> print(lst1)
['a', 'b', ['ab', 'd']]
```

Using the Method deepcopy from the Module copy

ဒီလိုမဖြစ်အောင် "copy" module ကိုသုံးရပါမယ်။ သူ့မှာ list တွေအတွက် "copy" method ပါပါတယ်။

```
from copy import deepcopy
lst1 = ['a','b',['ab','ba']]
lst2 = deepcopy(lst1)
lst2[2][1] = "d"
lst2[0] = "c";
print lst2
print lst1
```

file ကို save ပြီး cmd မှာ runကြည့်ပါ။

```
$ python deep_copy.py
['c', 'b', ['ab', 'd']]
['a', 'b', ['ab', 'ba']]
```

# Chapter 19

## Functions

### Syntax

Function တွေကတော့ structure program တွေအတွက် construct (ideal element )တွေဖြစ်ပါတယ်။ subroutines or procedure လို့လည်းခေါ်ကြပါတယ်။ function တွေမသုံးပဲ code တွေပြန်သုံးချင်ရင်တော့ copy paste လုပ်မှပဲရပါမယ်။

Python မှာ function ကို def statement နဲ့သတ်မှတ်ပါတယ်။

```
def function-name(Parameter list):
    statements, i.e. the function body
```

Parameter list ကတော့ ဘာမှမပါတာဖြစ်နိုင်လို တစ်ခုထက်ပိုတာလဲဖြစ်နိုင်ပါတယ်။ function call လုပ်ရင် parameter တွေကို argument လို့လည်းခေါ်ပါတယ်။ function body ထဲမှာတော့ indent လုပ်ထားတဲ့ statement တွေပါပါတယ်။ function call လုပ်တိုင်း function body ထဲက statement တွေ execute လုပ်ပါတယ်။ paramameter တွေက mandatory(မပါမဖြစ်) ဖြစ်နိုင်သလို optional(ရွေးချယ်နိုင်) လည်းဖြစ်နိုင်ပါတယ်။ optional parameter တွေက mandatory parameter နောက်မှာရှိရပါမယ်။

Function body မှာ return statement ပါနိုင်ပါတယ်။ function body မည့်သည့် နေရာမှာမဆို ဖြစ်နိုင်ပါတယ်။ return statement ရောက်ရင် function call execution ကိုရပ်ပြီး result (ရလဒ်) ကို return ပြန်ပေးပါတယ်။ return statement မပါရင်တော့ function body အဆုံးရောက်ရင် function အဆုံးသတ်ပါတယ်။

```
>>> def add(x, y):
...     """Return x plus y"""
...     return x + y
...
>>>
```

Function သတ်မှတ်ပြီးပါပြီ။ ပြန်သုံးကြည့်ပါမယ်။

```
>>> add(4,5)
9
>>> add(8,3)
11
>>>
```

### Example of a Function with Optional Parameters

```
>>> def add(x, y=5):
...     """Return x plus y, optional"""
...     return x + y
...
```

```
>>>
```

Function call ပြန်လုပ်မယ်။

```
>>> add(4)
9
>>> add(8,3)
11
>>>
```

Docstring

Function body ရဲ့ ပထမဆုံး statement က string တစ်ကြောင်းထားတတ်ပါတယ်။ အဲ့ဒီ string ကို function_name.__doc_ နဲ့ရယူသုံးစွဲနိုင်ပါတယ်။ Docstring လို့ခေါ်ပါတယ်။

```
>>> execfile("function1.py")
>>> add.__doc__
'Returns x plus y'
>>> add2.__doc__
'Returns x plus y, optional'
>>>
```

Keyword Parameters

Fuction call လုပ်ရာမှာ Keyword parameter တွေအသုံးပြုနိုင်ပါတယ်။

```
def sumsub(a, b, c=0, d=0):
    return a - b + c - d
```

Keyword parameter တွေက positional argument အဖြစ်မသုံးတာတွေပဲဖြစ်နိုင်ပါတယ်။

```
>>> execfile("funktion1.py")
>>> sumsub(12,4)
8
>>> sumsub(12,4,27,23)
12
>>> sumsub(12,4,d=27,c=23)
4
```

Arbitrary Number of Parameters

Parameter အရေအတွက်ဘယ်လောက်ဖြစ်မယ်ဆိုတာ ပြောမရနိုင်တဲ့အခြေအနေဖြစ်နိုင်ပါတယ်။ ဒါကို tuple reference သုံးပြီးဖြေရှင်းနိုင်ပါတယ်။ "*" ကို tuple reference အဖြစ်သတ်မှတ်ရာမှာ သုံးပါတယ်။ C မှာ တော့ "*" ကို pointer အနေနဲ့သုံးပါတယ်။

```
def arbitrary(x, y, *more):
    print "x=", x, ", y=", y
    print "arbitrary: ", more
```

x,y ကတော့ပုံမှန် parameter တွေဖြစ်ပြီးတော့ *more ကတော့ tuple reference ဖြစ်ပါတယ်။

```
>>> execfile("funktion1.py")
>>> arbitrary(3,4)
x= 3 , x= 4
arbitrary:  ()
>>> arbitrary(3,4, "Hello World", 3 ,4)
x= 3 , x= 4
arbitrary:  ('Hello World', 3, 4)
```

# Chapter 20

Recursive Functions

Definition

Recursive ဆိုတာကတော့ latin verb "recurrere"ကနေဆင်းသက်လာပြီးတော့ "ပြန်လာတယ်" ဆိုတဲ့ အဓိပ္ပာယ်ရှိပါတယ်။ recursive function ဆိုတာလည်းဒီသဘောပါပဲ function တစ်ခုက မိမိကိုယ်တိုင် ကို function call ပြန်လုပ်ရင် recursive ဖြစ်တယ်ပြောရပါမယ်။

```
n! = n * (n-1)!, if n > 1 and f(1) = 1
```

Definition of Recursion

Function တစ်ခုက မိမိကိုယ်ကို function call ပြန်လုပ်မယ်။ function call လုပ်တဲ့ value ကို return ပြန်ပေးပါတယ်။

Termination condition:

Recursive သုံးရင် အဆုံးသတ်ပေးဖို့(terminate) လုပ်ပေးဖို့လိုပါတယ်။ သတ်မှတ်ထားတဲ့ အခြေအနေတစ်ခုမှာ recursive function အဆုံးသတ်ပါတယ် base case လို့ခေါ်တယ်။ function call လုပ်တဲ့အခါ base case မှာ သတ်မှတ်ထားတဲ့ အခြေအနေမရောက်နိုင်ဘူးဆိုရင် တော့ infinite loop ဖြစ်သွားပါလိမ့်မယ်။

Example:

```
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1
```

**အစားထိုး**

```
4! = 4 * 3 * 2 * 1
```

ယေဘူယျပြောရင် recursive ဆိုတာက ပြဿနာတစ်ခုကို အပိုင်းလေးတွေခွဲဖြေရှင်းတာဖြစ်ပါတယ်။

Recursive Functions in Python

Python မှာ implementation လုပ်မယ်ဆိုရင်တော့

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

ဘယ်လိုအလုပ်လုပ်တယ်သိရအောင် Print function လေးတွေထည့်ကြည့်မယ်။

```
def factorial(n):
    print("factorial has been called with n = " + str(n))
    if n == 1:
        return 1
    else:
        res = n * factorial(n-1)
        print("intermediate result for ", n, " * factorial(" ,n-1, "):
",res)
        return res

print(factorial(5))
```

result က

```
factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1
intermediate result for  2  * factorial( 1 ):  2
intermediate result for  3  * factorial( 2 ):  6
intermediate result for  4  * factorial( 3 ):  24
intermediate result for  5  * factorial( 4 ):  120
120
```

Factorial ကို iterative နဲ့ရေးကြည့်မယ်။

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

The Pitfalls of Recursion

Fibonacci numbersတွေကတော့

```
0,1,1,2,3,5,8,13,21,34,55,89, .
```

သူတို့ကိုသတ်မှတ်တာကတော့

$F_n = F_{n-1} + F_{n-2}$
with $F_0 = 0$ and $F_1 = 1$

Fibonacci ကို fibonacci လို့လူသိများတဲ့ သချာ်ပညာရင် Leonardo of Pisa နာမည်ပေးထားတာပါ။ သူ့ရဲ့ "Liber Abaci" (publishes 1202) စာအုပ်ထဲမှာ ယုန်

တွေနဲ့ပတ်သတ်ပြီး လေ့ကျင့်ခန်း အနေနဲ့ ကိန်းစဉ်တန်းတစ်ခုဖော်ပြခဲ့ပါတယ်။ modern mathematic sequences တွေက $F_0=0$ ကစပေမဲ့ သူ့ရဲ့ Fibonacci sequence မှာတော့ $F_1=1$ ကနေစပါတယ်။

ယုန်ပေါက်ဖွားမှုပြဿနာလေးအတွက် အခြေအနေက

- ယုန်ထီးတစ်ကောင် ယုန်မတစ်ကောင်ကနေစမယ်
- အသက်တစ်လ မှာမိတ်လိုက်မယ် ဒုတိယလမှာ ယုန်နှောပ်တစ်စုံ(ထီး/မ)ထပ်တိုးမယ်။
- ယုန်တွေကမသေဘူး
- ယုန်တစ်စုံတိုင်း(ထီး/မ) တစ်စုံတိုင်းက ဒုတိယလ ကနေစပြီးတော့ လတိုင်း တစ်စုံ ထပ်ပေါက်မယ်။

Fibonacci number ကတော့ n month ပြီးရင်ရှိတဲ့ ယုန် အစုံအရေအတွက်ပါဖြစ်ပါတယ်။ ၁၀ လဆိုရင် $F_{10}$ဖြစ်ပါတယ်။

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

iterative solution

```
def fibi(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

fibi() က fib() ထက်ပိုမြန်ပါတယ်။ ဘယ်လောက်မြန်လဲသိချင်ရင် timeit module သုံးရပါမယ်။

```
from timeit import Timer
from fibo import fib

t1 = Timer("fib(10)","from fibo import fib")

for i in range(1,41):
        s = "fib(" + str(i) + ")"
        t1 = Timer(s,"from fibo import fib")
        time1 = t1.timeit(3)
        s = "fibi(" + str(i) + ")"
        t2 = Timer(s,"from fibo import fibi")
        time2 = t2.timeit(3)
        print("n=%2d, fib: %8.6f, fibi:  %7.6f, percent: %10.2f" % (i,
time1, time2, time1/time2))
```

time1 ကတော့ fib() အတွက် time2 ကတော့ fibi() အတွက်ဖြစ်ပါတယ်။ result ကိုကြည့်ရင် fib(20) ကို ၃ ခါခေါ်ရင် 14 milliseconds ကြာတယ်။ fibi(20) ကတော့ ၃ ကြိမ်ခေါ်ရာမှာ 0.011 milliseconds ပဲလိုပါတယ်။ ဒါကြောင့် fibi(20) က fib(20) ထက် အဆ ၁၃၀၀ မြန်ပါတယ်။

fib(40) ကသုံးကြိမ်အတွက် 215 second ကြာပါတယ်။ fibi(40)ကတော့ 0.016 milisecond ပဲလိုပါတယ်။ fibi(40) က fib(40)ထက် 13 millions ဆမြန်ပါတယ်။

```
n= 1, fib: 0.000004, fibi:  0.000005, percent:       0.81
n= 2, fib: 0.000005, fibi:  0.000005, percent:       1.00
n= 3, fib: 0.000006, fibi:  0.000006, percent:       1.00
n= 4, fib: 0.000008, fibi:  0.000005, percent:       1.62
n= 5, fib: 0.000013, fibi:  0.000006, percent:       2.20
n= 6, fib: 0.000020, fibi:  0.000006, percent:       3.36
n= 7, fib: 0.000030, fibi:  0.000006, percent:       5.04
n= 8, fib: 0.000047, fibi:  0.000008, percent:       5.79
n= 9, fib: 0.000075, fibi:  0.000007, percent:      10.50
n=10, fib: 0.000118, fibi:  0.000007, percent:      16.50
n=11, fib: 0.000198, fibi:  0.000007, percent:      27.70
n=12, fib: 0.000287, fibi:  0.000007, percent:      41.52
n=13, fib: 0.000480, fibi:  0.000007, percent:      69.45
n=14, fib: 0.000780, fibi:  0.000007, percent:     112.83
n=15, fib: 0.001279, fibi:  0.000008, percent:     162.55
n=16, fib: 0.002059, fibi:  0.000009, percent:     233.41
n=17, fib: 0.003439, fibi:  0.000011, percent:     313.59
n=18, fib: 0.005794, fibi:  0.000012, percent:     486.04
n=19, fib: 0.009219, fibi:  0.000011, percent:     840.59
n=20, fib: 0.014366, fibi:  0.000011, percent:    1309.89
n=21, fib: 0.023137, fibi:  0.000013, percent:    1764.42
n=22, fib: 0.036963, fibi:  0.000013, percent:    2818.80
n=23, fib: 0.060626, fibi:  0.000012, percent:    4985.96
n=24, fib: 0.097643, fibi:  0.000013, percent:    7584.17
n=25, fib: 0.157224, fibi:  0.000013, percent:   11989.91
n=26, fib: 0.253764, fibi:  0.000013, percent:   19352.05
n=27, fib: 0.411353, fibi:  0.000012, percent:   34506.80
n=28, fib: 0.673918, fibi:  0.000014, percent:   47908.76
n=29, fib: 1.086484, fibi:  0.000015, percent:   72334.03
n=30, fib: 1.742688, fibi:  0.000014, percent:  123887.51
n=31, fib: 2.861763, fibi:  0.000014, percent:  203442.44
n=32, fib: 4.648224, fibi:  0.000015, percent:  309461.33
n=33, fib: 7.339578, fibi:  0.000014, percent:  521769.86
n=34, fib: 11.980462, fibi:  0.000014, percent:  851689.83
n=35, fib: 19.426206, fibi:  0.000016, percent: 1216110.64
n=36, fib: 30.840097, fibi:  0.000015, percent: 2053218.13
n=37, fib: 50.519086, fibi:  0.000016, percent: 3116064.78
n=38, fib: 81.822418, fibi:  0.000015, percent: 5447430.08
n=39, fib: 132.030006, fibi:  0.000018, percent: 7383653.09
n=40, fib: 215.091484, fibi:  0.000016, percent: 13465060.78
```

recursion မှာဘာတွေမှားနေလဲ binary tree ဆွဲကြည့်ရင် subtree တွေကိုထပ်ခါထပ်ခါတွက်ထားတာကိုတွေ့မှာဖြစ်ပါတယ်။ recursion မှာတွက်ပြီးသားတွေကိုမှတ်မထားတဲ့အတွက် ဖြစ်ပါတယ်။

တွက်ပြီးသားတွေမှတ်မိနိုင်အောင် "memory" ကို dictionary သုံးပြီး implement လုပ်နိုင်ပါတယ်။

```
memo = {0:0, 1:1}
def fibm(n):
```

```
    if not n in memo:
        memo[n] = fibm(n-1) + fibm(n-2)
    return memo[n]
```

## fibi():ယှဉ်ကြည့်ရအောင်

```
        from timeit import Timer
from fibo import fib

t1 = Timer("fib(10)","from fibo import fib")

for i in range(1,41):
        s = "fibm(" + str(i) + ")"
        t1 = Timer(s,"from fibo import fibm")
        time1 = t1.timeit(3)
        s = "fibi(" + str(i) + ")"
        t2 = Timer(s,"from fibo import fibi")
        time2 = t2.timeit(3)
        print("n=%2d, fib: %8.6f, fibi:  %7.6f, percent: %10.2f" % (i,
time1, time2, time1/time2))
```

## iterative ထက်တောင်ပိုမြန်တာတွေ့ပါလိမ့်မယ်။

```
n= 1, fib: 0.000011, fibi:  0.000015, percent:       0.73
n= 2, fib: 0.000011, fibi:  0.000013, percent:       0.85
n= 3, fib: 0.000012, fibi:  0.000014, percent:       0.86
n= 4, fib: 0.000012, fibi:  0.000015, percent:       0.79
n= 5, fib: 0.000012, fibi:  0.000016, percent:       0.75
n= 6, fib: 0.000011, fibi:  0.000017, percent:       0.65
n= 7, fib: 0.000012, fibi:  0.000017, percent:       0.72
n= 8, fib: 0.000011, fibi:  0.000018, percent:       0.61
n= 9, fib: 0.000011, fibi:  0.000018, percent:       0.61
n=10, fib: 0.000010, fibi:  0.000020, percent:       0.50
n=11, fib: 0.000011, fibi:  0.000020, percent:       0.55
n=12, fib: 0.000004, fibi:  0.000007, percent:       0.59
n=13, fib: 0.000004, fibi:  0.000007, percent:       0.57
n=14, fib: 0.000004, fibi:  0.000008, percent:       0.52
n=15, fib: 0.000004, fibi:  0.000008, percent:       0.50
n=16, fib: 0.000003, fibi:  0.000008, percent:       0.39
n=17, fib: 0.000004, fibi:  0.000009, percent:       0.45
n=18, fib: 0.000004, fibi:  0.000009, percent:       0.45
n=19, fib: 0.000004, fibi:  0.000009, percent:       0.45
n=20, fib: 0.000003, fibi:  0.000010, percent:       0.29
n=21, fib: 0.000004, fibi:  0.000009, percent:       0.45
n=22, fib: 0.000004, fibi:  0.000010, percent:       0.40
n=23, fib: 0.000004, fibi:  0.000010, percent:       0.40
n=24, fib: 0.000004, fibi:  0.000011, percent:       0.35
n=25, fib: 0.000004, fibi:  0.000012, percent:       0.33
n=26, fib: 0.000004, fibi:  0.000011, percent:       0.34
n=27, fib: 0.000004, fibi:  0.000011, percent:       0.35
n=28, fib: 0.000004, fibi:  0.000012, percent:       0.32
n=29, fib: 0.000004, fibi:  0.000012, percent:       0.33
n=30, fib: 0.000004, fibi:  0.000013, percent:       0.31
n=31, fib: 0.000004, fibi:  0.000012, percent:       0.34
n=32, fib: 0.000004, fibi:  0.000012, percent:       0.33
n=33, fib: 0.000004, fibi:  0.000013, percent:       0.30
n=34, fib: 0.000004, fibi:  0.000012, percent:       0.34
n=35, fib: 0.000004, fibi:  0.000013, percent:       0.31
n=36, fib: 0.000004, fibi:  0.000013, percent:       0.31
n=37, fib: 0.000004, fibi:  0.000014, percent:       0.29
```

```
n=38, fib: 0.000004, fibi:  0.000014, percent:       0.29
n=39, fib: 0.000004, fibi:  0.000013, percent:       0.31
n=40, fib: 0.000004, fibi:  0.000014, percent:       0.29
```

## Exercises

1. Think of a recusive version of the function f(n) = 3 * n, i.e. the multiples of 3
2. Write a recursive Python function that returns the sum of the first `n` integers.
   (Hint: The function will be similiar to the factorial function!)
3. Write a function which implements the Pascal's triangle:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5   10  10  5   1
```

4. You find further exercises on our Python3 version of recursive functions, e.g. creating the Fibonacci numbers out of Pascal's triangle or produce the prime numbers recursively, using the Sieve of Eratosthenes.

## Solutions to our Exercises

1. Solution to our first exercise on recursion:
   Mathematically, we can write it like this:
   `f(1) = 3,`
   `f(n+1) = f(n) + 3`

   A Python function can be written like this:

```
2.  def mult3(n):
3.      if n == 1:
4.          return 3
5.      else:
6.          return mult3(n-1) + 3
7.
8.  for i in range(1,10):
9.      print(mult3(i))
```

10. Solution to our second exercise:

```
11. def sum_n(n):
12.     if n== 0:
13.         return 0
14.     else:
15.         return n + sum_n(n-1)
```

16. Solution for creating the Pacal triangle:

```
17. def pascal(n):
18.     if n == 1:
19.         return [1]
20.     else:
21.         line = [1]
22.         previous_line = pascal(n-1)
23.         for i in range(len(previous_line)-1):
```

```
24.              line.append(previous_line[i] + previous_line[i+1])
25.          line += [1]
26.      return line
27.
28.  print(pascal(6))
```

Alternatively, we can write a function using list comprehension:

```
def pascal(n):
    if n == 1:
        return [1]
    else:
        p_line = pascal(n-1)
        line = [ p_line[i]+p_line[i+1] for i in range(len(p_line)-1)]
        line.insert(0,1)
        line.append(1)
    return line

print(pascal(6))
```

# Chapter 21

Python Tests

## Errors and Tests

Debugging and testing ဆိုတာလည်း programmer များအချိန်များများပေးရတဲ့ အဆင့်ပါပဲ။ အမျိုးမျိုးသောအကြောင်း အရာတွေပေါ်မူတည်တဲ့အတွက် လည်း ဘယ်လောက်အချိန်ပေးရမယ့် ဆိုတာပြောရခက်ပါတယ်။

## Kinds of Errors

Error အမျိုးမျိုးရှိပါတယ်။ typo error မျိုး စကားလုံးတွေမှားတာ ကျန်တာ တွေကို syntactical error လို့ခေါ်ပါတယ်။

Syntactical error တွေကပြင်ရလွယ်ပါတယ်။ Semantic error ကတော့ စကားလုံးတွေ codeတွေက syntactically မှန်ပေမယ့် ကိုယ်လိုချင်သလိုအလုပ်မလုပ်တာတွေ ဖြစ်တာကြောင့် ရှာဖွေရခက်ပါတယ်။ ဥပမာ x ကို 1 တိုးချင်တဲ့အခါ x+=1 လို့ရေးရမယ့်အစား x=1 လို့ရေးမိတာမျိုး ။example

```
x = int(input("x? "))
y = int(input("y? "))

if x > 10:
    if y == x:
        print("Fine")
else:
    print("So what?")
```

တစ်ခုနဲ့တစ်ခု ထပ်ထားတဲ့ (nested) if statement နှစ်ကြောင်းတွေ့ပါလိမ့်မယ်။ syntactically အရတော့မှန်ပါတယ်။ ဒါပေမယ့် ရေးတဲ့သူက "so what"ဆိုတာပဲ ထုတ်ချင်တယ်ဆိုမှပါ။ x က 10 ထက်လည်းကြီးမယ့် y နဲ့လည်းမတူမှ "so what"ကိုထုတ်ချင်ရင်တော့ အောက်ပါပုံစံအတိုင်း ဖြစ်ရ ဖြစ်ရပါမယ်။

```
x = int(input("x? "))
y = int(input("y? "))

if x > 10:
    if y == x:
        print("Fine")
    else:
        print("So what?")
```

နှစ်ခုစလုံးက syntactically မှန်ပါတယ်။ ဒါပေမယ့် တစ်ခုက လိုချင်သလို (semantic)အလုပ်မလုပ်ပါဘူး။

နောက်တစ်ခုကြည့်ရအောင်

```
>>> for i in range(7):
...     print(i)
...
0
1
2
3
4
5
6
>>>
```

Syntactically မှန်တဲ့အတွက် Error မတက်ပဲ run ပါလိမ့်မယ်။ ဒါပေမယ့် programmer ကဘာကို output ထုတ်ပေးချင်တာလဲ (semantic) ကိုမသိနိုင်ပါဘူး။ 1 to 7 ထုပ်ချင်တာလည်းဖြစ်နိုင် ပါတယ်။ ဒီမှာတော့ ရေးတဲ့ သူက range function ကိုသေချာမသိပါဘူး။

ဒါကြောင့် semantic error ကိုနှစ်မျိုးခွဲခြားနိုင်ပါတယ်။

- language ရေးသားတဲ့ပုံစံကိုနားမလည်တာ
- Code တွေရေးတဲ့အခါ logically မှားတာ

Unit Tests

Code တွေထဲက တစ်စိတ်တစ်ပိုင်းတွေဖြစ်တဲ့ class တွေ function တွေ စမ်းရင်သုံးပါတယ်။ သဘောကတော့ system ကြီးကို unit လေးတွေအဖြစ် ပိုင်းပြီး စစ်ဆေးတာဖြစ်ပါတယ်။ ဒီလို စမ်းသပ်ဖို့အတွက် လည်း program တွေကို စမ်းသပ်လို့ရတဲ့ unit တွေ ခွဲခြားရပါတယ်။

Module Tests with __name__

Module တိုင်းမှာ နာမည် ရှိပါတယ်။ built in attribute တစ်ခုဖြစ်တဲ့ __name__ နဲ့သတ်မှတ် ထားပါတယ်။ xyz ဆိုတဲ့ module ရေးပြီး xyz.py လို့သိမ်းထားတယ်ဆိုပါစို့။ "import xyz" နဲ့ import လုပ်တဲ့အခါ __name__ ထဲကို xyz ဆိုတဲ့ string ရောက်သွားပါတယ်။ xyz.py ကို standalone သုံးမယ်ဆိုရင်တော့

```
$python xyz.py
```

__name__ ထဲမှာရှိမမယ့်တန်ဖိုး က '__main__' ဆိုတဲ့ string တန်ဖိုးဖြစ်မှာပါ။

အောက်ပါ Module က Fibonacci တန်ဖိုးတွက်ဖို့ပါ။ module ဘာလုပ်တယ်ဆိုတာ အရေးကြီးတယ် ပြောချင်တာမဟုတ်ပါဘူး။ modul ထဲမှာ module test ဘယ်လိုလုပ်တယ် ဆိုတာ demonstration လုပ်ပြချင်လို့ပါ။ module ကို standalone သုံးတာလား import လုပ်တာလား ဆုံးဖြတ်ချင်ရင် _name_ ထဲကတန်ဖိုးကိုစစ်ရပါတယ်။

အောက်ဖော်ပြပါ code ကို "fibonacci.py" နဲ့သိမ်းပါမယ်။

```
def fib(n):
    """ Calculates the n-th Fibonacci number iteratively """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

def fiblist(n):
    """ creates a list of Fibonacci numbers up to the n-th generation """
    fib = [0,1]
    for i in range(1,n):
        fib += [fib[-1]+fib[-2]]
    return fib
```

Python shell မှာ ဒီModule ကိုစမ်းချင်ရင်လည်းရပါတယ်။

```
>>> from fibonacci import fib, fiblist
>>> fib(0)
0
>>> fib(1)
1
>>> fib(10)
55
>>> fiblist(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> fiblist(-8)
[0, 1]
>>> fib(-1)
0
>>> fib(0.5)
Traceback (most recent call last):
  File "", line 1, in
  File "fibonacci.py", line 6, in fib
    for i in range(n):
TypeError: 'float' object cannot be interpreted as an integer
>>>
```

Positive integer တွေပဲဆိုရင်တော့ မှန်နေတာ တွေ့ပါလိမ့်မယ်။ negative input ဆို fib function က 0 return ပြန်ပြီး fiblist က [0,1] ပဲ return ပြန်ပါလိမ့်မယ်။ float ထည့်ပေးရင်တော့ နှစ်ခုလုံးက error ပြပါလိမ့်မယ်။

Module test လုပ်ဖို့အတွက် fib() နဲ့ fiblist() ကို နံပါတ်တွေနဲ့return တွေအသုံးပြုစစ်ဆေးနိုင်ပါတယ်။

```
if fib(0) == 0 and fib(10) == 55 and fib(50) == 12586269025:
    print("Test for the fib function was successful!")
else:
    print("The fib function is returning wrong values!")
```

standalone သုံးရင်အောက်ပါအတိုင်းမြင်ရပါလိမ့်မယ်။

```
python fibonacci.py
Test for the fib function was successful!
```

Error တွေထည့်ကြည့်ရအောင်

```
a, b = 0, 1
```

ကို

```
a, b = 1, 1
```

fib function ကတွက်တော့တွက်ပေးနေတုန်းပါပဲ ။ဒါပေမယ့် "n+1"အတွက် တွက်ပေးနေတာဖြစ်တဲ့ အတွက်ကြောင့် call လုပ်ရင်အောက်ပါအတိုင်းရပါလိမ့်မယ်။

```
python fibonacci.py
"The fib function is returning wrong values!
```

ဒီလိုရေးထားတာမကောင်းတာတစ်ခုရှိပါတယ်။ module import လုပ်ရင် test လုပ်တာအဆင်ပြေကြောင်း output အမြဲတွေ့ပါလိမ့်မယ်။ ဒါပေမယ့် ကျွန်တော်တို့ က module import လုပ်ရင် ဒါကိုမမြင်ချင်ပါဘူး။

```
>>> import fibonacci
Test for the fib function was successful!
```

အနောက်အယှက်လည်းဖြစ်သလို။module import လုပ်ရင် silent ဖြစ်နေသင့်ပါတယ်။

ဒါကိုဖြေရှင်းချင်ရင်တော့ _name_ ကိုစစ်ရပါမယ်။

```
def fib(n):
    """ Calculates the n-th Fibonacci number iteratively """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

def fiblist(n):
    """ creates a list of Fibonacci numbers up to the n-th generation """
    fib = [0,1]
    for i in range(1,n):
        fib += [fib[-1]+fib[-2]]
    return fib

if __name__ == "__main__":
    if fib(0) == 0 and fib(10) == 55 and fib(50) == 12586269025:
        print("Test for the fib function was successful!")
    else:
        print("The fib function is returning wrong values!")
```

ဒါဆို module import လုပ်ရင်မြင်ရတော့မှာမဟုတ်ပါဘူး။အရှင်းဆုံး module test ပါပဲ အကောင်း ဆုံးတော့မဟုတ်ပါဘူး။

## doctest Module

unittest ထက်စာရင် doctest ကိုပိုအသုံးများကြပါတယ်။ python နဲ့တွဲဖက်ပါဝင်လာတဲ့ test framework ဖြစ်ပါတယ်။ doctest ကmodule ရဲ့ documentation ထဲမှာ interactive python session နဲ့တူတဲ့ အစိတ်အပိုင်းတွေရှာပြီးတော့ run တာတွေတူလားစစ်ပါလိမ့်မယ်။ module document ထဲမှာ sample run ထားတာလေးတွေကို doctest ကပြန်စမ်းကြည့် ပြီး အဖြေတူမတူ စစ်ဆေးတာဖြစ်ပါတယ်။
doctest ကိုသုံးဖို့ import လုပ်ပေးရပါမယ်။ သက်ဆိုင်ရာ function ရဲ့ docstrin ထဲမှာ စမ်းသပ်တဲ့ python interactive session တွေထည့်ပေးရပါမယ်။example

```
import doctest

def fib(n):
    """ Calculates the n-th Fibonacci number iteratively """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

import လုပ်ပြီးစမ်းကြည့်ပါမယ်။

```
>>> from fibonacci import fib
>>> fib(0)
0
>>> fib(1)
1
>>> fib(10)
55
>>> fib(15)
610
>>>
```

စမ်းထားတဲ့ တစ်ခုလုံးကို function ရဲ့ docstring ထဲမှာထည့်ပါမယ်။ test လုပ်တဲ့အခါမှာ module ကို srandalone သုံးရင်တော့ testmod() ကို ခေါ်ပေးရပါမယ်။

```
import doctest

def fib(n):
    """
    Calculates the n-th Fibonacci number iteratively

    >>> fib(0)
    0
    >>> fib(1)
    1
    >>> fib(10)
    55
    >>> fib(15)
    610
    >>>

    """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a

if __name__ == "__main__":
    doctest.testmod()
```

```
python fibonacci_doctest.py
```
လို့စမ်းသပ်ကြည့်ရင် error မရှိတဲ့အတွက် output ဘာမှရမှာမဟုတ်ပဘူး။

မှားတဲ့အခါဘယ်လို error ပြမလဲသိရအောင်

```
a, b = 0, 1
```
ကို

```
a, b = 1, 1
```
ပြောင်းကြည့်ရအောင်

Module ကို run ကြည့်ရင်အောက်ပါအတိုင်းတွေ့ရပါလိမ့်မယ်။

```
$ python fibonacci_doctest.py
**********************************************************************
File "fibonacci_doctest.py", line 8, in __main__.fib
Failed example:
    fib(0)
Expected:
    0
Got:
    1
**********************************************************************
File "fibonacci_doctest.py", line 12, in __main__.fib
Failed example:
    fib(10)
Expected:
    55
Got:
    89
**********************************************************************
File "fibonacci_doctest.py", line 14, in __main__.fib
Failed example:
    fib(15)
Expected:
    610
Got:
    987
**********************************************************************
1 items had failures:
   3 of   4 in __main__.fib
***Test Failed*** 3 failures.
```
Error ပါတဲ့ call အားလုံးကိုဖော်ပြပေးတာတွေ့ပါလိမ့်မယ်။ "Failed example:" ဆိုတဲ့ line နောက်မှာ argument နဲ့ call ထားတာတွေတွေ့ပါလိမ့်မယ်။ "Expected:"နောက်မှာတော့ ဖြစ်ရမယ့် value တွေ့ပါပါတယ်။ "Got:" နောက်မှာတော့ တကယ်ရလာတဲ့ တန်ဖိုးဖြစ်ပါတယ်။

Test-driven Development (TDD)

ရေးပြီးသား function တွေကိုတော့စမ်းသပ်ပြီးပါပြီ။ မရေးရသေးတဲ့ implement မလုပ်ရသေးတဲ့ function တွေဆိုရင်ရော? မဖြစ်နိုင်ဘူးထင်ပါသလား test-drive development ကဒီအတွက်သုံးပါတယ်။

ဒီနည်းလမ်းအတွက် ခက်ခဲတာက သင်လျှော်တဲ့ test case တွေဖြစ်ပါတယ်။ အကောင်းဆုံး test case ကတော့ ဖြစ်နိုင်တဲ့ input အားလုံးနဲ့output တွေကို စစ်ဆေးပါလိမ့်မယ်။ လက်တွေ့မှာတော့ မဖြစ်နိုင်ပါဘူး။

အောက်ဖော်ပြပါ ဥပမာ မှာ return value ကို 0 သတ်မှတ်ပေးထားပါတယ်။

```
import doctest

def fib(n):
    """
    Calculates the n-th Fibonacci number iteratively

    >>> fib(0)
    0
    >>> fib(1)
    1
    >>> fib(10)
    55
    >>> fib(15)
    610
    >>>

    """

    return 0

if __name__ == "__main__":
    doctest.testmod()
```

ဒါကြောင့် fib(0) အတွက်ကလွဲပြီးကျန်တာတွေကို error တွေပေးပါလိမ့်မယ်။

```
$ python3 fibonacci_TDD.py
**********************************************************************
File "fibonacci_TDD.py", line 10, in __main__.fib
Failed example:
    fib(1)
Expected:
    1
Got:
    0
**********************************************************************
File "fibonacci_TDD.py", line 12, in __main__.fib
Failed example:
    fib(10)
Expected:
    55
Got:
    0
**********************************************************************
File "fibonacci_TDD.py", line 14, in __main__.fib
Failed example:
```

```
    fib(15)
Expected:
    610
Got:
    0
**********************************************************************
1 items had failures:
   3 of   4 in __main__.fib
***Test Failed*** 3 failures.
```

ဒီcodeကို test ကို pass မဖြစ်မချင်း ပြောင်းလဲရေးသားပေးရမှာဖြစ်ပါတယ်။

ဒါကြောင့် ဒါကို test driven software development လို့ခေါ်ကြပါတယ်။

Unittest

Python module တစ်ခုဖြစ်တဲ့ unittest ကတော့ unit testing framework တစ်ခုဖြစ်ပြီး Erich Gamma's JUnit and Kent Beck's Smalltalk testing framework ပေါ်မှာ အခြေခံတည်ဆောက် ထားပါတယ်။ ဒီmodule ထဲမှာ test case and suits တွေ ၊ test တွေ လုပ်ဖို့ text-based utility class တွေ result တွေ ၊report လုပ်ဖို့တွေ(TextTestRunner) တွေပါဂင်ပါတယ်။

"doctest" နဲ့မတူတာတစ်ခုကတော့ စမ်းသပ်မည့် Module ထဲမှာထည့်ဖို့မလိုပါဘူး။ အဓိက အကျိုးကျေးဇူးကတော့ program ရဲ့ description နဲ့ test description ကိုခွဲခြားလိုက်တာပဲဖြစ်ပါတယ်။ test case တွေတော့ ပိုလုပ်ပေးရပါမယ်။

Unittest မှာလိုပဲ fobonacci module ကိုပဲသုံးပြီးစမ်းပါမယ်။ fibonacci_unittest.py လုပ်မယ်။ ဒီထဲမှာ unittest တွေစမ်းမည့် module တွေ import လုပ်ဖို့လိုပါတယ်။

ပြီးတော့ unittest.TestCase ကို inherit လုပ်ထားတဲ့ class တစ်ခုလိုမယ် "FibonacciTest" လို့ခေါ်ပါမယ်။ test case တွေကို ဒီ class ထဲမှာ method တွေသုံးပြီးသတ်မှတ်ပါမယ်။ method name တွေကကြိုက်တာဖြစ်လို့ရပေမယ့် test နဲ့စရပါမယ်။ "testCalculation" ထဲမှာ TestCase. assertEqual class ထဲက assertEqual method ကိုသုံးထားပါတယ်။ "first" expression နဲ့ "second" expression တူလားစစ်တာပါ။ မတူရင် msg ကို output ထုတ်ပေးပါလိမယ့်မယ်။

```
import unittest
from fibonacci import fib

class FibonacciTest(unittest.TestCase):

    def testCalculation(self):
        self.assertEqual(fib(0), 0)
        self.assertEqual(fib(1), 1)
        self.assertEqual(fib(5), 5)
        self.assertEqual(fib(10), 55)
        self.assertEqual(fib(20), 6765)

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

ဒီ test case ကိုစမ်းရင်အောက်ပါအတိုင်းရပါလိမ့်မယ်။

```
$ python3 fibonacci_unittest.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

လိုချင်တဲ့ result ပါပဲ ။ error တွေစမ်းကြည့်ပါမယ်။

```
a, b = 0, 1
```

ကိုပဲပြောင်းကြည့်ပါဦးမယ်။

```
a, b = 1, 1
```

ဒါဆိုရင်တော့

```
$ python3 fibonacci_unittest.py
F
======================================================================
FAIL: testCalculation (__main__.FibonacciTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "fibonacci_unittest.py", line 7, in testCalculation
    self.assertEqual(fib(0), 0)
AssertionError: 1 != 0

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

testCalculation ရဲ့ ပထမဆုံးအကြောင်းမှာ error ပြပါတယ်။ တစ်ခြား assertEqual တွေက execute မလုပ်သေးပါဘူး။ error ကိုပြင်ပြီ နောက်တစ်ခုလုပ်ကြည့်ရအောင်။ ဒီတစ်ခါကျ 20 အတွက်ကလွဲပြီး ကျန်တာ မှန်ပါလိမ့်မယ်။

```
def fib(n):
    """ Iterative Fibonacci Function """
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    if n == 20:
        a = 42
    return a
```

ဒီတစ်ခါမှာတော့

```
$ python3 fibonacci_unittest.py
blabal
F
======================================================================
FAIL: testCalculation (__main__.FibonacciTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "fibonacci_unittest.py", line 12, in testCalculation
    self.assertEqual(fib(20), 6765)
AssertionError: 42 != 6765

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

testCalculation ရဲ့statement တွေအားလုံး execute လုပ်ပေမယ့် အားလုံးကို အဆင်ပြေနေတဲ့အတွက် output တွေ့ရမှာ မဟုတ်ပါဘူး။

```
self.assertEqual(fib(0), 0)
self.assertEqual(fib(1), 1)
self.assertEqual(fib(5), 5)
```

Methods of the Class TestCase

| Method | Meaning |
|---|---|
| `setUp()` | Hook method for setting up the test fixture before exercising it. This method is called before calling the implemented test methods. |
| `tearDown()` | Hook method for deconstructing the class fixture after running all tests in the class. |
| `assertEqual(self, first, second, msg=None)` | The test fails if the two objects are not equal as determined by the '==' operator. |
| `assertAlmostEqual(self, first, second, places=None, msg=None, delta=None)` | The test fails if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta.<br>Note that decimal places (from zero) are usually not the same as significant digits (measured from the most significant digit).<br>If the two objects compare equal then they will automatically compare almost equal. |
| `assertCountEqual(self, first, second, msg=None)` | An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.<br>`self.assertEqual(Counter(list(first)), Counter(list(second))`<br>Example:<br>`[0, 1, 1]` and `[1, 0, 1]` compare equal, because the number of ones and zeroes are the same.<br>`[0, 0, 1]` and `[0, 1]` compare unequal, because zero appears twice in the first list and only once in the second list. |
| `assertDictEqual(self, d1, d2, msg=None)` | Both arguments are taken as dictionaries and they are checked if they are equal. |
| `assertTrue(self, expr, msg=None)` | Checks if the expression "expr" is True. |
| `assertGreater(self, a, b,` | Checks, if a > b is True. |

| Method | Meaning |
|---|---|
| msg=None) | |
| assertGreaterEqual(self, a, b, msg=None) | Checks if a ≥ b |
| assertFalse(self, expr, msg=None) | Checks if expression "expr" is False. |
| assertLess(self, a, b, msg=None) | Checks if a < b |
| assertLessEqual(self, a, b, msg=None) | Checks if a ≤ b |
| assertIn(self, member, container, msg=None) | Checks if a in b |
| assertIs(self, expr1, expr2, msg=None) | Checks if "a is b" |
| assertIsInstance(self, obj, cls, msg=None) | Checks if isinstance(obj, cls). |
| assertIsNone(self, obj, msg=None) | Checks if "obj is None" |
| assertIsNot(self, expr1, expr2, msg=None) | Checks if "a is not b" |
| assertIsNotNone(self, obj, msg=None) | Checks if obj is not equal to None |
| assertListEqual(self, list1, list2, msg=None) | Lists are checked for equality. |
| assertMultiLineEqual(self, first, second, msg=None) | Assert that two multi-line strings are equal.q |
| assertNotRegexpMatches(self, text, unexpected_regexp, msg=None) | Fails, if the text Text "text" of the regular expression unexpected_regexp matches. |
| assertTupleEqual(self, tuple1, tuple2, msg=None) | Analogous to assertListEqual |

setUp နဲ့ a tearDown methodတွေထပ်ထည့်မယ်။

```
import unittest
from fibonacci import fib

class FibonacciTest(unittest.TestCase):

    def setUp(self):
        self.fib_elems = ( (0,0), (1,1), (2,1), (3,2), (4,3), (5,5) )
        print ("setUp executed!")

    def testCalculation(self):
        for (i,val) in self.fib_elems:
            self.assertEqual(fib(i), val)

    def tearDown(self):
        self.fib_elems = None
        print ("tearDown executed!")

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

result ကတော့

```
$ python3 fibonacci_unittest2.py
setUp executed!
tearDown executed!
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

TestCase methods တွေတော်တော်များများမှာ optional parameter "msg"ပါပါတယ်။ msg နဲ့ error အတွက်အကြောင်းအရာတွေထပ်ထည့်ပေးလို့ရပါတယ်။

Exercises

1. Exercise:

Can you find a problem in the following code?

```
import doctest

def fib(n):
    """ Calculates the n-th Fibonacci number iteratively

    >>> fib(0)
    0
    >>> fib(1)
    1
    >>> fib(10)
    55
    >>> fib(40)
    102334155
    >>>

    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

if __name__ == "__main__":
    doctest.testmod()
```

Answer:

The doctest is okay. The problem is the implementation of the fibonacci function. This recursive approach is "highly" inefficient. You need a lot of patience to wait for the termination of the test. The number of hours, days or weeks depend on your computer.☺

---

## Chapter 22

Memoization with Decorators

Memoization ဆိုတာကတော့ computing မှာ programတွေမြန်ဆန်အောင်လုပ်တဲ့ နည်းတစ်ခုပါ။ funcation call result လိုမျိုး တွက်ချက်မှုရလဒ်တွေကို သိမ်းထားတာဖြစ်ပါတယ်။ တူညီတဲ့ input တွေ function call တွေအတွက် ဆိုရင် ပထမသိမ်းထားတဲ့ result တွေထုပ်ပေးပြီး မလိုအပ်တဲ့ တွက်ချက်တာတွေကိုဖယ်ရှားပေးနိုင်ပါတယ်။ result တွေသိမ်းဖို့အတွက် array ကိုပဲ အသုံးများပေမယ့် တစ်ခြား structure တွေဖြစ်တဲ့ associative array တွေလည်းသုံးနိုင်ပါတယ်။

Memoization ကို programmer ကိုယ်တိုင်လုပ်နိုင်သလို python လို language မျိုးမှာတော့ meoization လုပ်ဖို့ Mechanism တွေပါပါတယ်။

Memoization with Function Decorators

Recursive function အခန်းမှာ Fibonacci number တွေကို iterative တွေ recursive တွေသုံးပြီး တွက်ခဲ့ပါတယ်။ recursive functionနဲ့ရေးတာက exponential behavior ဖြစ်တာလည်းသိပြီး ဖြစ်ပါလိမ့်မယ်။

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

dictionary သုံးပြီး function မှာ တွက်ခဲ့တာတွေမှတ်မိအောင်လုပ်ပြီး မြန်မြန်အောင်လုပ်တာလည်း ပြောပြခဲ့ပါတယ်။ ဒါကတော့ memoization ကိုကိုယ်တိုင်လုပ်ခဲ့တာဖြစ်ပါတယ်။ မကောင်းတာက တော့ရှင်းလင်းမှုတွေ recursive ရဲ့ပုံစံတွေပျောက်သွားတာဖြစ်ပါတယ်။

ပြဿနာက recursive fib function ကို ပြင်လိုက်လို့ဖြစ်တာပါ။ အောက်ဖော်ပြပါ code မှာတော့ fib function ကိုမပြင်ထားပါဘူး။ ဒါကြောင့် memoize ဆိုတဲ့ function တစ်ခုသုံးထားပါတယ်။ memoize() က function တစ်ခုကို argument အနေနဲ့လက်ခံပေးပါတယ်။ memoize() function က "memo" ဆိုတဲ့ dictionary သုံးပြီး result တွေကိုမှတ်ပါတယ်။ "memo"ရော "f" ရော local value တွေဆိုပေမယ့် helper method တွေသုံးပြီး return ပြန်ပါတယ်။ memoize(fib) ကိုခေါ်တဲ့ အခါမှာ helper() ရဲ့ referenceရယ် တွက်ထားတဲ့ result သိမ်းထားတဲ့ wrapper ရယ်ပြန်ပေးပါတယ်။

```
def memoize(f):
    memo = {}
    def helper(x):
```

```
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return helper


def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

fib = memoize(fib)
print(fib(40))
```

fib ကိုargument အနေနဲ့သုံးထားတဲ့ အကြောင်းကိုကြည့်ရအောင်

```
fib = memoize(fib)
```

ဒါဆိုရင် memoize က decorator ဖြစ်သွားပါတယ်။ fib = memoize(fib) အလုပ်မလုပ်ခင် function name တွေက သူတို့ body ကိုပဲ reference လုပ်ပါတယ်။

`fib = memoize(fib)`ကို execute လုပ်ပြီးတဲ့အချိန်မှာတော့ fib က helper function body ကို ညွှန်းပါတယ်။ မူလ fib function ကိုလဲ helper ရဲ့ f function ကနေမှတဆင့်သွားလို့ရတော့မှာ ဖြ ဖြစ်ပါတယ်။ fib အတွက် တစ်ခြား reference မရှိတော့ပါဘူး။ decorate လုပ်ထားတဲ့ Fibonacci function ကို `return fib(n-1) + fib(n-2)`နဲ့ခေါ်ထားပါတယ်။ memorize ကနေ return ပြန်လာ တဲ့ helper function ကိုပြောတာဖြစ်ပါတယ်။

decorator ထဲက context တွေကအခြား အဓိပ္ပာယ်ရှိပါသေးတယ်။ function တွေအများကြီးရေးမည့် အစား"memorize" နဲ့ decorate လုပ်တာဖြစ်ပါတယ်။

```
fib = memoize(fib)
func1 = memoize(func1)
func2 = memoize(func2)
func3 = memoize(func3)
# and so on
```

Decorator ကို python မှာရေးသလိုရေးမထားပါဘူး။အောက်ပါအတိုင်းရေးမယ့်အစား

```
fib = memoize(fib)
```

fib ကို "decorated" လုပ်သင့်ပါတယ်။

```
@memoize
```

ဒါပေမယ့် ဒီစာကြောင်းက decorated function ရှေ့မှာရှိရမှာပါ။

```
def memoize(f):
    memo = {}
    def helper(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return helper

@memoize
def fib(n):
    if n == 0:
        return 0
```

```
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

print(fib(40))
```

Using a Class Decorator for Memoization

Object orientation မသိသေးရင်ဒီအပိုင်းကျော်သွားလို့ရပါတယ်။

Result catching ကို class ထဲမှာလည်လုပ်နိုင်ပါတယ်။

```
class Memoize:
    def __init__(self, fn):
        self.fn = fn
        self.memo = {}
    def __call__(self, *args):
        if args not in self.memo:
            self.memo[args] = self.fn(*args)
        return self.memo[args]

@Memoize
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

dictionary သုံးတဲ့အတွက်ကြောင့် mutable argument တွေသုံးလို့မရပါဘူး။

# Chapter 23

Parameter Passing

"call by value" and "call by name"

Function တွေအတွက် argument passing လုပ်တဲ့အခါ နှစ်မျိုးသုံးကြပါတယ်။

- Call by Value

  အသုံးအများဆုံးကတော့ call by value သို့ pass-by-value ဖြစ်ပါတယ်။ call-by-value မှာတော့ arugment expression ကိုအရင်တွက် ပြီးမှ ရလာတဲ့ result ကို function ထဲမှာ ရှိတဲ့သက်ဆိုင်ရာ variable နဲ့ချိတ်ပေးပါတယ်။ expression က variable ဆိုရင် တန်ဖိုးကို local copy လုပ်ပြီးသုံးပါတယ်။အဲ့ဒါကြောင့် function call လုပ်တဲ့ဘက်မှာ variable အပြောင်း အလဲမရှိပါဘူး။

- Call by Reference

  Call by reference သို့ pass-by-reference လို့သိကြပါတယ်။ function က argument တွေရဲ့ value တွေအစား reference တွေသုံးပြီးအလုပ်လုပ်ပါတယ်။ ရလဒ်အနေနဲ့ function ကနေပြီးတော့ argument ကိုပြောင်းလဲပြင်ဆင်နိုင်ခြင်းပဲဖြစ်ပါတယ်။ ကောင်းတာ တစ်ချက်က argument တွေcopy ပွားစရာမလိုတာပါ။ မကောင်းတာကတော့ မတော်တဆ ပြင်ဆင်မိတာတွေဖြစ်နိုင်ပါတယ်။

and what about Python?

Python မှာတော့ call-by-object ကိုသုံးပါတယ်။ "Call by Object Reference" or "Call by Sharing" လို့လည်းခေါ်ကြပါတယ်။

Immutable argument တွေဖြစ်တဲ့ ntegers, strings or tuples to a function တွေနဲ့ဆိုရင် call-by-value ပုံစံအလုပ်လုပ်ပါတယ်။ function parameter ထဲကို object reference တွေရောက်သွား ပါတယ်။ သူတို့ကိုပြောင်းလဲလို့လုံးဂမရတဲ့အတွက်ကြောင့် function ထဲမှာလည်း ပြောင်းလို့မရပါဘူး(immutable)။ mutable argument တွေဆိုရင်တော့ ပြော ပြောင်းလဲလို့ရသွားပါတယ်.။ function ထဲကို list ပို့ချင်ရင် နှစ်မျိုးစဉ်းစားဖို့လိုပါတယ်။ list ကိုပြောင်း လဲလို့ရတယ်။ အသစ်ထည့်ပေးမယ်ဆိုရင်တော့ list အဟောင်းကိုသက်ရောက်မှုမရှိ ပါဘူး။

Integer value ကိုအရင်ကြည့်ရအောင်။ function ထဲက parameter သည် parameter မပြောင်း မချင်း argument variable ရဲ့ reference အဖြစ်ရှိပါတယ်။ တန်ဖိုးအသစ်တစ်ခုပြောင်းလဲလိုက်မယ် ဆိုရင်တော့ python က Local variable

အသစ်တစ်ခုဖန်တီးပါလိမ့်မယ်။ function call လုပ်တဲ့ (caller) ဘက်က variable ကပြောင်းမှာမဟုတ်တော့ပါဘူး။

```
def ref_demo(x):

    print "x=",x," id=",id(x)
    x=42
    print "x=",x," id=",id(x)
```

ဖော်ပြပါ ဥပမာမှာ id() function ကိုသုံးထားပါတယ်။ သူက object တစ်ခုကို parameter အဖြစ်သုံးပါတယ်။ id(obj) ဆိုရင် "obj" ရဲ့ "identity"ကို return ပြန်ပေးပါတယ်။ identity ဆိုတာကတော့ object အတက် သီးသန့် Integer တန်ဖိုတစ်ခုဖြစ်ပြီး unique and constant ဖြစ်ပါတယ်။ life time (non-overlapping lifetime)မထပ်တဲ့ မတူညီတဲ့ object နှစ်ခုဟာ identity တူတာမျိုးဖြစ်နိုင်ပါတယ်။

အောက်ပါအတိုင်း စစ်ကြည့်ရင် "x" က main scope မှာ id 41902552 ရှိပါတယ်။ ref_demo() function ရဲ့ ပထမဆုံးအကြောင်းမှာလည်း identity ကိုစစ်တဲ့အခါမှာလည်း တူနေတဲ့အတွက်ကြောင့် main scope က x ကိုအသုံးပြုတုန်းပါပဲ။ x ကို တန်ဖိုးအသစ်ဖြစ်တဲ့ 42 ပေးလိုက်တဲ့အခါမှာတော့ identity အသစ်အဖြစ် 41903752 ကိုပြောင်းလဲသွားပါတယ်။ global မှာရှိတဲ့ x နဲ့ memory location မတူတော့တာဖြစ်ပါတယ်။ ဒါကြောင့် main scope ကိုပြန်လာကြည့် တဲ့အခါ x ကနဂိုတန်ဖိုး 9 ကနေမပြောင်းလဲပဲရှိနေတာတွေ့နိုင်ပါတယ်။

ဆိုလိုတာက python ကအစတုန်းက call-by-reference ပုံစံမျိုးလုပ်ဆောင်ပေမယ့် variable value ကိုပြောင်းလဲတာနဲ့ call-by-value ပုံစံဖြစ်သွားပါတယ်။

```
>>> x = 9
>>> id(x)
41902552
>>> ref_demo(x)
x= 9  id= 41902552
x= 42  id= 41903752
>>> id(x)
41902552
>>>
```

Side effects

Function တစ်ခုမှာ တန်ဖိုးတစ်ခု ထုတ်ပေးတာအပြင် caller environement ကိုပြောင်းလဲတာ တွေဖြစ်စေရင် side effet ရှိတယ်ဆိုပါတယ်။ ဥပမာ- function က global/staitic variable တွေကိုပြင်တာ။ argument တွေကိုပြင်တာ။ exception တွေဖြစ်တာ။ display တွေ file တွေကို data တွေထည့်တာစတာတွေဖြစ်ပါတယ်။

များသောအားဖြင့် ဒါတွေကို function လိုအပ်ချက်အနေနဲ့အသုံးပြုကြပါတယ်။

List ကို function တစ်ခုဆီပို့တယ်ဆိုပါစို့။ function က list ကိုပြောင်းလဲမှုတွေမလုပ်ဘူးလို့ ယူဆမယ်။ side effect မရှိတာအရင်ကြည့်မယ်။ func1() ရဲ့ parameter list ကို list အသစ်တစ်ခု သတ်မှတ်ပေးလိုက်တဲ့အတွက် memory location အသစ်တစ်ခုတည်ဆောက်ပေးပါတယ်။

```
>>> def func1(list):
...     print list
...     list = [47,11]
...     print list
...
>>> fib = [0,1,1,2,3,5,8]
>>> func1(fib)
[0, 1, 1, 2, 3, 5, 8]
[47, 11]
>>> print fib
[0, 1, 1, 2, 3, 5, 8]
>>>
```

ဒါပေမယ့် List မှာ "+=" သုံးရင်တစ်မျိုးဖြစ်သွားပါပြီ။ example

```
>>> def func2(list):
...     print list
...     list += [47,11]
...     print list
...
>>> fib = [0,1,1,2,3,5,8]
>>> func2(fib)
[0, 1, 1, 2, 3, 5, 8]
[0, 1, 1, 2, 3, 5, 8, 47, 11]
>>> print fib
[0, 1, 1, 2, 3, 5, 8, 47, 11]
>>>
```

ဒါကိုရှောင်ရှားချင်ရင်တော့ list ကို copy ကူးပြီး function နဲ့အသုံးပြုရမှာပါ။ဒီမှာတော့ shallow copy နဲ့လုံလောက်ပါတယ်။

```
>>> def func2(list):
...     print list
...     list += [47,11]
...     print list
...
>>> fib = [0,1,1,2,3,5,8]
>>> func2(fib[:])
[0, 1, 1, 2, 3, 5, 8]
[0, 1, 1, 2, 3, 5, 8, 47, 11]
>>> print fib
[0, 1, 1, 2, 3, 5, 8]
>>>
```

Command Line Arguments

Python script ကို command line argument တွေသုံးပြီးရေးနိုင်ပါတယ်။ python script တွေကို shell ကနေအသုံးပြုတဲ့အခါမှာ argument တွေက script name

နောက်မှာရှိပါတယ်။ ပြီးတော့ argument တွေကြားထဲမှာ space ခံပါတယ်။ script ထဲကနေ ဒီ argument တွေကို သုံးချင်တယ် ဆိုရင်တော့ sys.argv ဆိုတဲ့ list variable ကိုသုံးရမှာပါ။ sys.argv[0] မှာ script name ၊ sys.argv[1] မှာ first parameter ၊ sys.argv[2]မှာ second parameter စသဖြင့်ပါပါတယ်။

အောက်ဖော်ပြပါ (arguments.py)မှာပါတာတွေဖြစ်ပါတ.ယ်

```
# Module sys has to be imported:
import sys

# Iteration over all arguments:
for eachArg in sys.argv:
        print eachArg
```

shell/cmd ကနေခေါ်ကြည့်မယ်။

```
python argumente.py python course for beginners
```

ထွက်လာတဲ့ output ကတော့

```
argumente.py
python
course
for
beginners
```

Variable Length of Parameters

Argument တွေအများကြီးလက်ခံနိုင်တဲ့ function တွေအကြောင်းပြောပါမယ်။ Python မှာ "*" ကို variable argument တွေကိုသတ်မှတ်ရင်သုံးပါတယ်။

```
>>> def varpafu(*x): print(x)
...
>>> varpafu()
()
>>> varpafu(34,"Do you like Python?", "Of course")
(34, 'Do you like Python?', 'Of course')
>>>
```

ဒီမှာ function ထဲကိုရောက်လာတာတွေကို tuple အနေနဲ့လက်ခံတာတွေ့ရပါလိမ့်မယ်။ argument မပါရင်တော့ empty tuple ပဲဖြစ်ပါလိမ့်မယ်။

တစ်ခါတစ်ရံ an arbitrary number of parameters(variable များ) နောက်မှာ positional parameter တွေပါတာမျိုးရှိနိုင်ပါတယ်။ဖြစ်နိုင်တယ်ဆိုပေမယ့် positional parameter က arbitrary parameter ရှေ့မှာအမြဲရှိရပါမယ်။ အောက်ဖော်ပြပါ ဥပမာမှာ "city" ဆိုတဲ့ positional parameter ပါပြီးတော့ ၎င်းနောက်မှာ arbitrary parameter တွေပါဝင်ပါတယ်။

```
>>> def locations(city, *other_cities): print(city, other_cities)
...
>>> locations("Paris")
('Paris', ())
>>> locations("Paris", "Strasbourg", "Lyon", "Dijon", "Bordeaux",
"Marseille")
('Paris', ('Strasbourg', 'Lyon', 'Dijon', 'Bordeaux', 'Marseille'))
>>>
```

List နဲ့သုံးချင်ရင်တော့ function call လုပ်ရာမှာ "*" ထည့်ပေးရပါမယ်။

* in Function Calls

"*" က function call လုပ်ရာမှာလည်းပါလာနိုင်ပါတယ်။ tuple တွေ List တွေဆိုရင် singular ပုံစံပြောင်းသွားပါတယ်။

```
>>> def f(x,y,z):
...     print(x,y,z)
...
>>> p = (47,11,12)
>>> f(*p)
(47, 11, 12)
```

အောက်ပါပုံစံအတိုင်းလည်းရပါတယ်။

```
>>> f(p[0],p[1],p[2])
(47, 11, 12)
>>>
```

Arbitrary Keyword Parameters

Arbitrary number of keyword parameter တွေအတွက်လည်း mechanism တွေရှိပါတယ်။ "**" ကိုသုံးပါတယ်။

```
>>> def f(**args):
...     print(args)
...
>>> f()
{}
>>> f(de="German",en="English",fr="French")
{'fr': 'French', 'de': 'German', 'en': 'English'}
>>>
```

Double Asterisk in Function Calls

"**"ကို function call မှာသုံးတဲ့ပုံစံ

```
>>> def f(a,b,x,y):
...     print(a,b,x,y)
...
>>> d = {'a':'append', 'b':'block','x':'extract','y':'yes'}
>>> f(**d)
('append', 'block', 'extract', 'yes')
```

"*" နဲ့တွဲသုံးတဲ့ပုံစံ

```
>>> t = (47,11)
>>> d = {'x':'extract','y':'yes'}
>>> f(*t, **d)
(47, 11, 'extract', 'yes')
>>>
```

# Chapter 24

Namespaces and Scopes

## Namespaces

Namespace ဆိုတာကတော့ system ထဲမှာ တစ်ခုနဲ့တစ်ခုမရုပ်ထွေးအောင်(ambiguity) နာမည်တွေကို unique(ထပ်တူမရှိတာ) ဖြစ်အောင်လုပ်တာဖြစ်ပါတယ်။ လူတွေမှာလည်း နာမည်တူရင်တောင် family name (surname)နဲ့ခွဲကြပါတယ်။ network တွေမှာဆိုရင် လည်း ချိတ်ဆက်မည့် device တစ်ခုခြင်းဆီမှာ unique name/ip address တွေရှိပါတယ်။ နောက်တစ်ခု က file system တွေရဲ့ directory structure ပါ။ file name တူရင်တောင် directory မတူရင်သုံး လို့ရပါတယ်။ programming language တွေမှာလည်း identifier တွေကို namespace တွေထဲမှာ သတ်မှတ်နိုင်ပါတယ်။ တူညီတဲ့ indentifier တွေကိုလည်း မတူညီတဲ့ namespace တွေထဲမှာ အသုံးပြုနိုင်ပါတယ်။

Python မှာ namespace တွေကို python dictionaries တွေနဲ့အသုံးပြုပါတယ်။ names(keys) object(values)အဖြစ်mapping လုပ်ထားတာဖြစ်ပါတယ်။ အသုံးပြုသူအနေနဲ့တော့ program ထဲမှာ namespace တွေအသုံးပြုဖို့ ဒါတွေသိစရာမလိုပါဘူး။

- **global names** of a module
- **local names** in a function or method invocation
- **built-in names**: this namespace contains built-in functions (e.g. abs(), cmp(), ...) and built-in exception names

## Lifetime of a Namespace

Script သို့ program ထဲမှာသုံးထားတဲ့ namespace တွေကို သုံးချင်တဲ့အချိန်သုံးလို့တော့မရပါဘူး။ namespace ဖန်တီးခဲ့တဲ့အချိန်တွေမတူညီတဲ့အတွက် မတူညီတဲ့ Lifetime တွေရှိကြပါတယ်။ အစကနေအဆုံး အထိရှိနေတဲ့ namespace တစ်ခုတော့ရှိပါတယ်။ built-in name တွေပါတဲ့ namespace ကတော့ python interpreter စ run ထဲကတည်ရှိပြီး ဘယ်တော့မှမဖျက်ပါဘူး။ module တစ်ခုကိုစတင်အလုပ်လုပ်တဲ့ အခါ ၎င်း module ရဲ့ global namespace ထုပ်ပေးပါတယ်။ module namespace တွေကတော့ script မဆုံးမခြင်း interpreter ကနေမထွက်မခြင်း တော့ရှိနေတတ်ပါတယ်။ function တစ်ခုကို callလုပ်ရာမှာ လည်း ၎င်း function အတွက် local namespace တစ်ခုပြုလုပ်ပေးပါတယ်။ function ဆုံးသွားရင်တော့ ဖျက်ပါတယ်။

## Scopes

Scope ကတော့ namespace တစ်ခုကိုတိုက်ရိုက်အသုံးပြုနိုင်တဲ့ နယ်ပယ်တစ်ခုဖြစ်ပါတယ်။ နောက်တစ်မျိုးပြောရရင် တော့ scope ဆိုသည်မှာ name တစ်ခုကို ambigus မရှိပဲ အသုံးပြုနိုင်သောနေရာဖြစ်ပါတယ်။ name တစ်ခုရဲ့ namespace ဆိုတာကတော့

သူ့ရဲ့ scope နဲ့ အတူတူပါပဲ.။ scope တွေကို static အဖြစ်သတ်မှတ်ထားပေမယ့် dynamic သုံးကြပါတယ်။

Program execution လုပ်ရာမှာ ရှိတဲ့ nested scope တွေကတော့

- အတွင်းအကျဆုံး scope မှာ အရင်ရှာပြီးတော့ ၎င်းမှာ local name တွေပါပါတယ်။
- တကယ်လို့ တစ်ခြား scope တစ်ခုထဲမှာရှိတယ်ဆိုရင်လည်း အနီးဆုံးကို အရင်ရှာပါတယ်။
- နောက် scope တွေမှာတော့ လက်ရှိ Module ရဲ့ global name တွေပါပါတယ်။
- အပြင်အကျဆုံးနဲ့ နောက်ဆုံးကတော့ built-in name တွေပါတဲ့ namespace ဖြစ်ပါတယ်။

# Chapter 25

## Global and Local Variables

### Global and local Variables in Functions

အောက်ပါ function ကိုကြည့်ကြည့်ပါ။

```
def f():

    print s
s = "I hate spam"
f()
```

variable s ကို "I hate spam" လို့ function f() ကိုမခေါ်ခင်သုံးထားပါတယ်။ f() ထဲ မှာက "print s" တစ်ကြောင်းပဲရှိပါတယ်။ s ဆိုတဲ့ local variable မရှိတဲ့အတွက် global variable s ကိုအသုံးပြု ပါလိမ့်မယ်။ ဒါကြောင့် output က "I hate spam"ဖြစ်ပါတယ်။ function f() ထဲမှာ s ကိုပြောင်းလဲ မှုတွေလုပ်ရင်ဘာဖြစ်မလဲ

```
def f():
    s = "Me too."
    print s

s = "I hate spam."
f()
print s
```

ရလဒ်ကတော့

```
Me too.
I hate spam.
```

ပထမ ဥပမာ နဲ့ဒုတိယဥပမာပေါင်းရေးကြည့်မယ်။ အရင် ဆုံး s ကိုယူသုံးမယ် ပြီးမှ တန်ဖိုးအသစ် assign လုပ်မယ်။

```
def f():
        print s
        s = "Me too."
        print s

s = "I hate spam."
f()
print s
```

ဒါဆိုရင်တော့ error ပြပါလိမ့်မယ်

```
UnboundLocalError: local variable 's' referenced before assignment
```

Pythonက f() ထဲမှာ s ကို assignment လုပ်ထားတာတွေ့တဲ့အတွက်ကြောင့် local variable ကိုသုံးချင်တာလို့ယူဆပါလိမ့်မယ်။ function ထဲမှာ ဖန်တီးသမျှ variable အားလုံးက Local ဖြစ်ပါတယ်။ global ကိုသုံးချင်တယ်ဆိုရင်တော့ "global" keyword ကိုသုံးပေးရပါမယ်။

```
def f():
    global s
    print s
    s = "That's clear."
    print s

s = "Python is great!"
```

```
f()
print s
```

ဒါဆိုရင်တော့ ambiguity မရှိတော့ပါ။

```
Python is great!
That's clear.
That's clear.
```

Function ထဲက local variable ကိုလည်း အပြင်ကနေ သုံးလို့မရပါဘူး။

```
def f():
    s = "I am globally not known"
    print s

f()
print s
```

run ကြည့်ရင် error ပြပါလိမ့်မယ်။

```
I am globally not known
Traceback (most recent call last):
  File "global_local3.py", line 6, in <module>
    print s
NameError: name 's' is not defined
```

Global နဲ့ local variable သုံးထားတဲ့ example

```
def foo(x, y):
    global a
    a = 42
    x,y = y,x
    b = 33
    b = 17
    c = 100
    print a,b,x,y

a,b,x,y = 1,15,3,4
foo(17,4)
print a,b,x,y
```

The output of the script above looks like this:

```
42 17 4 17
42 15 3 4
```

# Chapter 26

File I/O

Working with Files

File ဆိုတာကတော့ program တွေကနေ data တွေinformation တွေစုထားတဲ့ computer file တွေဖြစ်ပါတယ်။ file တွေကို durable storage တွေမှာသိမ်းပါတယ်။ လူတွေ program တွေကနေ file တွေကို ဖတ်ရှုပြင်ဆင် တာတွေလုပ်ဖို့အတွက် unique name နဲ့ path တွေအသုံးပြုကြပါတယ်။

"file" ဆိုတဲ့ အသုံးအနှုံးကတော့ punch card အသုံးပြုထဲ 1952 ထဲကပေါ်ပေါက်ခဲ့ပါတယ်။

အချက်အလက်တွေကို သိမ်းဆည်းမှုတွေ ပြန်လည်အသုံးပြုမှုတွေမလုပ်နိုင်တဲ့ programming language တစ်ခုကသိပ်အသုံးမဝင်နိုင်ပါဘူး။

File manipulation မှာ ပါတဲ့ အဓိကလုပ်ဆောင်ချက်တွေကတော့ file တွေကနေ data တွေ ဖတ်တာတွေ data တွေပြန်ထည့်တာ ထပ်ပေါင်းတာတွေဖြစ်ပါတယ်။

File ကနေ data ဖတ်ကြည့်ပါတယ်။ ပထမ parameter ကတော့ အသုံးပြုမယ့် file name ဖြစ်ပြီးတော့ ဒုတိယ parameter ကတော့ "r" လို့ပေးထားပြီး read လုပ်မယ်လို့ပြောတာဖြစ်ပါတယ်။

```
fobj = open("ad_lesbiam.txt", "r")
```

file နဲ့ပတ်သတ်တဲ့ အလုပ်တွေလုပ်ပြီးရင်တော့ close လုပ်ပေးရပါမယ်။

```
fobj.close()
```

examples

```
fobj = open("ad_lesbiam.txt")
for line in fobj:
    print line.rstrip()
fobj.close()
```

"file_read.py"လို့သိမ်းပြီး runကြည့်ရင်

```
$ python file_read.py
V. ad Lesbiam

VIVAMUS mea Lesbia, atque amemus,
rumoresque senum severiorum
omnes unius aestimemus assis!
soles occidere et redire possunt:
nobis cum semel occidit breuis lux,
nox est perpetua una dormienda.
da mi basia mille, deinde centum,
dein mille altera, dein secunda centum,
deinde usque altera mille, deinde centum.
dein, cum milia multa fecerimus,
conturbabimus illa, ne sciamus,
aut ne quis malus inuidere possit,
```

```
cum tantum sciat esse basiorum.
(GAIUS VALERIUS CATULLUS)
```

## Writing into a File

Write လုပ်တာလည်းလွယ်ပါတယ်။ writeလုပ်ဖို့ file ကိုဖွင့်မယ်ဆိုရင် second parameter ကို "r" အစား "w"ပြောင်းပေးရပါမယ်။ data write လုပ်ဖို့ file object ရဲ့ write() method ကိုသုံးရပါမယ်။

```
fobj_in = open("ad_lesbiam.txt")
fobj_out = open("ad_lesbiam2.txt","w")
i = 1
for line in fobj_in:
    print line.rstrip()
    fobj_out.write(str(i) + ": " + line)
    i = i + 1
fobj_in.close()
fobj_out.close()
```

input text file ရဲ့ စာကြောင်းတိုင်းမှာ line number လေးနဲ့စပါလိမ့်မယ်။

```
$ more ad_lesbiam2.txt
1: V. ad Lesbiam
2:
3: VIVAMUS mea Lesbia, atque amemus,
4: rumoresque senum seueriorum
5: omnes unius aestimemus assis!
6: soles occidere et redire possunt:
7: nobis cum semel occidit breuis lux,
8: nox est perpetua una dormienda.
9: da mi basia mille, deinde centum,
10: dein mille altera, dein secunda centum,
11: deinde usque altera mille, deinde centum.
12: dein, cum milia multa fecerimus,
13: conturbabimus illa, ne sciamus,
14: aut ne quis malus inuidere possit,
15: cum tantum sciat esse basiorum.
16: (GAIUS VALERIUS CATULLUS)
```

ဖြစ်နိုင်တဲ့ ပြဿနာတစ်ခုပြောပါမယ်။ ရှိနေပြီးသား file တစ်ခုကို write လုပ်ဖို့ open လုပ်မိရင်ရော။ ဒါဆိုရင်တော့ နဂိုရှိနေတဲ့ file ကိုဖျက်လိုက်ပါတယ်။

ရှိပြီးသား file ကိုမှ ထပ်ပေါင်းချင်ရင် "a" ကိုသုံးရပါမယ်။

## Reading in one

များစွာဆိုသလို file တွေကို for loop သုံးပြီး တစ်ကြောင်းချင်းအလုပ်လုပ်ကြပါတယ်။ file ကသိပ်မကြီးဘူးဆိုရင် file ရဲ့ data structre တစ်ခုလုံးကို read လုပ်တာပိုအဆင်ပြေပါတယ်.

```
>>> poem = open("ad_lesbiam.txt").readlines()
>>> print poem
['V. ad Lesbiam \n', '\n', 'VIVAMUS mea Lesbia, atque amemus,\n',
'rumoresque senum seueriorum\n', 'omnes unius aestimemus assis!\n', 'soles
```

```
occidere et redire possunt:\n', 'nobis cum semel occidit breuis lux,\n',
'nox est perpetua una dormienda.\n', 'da mi basia mille, deinde centum,\n',
'dein mille altera, dein secunda centum,\n', 'deinde usque altera mille,
deinde centum.\n', 'dein, cum milia multa fecerimus,\n', 'conturbabimus
illa, ne sciamus,\n', 'aut ne quis malus inuidere possit,\n', 'cum tantum
sciat esse basiorum.\n', '(GAIUS VALERIUS CATULLUS)']
>>> print poem[2]
VIVAMUS mea Lesbia, atque amemus,
```

ဖော်ပြပါ ဥပမာမှာ poem တစ်ခုလုံးကို list poem အနေနဲ့ read လုပ်ပေးပါတယ်။ line 3 ကိုလိုချင်ရင်ဂ poem[2] လို့အသုံးပြုနိုင်ပါတယ်။

နောက်တစ်နည်းကတော့ read() ဖြစ်ပါတယ်။ သူကတော့ file တစ်ခုလုံးကို string ပြောင်းပေးပါတယ်။

```
>>> poem = open("ad_lesbiam.txt").read()
>>> print poem[16:34]
VIVAMUS mea Lesbia
>>> type(poem)
<type 'str'>
>>>
```

String မှာ carriage return ၊ line feed တွေပါပါပါတယ်။

"How to get into a Pickle"

Data တွေကိုနောက်တစ်ခါလွယ်လွယ် ပြန်သုံးနိုင်အောင် ဘယ်လိုလုပ်မလဲ? Data တွေကို "pickling" လုပ်ပါမယ်။

Python မှာ "pickle"လို့ခေါ်တဲ့ Module တစ်ခုပါပါတယ်။ pickle module သုံးပြီးတော့ python object တွေကို serialize / de-serialize လုပ်နိုင်ပါတယ်။ "pickilng" ဆိုတာကတော့ python object တွေကို byte stream အဖြစ်ပြောင်းလဲတာဖြစ်ပြီး ။ "unpickling"ကတော့ ပြောင်းပြန်ဖြစ်ပါတယ်။ byte stream ကနေ python object ပြောင်းပေးတာပါ။ pickling and unpickling ကို data structre ကို "serialization" သို့ "flattening"လို့လည်းသိကြပါတယ်။

Pickle module ရဲ့ dump method နဲ့ dump(ပြောင်းလဲ) နိုင်ပါတယ်။

```
pickle.dump(obj, file[,protocol])
```

object "obj"ရဲ့ serialized version က file ထဲကိုရောက်သွားပါမယ်။ protocol ကတော့ object ကိုဘယ်လို write လုပ်မလဲပြောတာပါ။

- 0=ascii
- 1=compact (not human readable)
- 2=optimised classes

Pickle.dump method နဲ့ file ထဲကို dump လုပ်ခဲ့တဲ့ object တွေကို program ကနေပြီးတော့ pickle.load(file) နဲ့ ပြန်ဖတ်ပါတယ်။ pickle.load ကဘယ်format နဲ့ data write လုပ်ခဲ့လဲ အလိုအလျောက်သိပါတယ်.

```
import pickle
```

```
data = (1.4,42)
output = open('data.pkl', 'w')
pickle.dump(data, output)
output.close()
```

code execute လုပ်ပြီးရင်တော့ data.pkl ထဲမှာ ဒီလိုရှိပါတယ်။

```
(F1.3999999999999999
I42
tp0
.
```

File ကိုလွယ်ကူစွာပဲ read ပြန်လုပ်နိုင်ပါတယ်။

```
>>> import pickle
>>> f = open("data.pkl")
>>> data = pickle.load(f)
>>> print data
(1.3999999999999999, 42)
>>>
```

Object တွေသာ မှတ်ပေးတာဖြစ်ပြီး object name တွေတော့မပါပါဘူး။ ဒါကြောင့် data ကို assign လုပ်ခဲ့တာဖြစ်ပါတယ်။data = pickle.load(f).

# Chapter 27

Modular Programming and Modules

## Modular Programming

မိမိရဲ့ program တွေကိုလွယ်လွယ်ကူကူ ထိန်းသိမ်းနိုင်အောင် Modular software design တွေသုံးသင့်ပါတယ်။ modular programming ကတော့ code တွေကို သီးခြားအစိတ်အပိုင်းလေး တွေအဖြစ်ခွဲခြားတဲ့ software desing technique တစ်ခုပဲဖြစ်ပါတယ်။ ၎င်းအစိတ်အပိုင်း လေးတွေကို module လို့ခေါ်ပါတယ်။ ဒီလိုပိုင်းခြားရတဲ့အခါမှာ တစ်ပိုင်းနဲ့တစ်ပိုင်း မှီခိုမှု(dependency )နည်းအောင်းအလေးထားရပါတယ်။ Modular system တစ်ခုတည်ဆောက် ရာမှာ module တွေကိုသီးခြားစီတည်ဆောက်ပြီးတော့ ၎င်းတို့ပေါင်းစပ်ရမှ executable application တစ်ခုဖြစ်ပါတယ်။

Python မှာ module ဆိုတာကတော့ python statement တွေ definition တွေပါတဲ့ file တစ်ခုပါပဲ။ module နာမည်ကတော့ file name ပဲဖြစ်ပါတယ်။ နောက်က file extension ဖြစ်တဲ့ .py ထည့်စာမလိုပါဘူး။ fibonacci.py ဆိုရင် Module name က Fibonacci ဖြစ်ပါတယ်။
eg.fibonacci.py

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
def ifib(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

ဒီmodule ကို interactive shell မှာ Import လုပ်ပြီးအသုံးပြုနိုင်ပါတယ်။

```
>>> import fibonacci
>>> fibonacci.fib(30)
832040
>>> fibonacci.ifib(100)
354224848179261915075L
>>> fibonacci.ifib(1000)
43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875L
>>>
```

module function name တွေကိုတိုသွားအောင် local name တွေနဲ့အသုံးပြုနိုင်ပါတယ်။

```
>>> fib = fibonacci.ifib
>>> fib(10)
55
>>>
```

More on Modules

Module တွေထဲမှာ function တွေပါနိုင်သလို statement တွေလည်းပါနိုင်ပါတယ်။ ၎င်း statement တွေကို module initialize လုပ်ဖို့သုံးနိုင်ပါတယ်။ ပြီးတော့ ၎င်း statement တွေကို module import လုပ်တဲ့အခါမှာပဲ execute လုပ်ပါတယ်။

Module မှာ private symbol table ရှိပါတယ်။ module ထဲမှာ ရှိတဲ့ function တွေကတော့ ၎င်းကို global symbol table အဖြစ်အသုံးပြုကြပါတယ်။ user ရဲ့ global variable နဲ့ module ရဲ့ global variable တွေ နာမည်တူတဲ့အခါ clash (error)မဖြစ်အောင်ကာကွယ်ပေးပါတယ်။ module ရဲ့ global variable ကို function တွေလိုပဲ အသုံးပြုနိုင်ပါတယ်။ i.e. modname.name

Module မှာ အခြား module တွေကို import လုပ်နိုင်ပါတယ်။ script သို့ module တွေမှာ Import ကိုအရင်ထိပ်ဆုံးမှာ ရေးကြပါတယ်။

Importing Names from a Module Directly

Module ထဲက name တွေကို import လုပ်နေတဲ့ module ရဲ့ symbol table ထဲ တိုက်ရိုက် import လုပ်နိုင်ပါတယ်။

```
>>> from fibonacci import fib, ifib
>>> ifib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Local symbol table ထဲမှာတော့ ဘယ်Module ကနေimport လုပ်တယ်ဆိုတာရှိမှာမဟုတ်ပါဘူး။ module ထဲမှာ ရှိတဲ့ name အားလုံးကို import လုပ်နိုင်ပေမယ့် recommend မပေးပါဘူး။

```
>>> from fibonacci import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

မလုပ်သင့်ပေမဲ့interactive session တွေမှာ တော့ typing ရိုက်ရသက်သာစေပါတယ်။

အရေးကြီးတာတစ်ခုက interpreter session တစ်ခုမှာ တစ်ခါပဲ Import လုပ်နိုင်ပါတယ်။ တကယ် လို့module မှာ ပြောင်းလဲမှုတစ်ခုရှိရင် interpreter ကို restart လုပ်ရင်လုပ် မလုပ်ရင် reload(modulename) လုပ်ပေးရပါမယ်။

Executing Modules as Scripts

Python module တွေဟာ script တွေပဲဖြစ်တာကြောင့် script တွေလိုပဲ run နိုင်ပါတယ်။

```
python fibo.py
```

module ကို script အဖြစ်သုံးတဲ့အခါ execute လုပ်ပေမယ့် ချွင်းချက်တစ်ခုရှိပါတယ်။ system variable __name__ ကို "__main__" အဖြစ်သတ်မှတ်လိုက်တာပါ။ ဒါကြောင့် module ကို အခြေအနေ နှစ်မျိုးနဲ့ အသုံးပြုနိုင်အောင် ရေးသားနိုင်ပါတယ်။ အောက်ပါပုံစံအတိုင်းဆိုလျှင် file ကို module သို့ script အဖြစ်သုံးနိုင်ပါတယ်။ ဒါပေမယ့် script အဖြစ် သုံးမှသာလျှင် fib method က command line arguament တစ်ခုနဲ့ စတင်ပါလိမ့်မယ်။

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

script အဖြစ် သုံးကြည့်ရင် အောက်ပါအတိုင်းမြင်ရပါလိမ့်မယ်။

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Import လုပ်ပြီးသုံးရင်တော့ if block ထဲကအလုပ်လုပ်မှာမဟုတ်ပါဘူး

```
>>> import fibo
>>>
```

## Renaming a Namespace

Module Import လုပ်တဲ့အခါ namespace ရဲ့ name ကိုပြောင်းလဲနိုင်ပါတယ်။

```
>>> import math as mathematics
>>> print mathematics.cos(mathematics.pi)
-1.0
```

Namespace ရှိနေပေမဲ့ နာမည်တော့တူတော့မှာမဟုတ်ပါဘူး။ method အချို့ကိုပဲ Import လုပ်ချင်ရင်လည်းရပါတယ်။

```
>>> from math import pi,pow as power, sin as sinus
>>> power(2,3)
8.0
>>> sinus(pi)
1.2246467991473532e-16
```

## Kinds of Modules

Module အမျိုးအစားများစွာရှိပါတယ်

- Python မှာ ရေးသားထားတာတွေ
  They have the suffix: .py
- Dynamically linked C modules
  Suffixes are: .dll, .pyd, .so, .sl, ...
- C-Modules linked with the Interpreter:
  It's possible to get a complete list of these modules:
  - import sys
  - print sys.builtin_module_names

An error message is returned for Built-in-Modules.

Module Search Path

Module import လုပ်ရာမှာ interpreter က အောက်ဖော်ပြပါအစဉ်အတိုင်း module ကိုရှာပါတယ်။

1.top-level file ရှိတဲ့နေရာ ။လက်ရှိ run နေတဲ့ file ရှိရာနေရာဖြစ်ပါတယ်။

2. PYTHONPATH global variable ထဲမှာ သတ်မှတ်ထားတဲ့ directory တွေ

3.standard installation path eg./usr/lib/python2.5

Module import လုပ်ပြီးသွားရင် သူဘယ်မှာရှိလဲကြည့်နိုင်ပါတယ်။

```
>>> import math
>>> math.__file__
'/usr/lib/python2.5/lib-dynload/math.so'
>>> import random
>>> random.__file__
'/usr/lib/python2.5/random.pyc'
```

Content of a Module

Built-in function dir() နဲ့ module နဲ့သက်ဆိုင်တဲ့ attribute တွေmethod တွေကို သိရှိနိုင်ပါတယ်။

```
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin',
'atan', 'atan2', 'ceil', 'cos', 'cosh', 'degrees',
'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot',
'ldexp', 'log', 'log10', 'modf', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh']
>>>
```

Dir() မှာ argument မပါရင်တော့ လက်ရှိ local scoper ထဲက name တွေပါပါမယ်။

```
>>> import math
>>> col = ["red","green","blue"]
>>> dir()
['__builtins__', '__doc__', '__name__', 'col', 'math']
```

Built-in function တွေကိုလည်းကြည့်နိုင်ပါတယ်။

```
>>> import __builtin__
>>> dir(__builtin__)
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
'Exception', 'False', 'FloatingPointError', 'FutureWarning',
'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning',
...
```

Module Packages

Module တွေကို package ထဲမှာလည်း ထည့်နိုင်ပါတယ်။ pyton code တွေစုထားတဲ့ directory ကို package လို့ခေါ်ပါတယ်။ package က Moudle import လုပ်သလိုပဲသုံးပါတယ်။

Package ထဲမှာ ပါဝင်တဲ့ directory တိုင်းမှာ __init__.py ဆိုတဲ့ file ပါဝင်ရပါမယ်။ မဟုတ်ရင်တော့ import လုပ်လို့မရပါဘူး။

# Chapter 28

Regular Expressions

"regular expression"ကို တစ်ခါတစ်ရံ regex သို့ rexexp လို့လည်းခေါ်ကြပါတယ်။

Text တွေ textstring တွေကို filtering လုပ်ဖို့ regular expression တွေကိုသုံးကြပါတယ်။

Regular expression အတွက် syntax တွေဟာ programming language တွေအားလုံးမှာ အတူတူပဲဖြစ်ပါတယ်။

Introduction

Sequential data type တွေတုန်းက "in" operator ကိုပြောခဲ့ပါတယ်။ string တစ်ခုထဲမှာ အခြား string တစ်ခုပါလား စစ်ကြည့်ရအောင်

```
>>> s = "Regular expressions easily explained!"
>>> "easily" in s
True
>>>
```

တဆင့်ခြင်းကြည့်ရအောင်

String sub="abc"

က string s= "xaababcbcd" မှာပါလားစစ်တယ်။

အရင်ဆုံးပထမနေရာတူလားစစ်မယ် i.e. s[0] == sub[0].

မတူဘူးဆိုတော့ နောက်တစ်ခုကိုရွှေ့မယ်။

ပထမတူရင် ဒုတိယတူလားဆက်စစ်ရမယ် မတူဆက်ရွှေ့ ပ ဒု တူရင် တတိယစစ် မတူရွှေ့ တူရင် ok ပေါ့။

အကုန်ကုန်လို့မှမတူရင်တော့ s ထဲမှာ sub မပါဘူးပေါ့။

A Simple Regular Expression

ဒါကို regular expression သုံးပြီး လွယ်ကူစွာဖြေရှင်းနိုင်ပါတယ်။ pyrhon မှာ regular expression သုံးမယ်ဆိုရင် "re" module ကို import လုပ်ပေးရပါမယ်။

Representing Regular Expressions in Python

အခြား Language တွေမှာ regular expression ကို "/"ကြားမှာ ရေးပါလိမ့်မယ်။ python မှာတော့ regular expression ကို ပုံမှန် string တွေလိုပဲရေးပါတယ်။

ပြဿနာတစ်ခုကတော့ backslash က regular expression မှာ special character အနေနဲ့ အသုံးပြု ပြီးတော့ string မှာ ကတော့ escape character အဖြစ်အသုံးပြုပါတယ်။ ဒီ အခါမှာ python က string ရဲ့ backslash တွေအဖြစ် evaluate အရင်လုပ်မှာဖြစ်ပြီး ကျန်တာတွေကိုမှ regular expression အဖြစ်သတ်မှတ်ပါလိမ်မယ့်။ ဒါကိုကာကွယ်ဖို့အတွက် double backslash "\\" အသုံးပြုနိုင်ပါတယ်။ အကောင်းဆုံးကတော့ regular expression တွေကို raw string အဖြစ်အသုံးပြုတာဖြစ်ပါတယ်။

```
r"^a.*\.html$"
```

ဒီဥပမာမှာ a နဲ့ စတင်ပြီး .html နဲ့ အဆုံးသတ်တဲ့ file အားလုံးဖြစ်ပါတယ်။

Syntax of Regular Expressions

r"cat" ဆိုတာ regular expression တစ်ခုပါ။ သူက "A cat and a rat can't be friends." နဲ့ ဆိုရင် match ဖြစ်ပါတယ်။

ဒါပေမယ့်ဒီမှာ အမှားတစ်ခုရှိပါတယ်။ cat လို့ဆိုလိုပေမယ့် အခြားစကားလုံးတွေနဲ့လည်း ကိုက်ညီနေပါသေးတယ်။ ဥပမာ "education" "communicate", "falsification", "ramifications", "cattle" စသဖြင့်စကားလုံးတွေပါခဲ့ရင်လည်း match ဖြစ်နေပါလိမ့်မယ်။ ဒါကို "over matching"လို့ဆိုပါတယ်။

ဒါကိုရှောင်ရှားချင်တဲ့အခါ r" cat "ဆိုပြီး ဘေးမှာ black တွေထားချင်ထားနိုင်ပါတယ်။ ခုနစကားလုံးတွေမပါတော့ပေမယ့် နောက်ပြဿနာတစ်ခုထပ်ဖြစ်ပါတယ်။ "The cat, called Oscar, climbed on the roof." မှာဆိုရင် cat နောက်မှာ ",”ပါလာတဲ့အတွက် match မဖြစ်တော့ပါ ဘူး။

Python မှာ regular expression ကိုဘယ်လိုသုံးလဲအရင်ကြည့်ရအောင်။

```
>>> import re
>>> x = re.search("cat","A cat and a rat can't be friends.")
>>> print x
<_sre.SRE_Match object at 0x7fd4bf238238>
>>> x = re.search("cow","A cat and a rat can't be friends.")
>>> print x
None
```

အထက်ဖော်ပြပါဥပမာမျာ regular expression ကိုအသုံးပြုဖို့ re module ကို import လုပ်ထားတာတွေ့ရပါမယ်။ အရေးအကြီးဆုံး နဲ့ အသုံးအများဆုံးဖြစ်ပါတယ်။ re.search(expr,s) ကတော့ s ဆိုတဲ့ string ထဲမှာ expr ခေါ် regular expression နဲ့ကိုက်တဲ့ substring တွေရှာပေးပါတယ်။ ပထမဆုံး ကိုက်ညီတဲ့ substring ကို ထုပ်ပေးပါတယ်။ match ဖြစ်ရင် match object ကို result အဖြစ်ထုတ်ပေးပြီး မဖြစ်ရင်တော့ none ပါ။

```
>>> if re.search("cat","A cat and a rat can't be friends."):
...     print "Some kind of cat has been found :-)"
... else:
...     print "No cat has been found :-("
...
```

```
Some kind of cat has been found :-)
>>> if re.search("cow","A cat and a rat can't be friends."):
...     print "Cats and Rats and a cow."
... else:
...     print "No cow around."
...
No cow around.
```

Any Character

အပေါ်မှာပြခဲ့တဲ့ example လို cat ကိုမရှာပဲ at နဲ့ အဆုံးသတ်တဲ့ character သုံးလုံးပါ ကိုရှာမယ်။

“any character” အတွက် က “.”ဖြစ်ပါတယ်။ ဒါကြောင့် ခု example အတွက်ဆိုရင်

r“ .at ”

ဖြစ်ပါတယ်။ဒါဆိုရင် "rat", "cat", "bat", "eat", "sat" တွေအခြားတိုက်ဆိုင်တာတွေရလာပါလိမ့်မယ်။

ဒါပေမယ့် "@at" or "3at" လိုစကားလုံးတွေပါရင်လည်းဖော်ပြပါလိမ့်မယ်။ overmatching ပါပဲ

Character Classes

Square bracket “[” “]” တွေကိုပါစေချင်တဲ့စကားလုံးတွေရေးရင်သုံးပါတယ်။ [xyz]ဆိုရင် x သို့ y သို့ z လို့ဆိုလိုပါတယ်။

```
r"M[ae][iy]er"
```

German surname တစ်ခုနဲ့တိုက်တာရှာတာပါ။ Maier, Mayer, Meier, Meyer စတဲ့ သံတူကြောင်း ကွဲ စကားလုံးတွေအတွက်ဖြစ်ပါတယ်။

အခုလို စကားလုံးတစ်လုံးနှစ်လုံးအတွက် မဟုတ်ပဲ အစုလိုက်စစ်တာတွေလည်းဖြစ်နိုင်ပါတယ်။

ဒါဆိုရင်တော့ “-” ကိုသုံးရပါမယ်။ [a-e] ဆိုရင် [abcde] [0-5]ဆိုရင် [012345]ဖြစ်ပါတယ်။

“-” ကိုစကားလုံးကြားတွေထဲမှာမသုံးပဲ ရှေ့မှာသို့ နောက်မှာသုံးရင်တော့ “-”တစ်ခုကိုဆိုလိုပါတယ်။ [-az] ဆိုရင် - or a or z လို့ဆိုလိုပါတယ်။

Predefined Character Classesing list:

\d =[0-9]

\D=[^0-9]

\s=[\t\n\r\f\v]

\S=[^\t\n\r\f\v]

\w=[a-zA-Z0-9]

\W=\w မဟုတ်တာ

\b=empty string ဖြစ်ပြီးတော့ စကားလုံးတစ်ခုရဲ့ရှေ့ သို့ နောက်မှာရှိမယ်။

\B= empty string ဖြစ်ပြီးတော့ စကားလုံးတစ်ခုရဲ့ရှေ့ သို့ နောက်မှာ မရှိရဘူး

\\=\

Matching Beginning and End

the expression r"M[ae][iy]er" က string ထဲမှာ ရှိတဲ့ဘယ်နေရာမှာမဆိုရှာဖွေနိုင်ပါတယ်။

```
>>> import re
>>> line = "He is a German called Mayer."
>>> if re.search(r"M[ae][iy]er",line): print "I found one!"
...
I found one!
>>>
```

မိမိက string အစမှာပဲရှာချင်တယ်ဆိုရင်ဘယ်လိုလုပ်မလဲ၊ re module မှာ matching အတွက် function နှစ်ခုပါပါတယ်။ search() and match()

```
>>> import re
>>> s1 = "Mayer is a very common Name"
>>> s2 = "He is called Meyer but he isn't German."
>>> print re.search(r"M[ae][iy]er", s1)
<_sre.SRE_Match object at 0x7fc59c5f26b0>
>>> print re.search(r"M[ae][iy]er", s2)
<_sre.SRE_Match object at 0x7fc59c5f26b0>
>>> print re.match(r"M[ae][iy]er", s1)
<_sre.SRE_Match object at 0x7fc59c5f26b0>
>>> print re.match(r"M[ae][iy]er", s2)
None
>>>
```

ဒါဆိုရင်တော့ရပါပြီ ဒါပေမယ့် ဒါက python မှာပဲသုံးလို့ရမှာပါ။

"^" က string အစကိုပြောတာဖြစ်ပါတယ်။ MULTILINE mode မှာတော့ newline စတိုင်း match ဖြစ်ပါတယ်။

```
>>> import re
>>> s1 = "Mayer is a very common Name"
>>> s2 = "He is called Meyer but he isn't German."
>>> print re.search(r"^M[ae][iy]er", s1)
<_sre.SRE_Match object at 0x7fc59c5f26b0>
>>> print re.search(r"^M[ae][iy]er", s2)
None
```

ဒါပေမယ့် s1 နဲ့ s2 ကို ဒီလိုပေါင်းလိုက်ရင်ဘာဖြစ်နိုင်ပါသလဲ။

```
s = s2 + "\n" + s1
```

maier လိုစကားလုံးအစတွေမရှိပေမဲ့ newline character တစ်ခုပါလာပါတယ်။

```
>>> s = s2 + "\n" + s1
>>> print re.search(r"^M[ae][iy]er", s)
None
>>>
```

String အစကိုပဲစစ်တာဖြစ်တဲ့အတွက် name ကိုမတွေ့တော့ပါဘူး။ multine mode သုံးရင်တော့ တစ်မျိုးဖြစ်သွားပါလိမ့်မယ်။

```
>>> print re.search(r"^M[ae][iy]er", s, re.MULTILINE)
<_sre.SRE_Match object at 0x7fc59c5f26b0>
>>> print re.search(r"^M[ae][iy]er", s, re.M)
<_sre.SRE_Match object at 0x7fc59c5f26b0>
>>> print re.match(r"^M[ae][iy]er", s, re.M)
None
>>>
```

စာကြောင်းအစမှာ ရှာပြီးတော့ အဆုံးမှာရှာရအောင် ဒါဆိုရင်တော့ "$" ကိုသုံးပါမယ်။

```
>>> print re.search(r"Python\.$","I like Python.")
<_sre.SRE_Match object at 0x7fc59c5f26b0>
>>> print re.search(r"Python\.$","I like Python and Perl.")
None
>>> print re.search(r"Python\.$","I like Python.\nSome prefer Java or
Perl.")
None
>>> print re.search(r"Python\.$","I like Python.\nSome prefer Java or
Perl.", re.M)
<_sre.SRE_Match object at 0x7fc59c5f26b0>
>>>
```

Optional Items

Mayer ဆိုတဲ့ နာမည်မှာအရင် နာမည်တွေအပြင် e ကိုဖြုတ်ထားတဲ့ပုံစံတွေ ["Mayr", "Meyr", "Meir", "Mair"] plus our old set ["Mayr", "Meyr", "Meir", "Mair"].ရှိပါသေးတယ်။

e ကချင်ပါနိုင်တယ်လို့ပြောချင်ရင် "?" လေးသုံးပါတယ်။

```
r"M[ae][iy]e?r"
```

"feb 2011" သို့ "February 2011" အတွက်ဆိုလျှင်

```
r"Feb(ruary)? 2011"
```

Quantifiers

zero to infinity အကြိမ်ရေအတွက်ဆိုရင် "*"

```
r"[0-9]*"
```

အကြိမ်အရေအတွက်အတိအကျ သို့ range တစ်ခုနဲ့ရေးချင်တယ်ဆိုရင် {}ထဲမှာရေးပေးရမယ်

```
r"^[0-9]{4} [A-Za-z]*"
```

min max အတွက်ဆိုရင် {from, to}ပုံစံရေးရမယ်

```
r"^[0-9]{4,5} [A-Z][a-z]{2,}"
```

Grouping

Regurlar expression တွေကို group ဖွဲ့ချင်ရင် () နဲ့ အသုံးပြုနိုင်ပါတယ်။

Capturing Groups and Backreferences

A Closer Look at the Match Objects

Re.search() method သုံးတာမှာ match ဖြစ်ရင် Match object return ပြန်ပြီး မဖြစ်ရင် None ပါ။ match object ထဲမှာ data တွေအမြားအပြားပါဝင်ပါတယ်။

Match object မှာ group(),span(),start(),end() စတဲ့ method တွေပါဝင်ပါတယ်။

```
>>> import re
>>> mo = re.search("[0-9]+", "Customer number: 232454, Date: February 12,
2011")
>>> mo.group()
'232454'
>>> mo.span()
(17, 23)
>>> mo.start()
17
>>> mo.end()
23
>>> mo.span()[0]
17
>>> mo.span()[1]
23
>>>
```

Span() က အစနဲ့အဆုံး position တွေ return ပြန်ပေးပါတယ်။ start() နဲ့ end() ကလည်း ဒီသဘောပါပဲ။ group() ကတော့argument မပါရင် regular expression နဲ့တိုက်တဲ့ substring ကိုထုတ်ပေးပါတယ်။

```
>>> import re
>>> mo = re.search("([0-9]+).*: (.*)", "Customer number: 232454, Date:
February 12, 2011")
```

```
>>> mo.group()
'232454, Date: February 12, 2011'
>>> mo.group(1)
'232454'
>>> mo.group(2)
'February 12, 2011'
>>> mo.group(1,2)
('232454', 'February 12, 2011')
>>>
```

Xml/html tag တွေနဲ့ဥပမာကြည့်ကြည့်ရအောင်

```
<composer>Wolfgang Amadeus Mozart</composer>
<author>Samuel Beckett</author>
<city>London</city>
```

ဒီtextကိုအောက်ပါအတိုင်းပြောင်းချင်တယ်။

```
composer: Wolfgang Amadeus Mozart
author: Samuel Beckett
city: London
```

အောက်ပါ python script လေးကအလုပ်လုပ်ပေးပါလိမ့်မယ်။ အဓိကကတော့ regular expresson ပါ။ "<" အရင်ရှာတယ်။ ">" မတွေ့မချင်း lower case letter တွေကိုဖတ်တယ်။ "<" ">" ကြားထဲတွေ့သမျှကို backreference အနေနဲ့သိမ်းထားပြီး \1 ဆိုတဲ့ expression နဲ့ ပြန်သုံးနိုင်တယ်။expression က ">"ကိုရောက်တဲ့အခါမှာ \1 မှာ composer ပါတယ်ဆိုပါဆို့။

```
import re
fh = open("tags.txt")
for i in fh:
     res = re.search(r"<([a-z]+)>(.*)</\1>",i)
     print res.group(1) + ": " + res.group(2)
```

() တစ်စုံထက်ပိုရင် backreference တွေကို \1,\2,\3 စသဖြင့်သတ်မှတ်ပါတယ်။

Named Backreferences

အပေါ်မှာ ပြခဲ့တာကို "Numbered Capturing Groups" and "Numbered Backreferences" လို့ခေါ်နိုင်ပါတယ်။ numbered capturing group အစား capturing group တွေသုံးရင် တော့ automatic number တွေအစား နာမည်လေးတွေပေးပြီးအသုံးပြုနိုင်ပါတယ်။

```
>>> import re
>>> s = "Sun Oct 14 13:47:03 CEST 2012"
>>> expr = r"\b(?P<hours>\d\d):(?P<minutes>\d\d):(?P<seconds>\d\d)\b"
>>> x = re.search(expr,s)
>>> x.group('hours')
'13'
>>> x.group('minutes')
'47'
>>> x.start('minutes')
14
>>> x.end('minutes')
16
>>> x.span('seconds')
```

```
(17, 19)
>>>
```

## Comprehensive Python Exercise

In this comprehensive exercise, we have to bring together the information of two files. In the first file, we have a list of nearly 15000 lines of post codes with the corresponding city names plus additional information. Here are some arbitrary lines of this file:

```
68309,"Mannheim",8222,"Mannheim",8,"Baden-Wrttemberg"
68519,"Viernheim",6431,"Bergstraße",6,"Hessen"
68526,"Ladenburg",8226,"Rhein-Neckar-Kreis",8,"Baden-Württemberg"
68535,"Edingen-Neckarhausen",8226,"Rhein-Neckar-Kreis",8,"Baden-
Württemberg"
```

The other file contains a list of the 19 largest German cities. Each line consists of the rank, the name of the city, the population, and the state (Bundesland):

```
1.  Berlin          3.382.169 Berlin
2.  Hamburg         1.715.392 Hamburg
3.  München         1.210.223 Bayern
4.  Köln              962.884 Nordrhein-Westfalen
5.  Frankfurt am Main 646.550 Hessen
6.  Essen             595.243 Nordrhein-Westfalen
7.  Dortmund          588.994 Nordrhein-Westfalen
8.  Stuttgart         583.874 Baden-Württemberg
9.  Düsseldorf        569.364 Nordrhein-Westfalen
10. Bremen            539.403 Bremen
11. Hannover          515.001 Niedersachsen
12. Duisburg          514.915 Nordrhein-Westfalen
13. Leipzig           493.208 Sachsen
14. Nürnberg          488.400 Bayern
15. Dresden           477.807 Sachsen
16. Bochum            391.147 Nordrhein-Westfalen
17. Wuppertal         366.434 Nordrhein-Westfalen
18. Bielefeld         321.758 Nordrhein-Westfalen
19. Mannheim          306.729 Baden-Württemberg
```

Our task is to create a list with the top 19 cities, with the city names accompanied by the postal code. If you want to test the following program, you have to save the list above in a file called largest_cities_germany.txt and you have to download and save the list of German post codes

```
# -*- coding: iso-8859-15 -*-

import re


fh_post_codes = open("post_codes_germany.txt")
PLZ = {}
for line in fh_post_codes:
    (post_code, city, rest) = line.split(",",2)
    PLZ[city.strip("\"")] =  post_code


fh_largest_cities = open("largest_cities_germany.txt")

for line in fh_largest_cities:
```

```
    re_obj = re.search(r"^[0-9]{1,2}\.\s+([\wÄÖÜäöüß\s]+\w)\s+[0-9]",line)
    city = re_obj.group(1)
    print city, PLZ[city]
```

## Another Comprehensive Example

We want to present another real life example in our Python course. A regular expression for UK postcodes.
We write an expression, which is capable of recognizing the postal codes or postcodes of the UK.

Postcode units consist of between five and seven characters, which are separated into two parts by a space. The two to four characters before the space represent the so-called outward code or out code intended to direct mail from the sorting office to the delivery office. The part following the space, which consists of a digit followed by two uppercase characters, comprises the so-called inward code, which is needed to sort mail at the final delivery office. The last two uppercase characters can be only one of these ABDEFGHJLNPQRSTUWXYZ

The outward code can have the form: One or two uppercase characters, followed by either a digit or the letter R, optionally followed by an uppercase character or a digit. (We do not consider all the detailed led rules for postcodes, i.e only certain character sets are valid depending on the position and the context.)
A regular expression for matching this superset of UK postcodes looks like this:

```
r"\b[A-Z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}\b"
```

The following Python program uses the regexp above:

```
import re

example_codes = ["SW1A 0AA", # House of Commons
                 "SW1A 1AA", # Buckingham Palace
                 "SW1A 2AA", # Downing Street
                 "BX3 2BB", # Barclays Bank
                 "DH98 1BT", # British Telecom
                 "N1 9GU", # Guardian Newspaper
                 "E98 1TT", # The Times
                 "TIM E22", # a fake postcode
                 "A B1 A22", # not a valid postcode
                 "EC2N 2DB", # Deutsche Bank
                 "SE9 2UG", # University of Greenwhich
                 "N1 0UY", # Islington, London
                 "EC1V 8DS", # Clerkenwell, London
                 "WC1X 9DT", # WC1X 9DT
                 "B42 1LG", # Birmingham
                 "B28 9AD", # Birmingham
                 "W12 7RJ", # London, BBC News Centre
                 "BBC 007" # a fake postcode
                ]

pc_re = r"[A-z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}"

for postcode in example_codes:
    r = re.search(pc_re, postcode)
    if r:
        print postcode + " matched!"
```

```
else:
    print postcode + " is not a valid postcode!"
```

# Chapter 29

Regular Expressions(Advanced)

Regular expression introduction မှာ regular expression အတွက် basic concept တွေ လေ့လာခဲ့ကြပါတယ်။ ရိုးရှင်းတဲ့ regular expression တွေ။ re module က search() match() method တွေ။ formatting နဲ့ built-in character clsss တွေ \d, \D, \s, \S, တွေလိုမျိုး predefined character class တွေ။ string အစနဲ့အဆုံးတွေဘယ်လို တိုက်မလဲ။optional item တွေအတွက် ? အသုံးပြုပုံတွေ။ quantifier တွေသုံးပြီး အကြိမ်အရေအတွက် အတွက်သုံးတာတွေ စသဖြင့်သိခဲ့ပါတယ်။

Grouping နဲ့ backreference လည်းသိရပါမယ်။ နောက်ပြီးတော့ re module ရဲ့ match object တွေ သူမှာ ဘာ information တွေပါလဲ` information တွေကိုspan(), start(), end(), and group() တွေသုံးပြီးဘယ်လိုရနိုင်မလဲစသဖြင့်သိရပါမယ်။

Finding all Matched Substrings

Re module မှာ အခြား Language တွေမှာမပါတဲ့ method တစ်ခုပါပါသေးတယ်။

```
re.findall(pattern, string[, flags])
```

ဆိုရင် တိုက်ဆိုင်တဲ့ string pattern တွေကို string list အနေနဲ့ return ပြန်ပေးပါတယ်။ ဘယ်မှာ ညာ စစ်တာဖြစ်ပြီးတော့ တွေ့ရှိတဲ့ အစဉ်အတိုင်း return ပြန်ပါတယ်။

```
>>> t="A fat cat doesn't eat oat but a rat eats bats."
>>> mo = re.findall("[force]at", t)
>>> print mo
['fat', 'cat', 'eat', 'oat', 'rat', 'eat']
```

Pattern ထဲမှာ group တွေပါရင် group တွေကို List အနေနဲ့ return ပြန်ပေးပါတယ်။ group တစ်ခုထက်ပိုရင်တော့ tuples တွေကို list အနေနဲ့ return ပြန်ပေးပါတယ်။

```
>>> import re
>>> items = re.findall("[0-9]+.*: .*", "Customer number: 232454, Date:
February 12, 2011")
>>> print items
['232454, Date: February 12, 2011']
>>> items = re.findall("([0-9]+).*: (.*)", "Customer number: 232454, Date:
February 12, 2011")
>>> print items
[('232454', 'February 12, 2011')]
>>>
```

Alternations

Regular expression introduction မှာ character classes တွေအကြောင်းပြောခဲ့တယ်။ character class တွေမှာဆိုရင် ကိုယ်လိုချင်တဲ့ character set

ကိုအသုံးပြုနိုင်တယ်။ နောက်တစ်မျိုးကတော့ regular expression တွေထဲကမှာ ပြ ပြန်ရွေးတာဖြစ်ပါတယ်။ logical or ဖြစ်ပါတယ်။ "|"

အောက်ဖော်ပြပါ ဥပမာမှာ မြို့တွေထဲကနေရွေးတာဖြစ်ပါတယ်။ပေးထားတဲ့မြို့တွေထဲက တစ်ခုပေါ့

```
>>> import re
>>> str = "The destination is London!"
>>> mo = re.search(r"destination.*(London|Paris|Zurich|Strasbourg)",str)
>>> if mo: print mo.group()
...
destination is London
>>>
```

## Compiling Regular Expressions

Regexp တစ်ခုထဲကို ထပ်ခါထပ်ခါပြန်သုံးချင်ရင် regular expression object ကိုသုံးသင့်ပါတယ်။

```
re.compile(pattern[, flags])
```

compile က regex object ကို return ပြန်ပေးပါတယ်။ သူ့ကိုသုံးပြီးရှာဖွေတာတွေ အစားထိုးတာတွေ လုပ်နိုင်ပါတယ်။ expression ရဲ့ behavior ကိုတော့ flag value နဲ့သတ်မှတ်ပေးပါတယ်။

| **Abbreviation** | **Full name** | **Description** |
|---|---|---|
| re.I | re.IGNORECASE | Makes the regular expression case-insensitive |
| re.L | re.LOCALE | The behaviour of some special sequences like \w, \W, \b,\s, \S will be made dependant on the current locale, i.e. the user's language, country aso. |
| re.M | re.MULTILINE | ^ and $ will match at the beginning and at the end of each line and not just at the beginning and the end of the string |
| re.S | re.DOTALL | The dot "." will match every character **plus the newline** |
| re.U | re.UNICODE | Makes \w, \W, \b, \B, \d, \D, \s, \S dependent on Unicode character properties |
| re.X | re.VERBOSE | Allowing "verbose regular expressions", i.e. whitespace are ignored. This means that spaces, tabs, and carriage returns are not matched as such. If you want to match a space in a verbose regular expression, you'll need to escape it by escaping it with a backslash in front of it or include it in a character class.<br># are also ignored, except when in a character class or preceded by an non-escaped backslash. Everything following a "#" will be ignored until the end of the line, so this character can be used to start a comment. |

Example

UK postcodes အတွက် regular expression

```
r"[A-z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}"
```

ဒါကို regex object နဲ့ သုံးမယ်။

```
>>> import re
>>> regex = r"[A-z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}"
>>> address = "BBC News Centre, London, W12 7RJ"
>>> compiled_re = re.compile(regex)
>>> res = compiled_re.search(address)
>>> print res
<_sre.SRE_Match object at 0x7fc9f688f6b0>
>>>
```

Splitting a String With or Without with Regular Expressions

String တွေကို substring တွေအဖြစ်ပိုင်းတဲ့ split ဆိုတဲ့ string method တစ်ခုရှိပါတယ်။

```
str.split([sep[, maxsplit]])
```

Split မှာ optional parameter နှစ်ခုရှိတယ်။ ဘာမှမထည့်ပေးရင် white space တွေသုံးပြီး ပိုင်းပေးမယ်။

Exampls

```
>>> law_courses = "Let reverence for the laws be breathed by every American
mother to the lisping babe that prattles on her lap. Let it be taught in
schools, in seminaries, and in colleges. Let it be written in primers,
spelling books, and in almanacs. Let it be preached from the pulpit,
proclaimed in legislative halls, and enforced in the courts of justice.
And, in short, let it become the political religion of the nation."
>>> law_courses.split()
['Let', 'reverence', 'for', 'the', 'laws', 'be', 'breathed', 'by', 'every',
'American', 'mother', 'to', 'the', 'lisping', 'babe', 'that', 'prattles',
'on', 'her', 'lap.', 'Let', 'it', 'be', 'taught', 'in', 'schools,', 'in',
'seminaries,', 'and', 'in', 'colleges.', 'Let', 'it', 'be', 'written',
'in', 'primers,', 'spelling', 'books,', 'and', 'in', 'almanacs.', 'Let',
'it', 'be', 'preached', 'from', 'the', 'pulpit,', 'proclaimed', 'in',
'legislative', 'halls,', 'and', 'enforced', 'in', 'the', 'courts', 'of',
'justice.', 'And,', 'in', 'short,', 'let', 'it', 'become', 'the',
'political', 'religion', 'of', 'the', 'nation.']
>>>
```

White space ရှိတဲ့နေရာတွေကိုပိုင်းပေးသွားပါတယ်။ semicolon ";" နဲ့ပိုင်းကြည့်ရအောင်။

```
>>> line = "James;Miller;teacher;Perl"
>>> line.split(";")
['James', 'Miller', 'teacher', 'Perl']
```

နောက် optional parameter က maxsplit ပါ။

Maxsplit ထည့်ပေးရင် result မှာ "maxsplit+1" element ပါပါမယ်။

```
>>> mammon = "The god of the world's leading religion. The chief temple is
in the holy city of New York."
>>> mammon.split(" ",3)
['The', 'god', 'of', "the world's leading religion. The chief temple is in
the holy city of New York."]
```

Blank ကို delimiter string အနေနဲ့သုံးခဲ့တာတွေ့မှာပါ။ အတွဲလိုက်ရှိတဲ့ whitespace တွေဆိုရင် split() က blank တစ်ခုပြီးတိုင်း တစ်ခါပိုင်းမှာပါ။

```
>>> mammon = "The god  \t of the world's leading religion. The chief temple
is in the holy city of New York."
>>> mammon.split(" ",5)
['The', 'god', '', '\t', 'of', "the world's leading religion. The chief
temple is in the holy city of New York."]
>>>
```

Empty string တွေမပါချင်ရင် first argument ကို None ထည့်ပေးရပါမယ်။

```
>>> mammon.split(None,5)
['The', 'god', 'of', 'the', "world's", 'leading religion. The chief temple
is in the holy city of New York.']
```

## Regular Expression Split

String split() Method ကတော်တော်များများမှာ အဆင်ပြေပါတယ်။ ပိုမိုရှုပ်ထွေးတဲ့ အလုပ်တွေ အတွက်တော့ re module ထဲက split ကိုသုံးရပါမယ်။

```
>>> import re
>>> metamorphoses = "OF bodies chang'd to various forms, I sing: Ye Gods,
from whom these miracles did spring, Inspire my numbers with coelestial
heat;"
>>> re.split("\W+",metamorphoses)
['OF', 'bodies', 'chang', 'd', 'to', 'various', 'forms', 'I', 'sing', 'Ye',
'Gods', 'from', 'whom', 'these', 'miracles', 'did', 'spring', 'Inspire',
'my', 'numbers', 'with', 'coelestial', 'heat', '']
```

## Example

```
>>> import re
>>> lines = ["surname: Obama, prename: Barack, profession: president",
"surname: Merkel, prename: Angela, profession: chancellor"]
>>> for line in lines:
...     re.split(",* *\w*: ", line)
...
['', 'Obama', 'Barack', 'president']
['', 'Merkel', 'Angela', 'chancellor']
>>>
```

First element မှာ empty string မပါချင်ရင် slice operator သုံးနိုင်ပါတယ်။

```
>>> import re
>>> lines = ["surname: Obama, prename: Barack, profession: president",
"surname: Merkel, prename: Angela, profession: chancellor"]
>>> for line in lines:
...     re.split(",* *\w*: ", line)[1:]
...
['Obama', 'Barack', 'president']
['Merkel', 'Angela', 'chancellor']
>>>
```

## Search and Replace with sub

```
re.sub(regex, replacement, subject)
```

ဒါဆိုရင် regular expression နဲ့တိုက်တာတွေကို အစားထိုးပေးပါလိမ့်မယ်။

Example:

```
>>> import re
>>> str = "yes I said yes I will Yes."
>>> res = re.sub("[yY]es","no", str)
>>> print res
no I said no I will no.
```

# Chapter 30

Lambda, filter, reduce and map

## Lambda Operator

Lambada operator or lambda function ဆိုတာကတော့ anonymous functions လေးတွေ လုပ်တာဖြစ်ပါတယ်။ နာမည်မရှိတဲ့ function ပေါ့။ သုံးမယ်နေရာ မှာ တစ်ခါထည်း ဖန်းတီးတာ့ ဖြ ့ဖြစ်တဲ့အတွက် throw-away function လို့လည်းခေါ်ပါတယ်။ lambda function တွေကို filter(), map(), reduce() တွေနဲ့တွဲသုံးလေ့ရှိပါတယ်။ lisp programmer တွေတောင်းဆိုတာ ကြောင့် lambda feature ကိုထည့်ပေးခဲ့တာဖြစ်ပါတယ်။

Lambda function ရေးတဲ့ပုံစံကတော့ရိုးရှင်းပါတယ်။

lambda argument_list: expression

arugment list ထဲမှာ comma ခြားထားတဲ့ list of argument တွေပါပါတယ်။ expression ကတော့ ၎င်း argument တွေကိုသုံးမယ့် arithematic expression ဖြစ်ပါတယ်။ function ကိုနာမည်ပေး ချင်ရင်လည်း variable တစ်ခုထဲကို assign လုပ်နိုင်ပါတယ်။

အောက်ပါ ဥပမာ မှာ argument နှစ်ခုပေါင်းရလဒ်ကို ထုတ်ပေးပါတယ်။

```
>>> f = lambda x, y : x + y
>>> f(1,1)
2
```

## The map() Function

Lambda operator ရဲ့အသုံးဝင်မှုကို map() function နဲ့တွဲသုံးတဲ့အခါမြင်နိုင်ပါတယ်။

Map() ဆိုတာကတော့ argument နှစ်ခုပါတဲ့ function ဖြစ်ပါတယ်။

```
r = map(func, seq)
```

func ဆိုတဲ့ ပထမ argument ကတော့ function name တစ်ခုခု ဖြစ်ပြီးတော့ ဒုတိယ argument ကတော့ sequence တစ်ခု(eg.list)ဖြစ်ပါတယ်။ map() က sequence seq ထဲက element တွေအားလုံးသုံးပြီးတော့ fynction func အတိုင်းတွက်ချက်ပေးတာဖြစ်ပါတယ်။ ရလဒ်အနေ နဲ့ list အသစ်တစ်ခု ထုတ်ပေးပါတယ်။

```
def fahrenheit(T):
    return ((float(9)/5)*T + 32)
def celsius(T):
    return (float(5)/9)*(T-32)
temp = (36.5, 37, 37.5,39)

F = map(fahrenheit, temp)
C = map(celsius, F)
```

အထက်ဖော်ပြပါ ဥပမာမှာ lambda မသုံးထားပါဘူး။ lambda သုံးရင် fahrenheit() celsius() စတဲ့ function တွေကို သတ်မှတ် နာမည်ပေးစရာမလိုပါဘူး။

အောပ်ပါ interactive session မှာတွေ့မြင်နိုင်ပါတယ်။

```
>>> Celsius = [39.2, 36.5, 37.3, 37.8]
>>> Fahrenheit = map(lambda x: (float(9)/5)*x + 32, Celsius)
>>> print Fahrenheit
[102.56, 97.700000000000003, 99.140000000000001, 100.03999999999999]
>>> C = map(lambda x: (float(5)/9)*(x-32), Fahrenheit)
>>> print C
[39.200000000000003, 36.5, 37.300000000000004, 37.799999999999997]
>>>
```

Map() က List တစ်ခုထက်လည်း ပိုအလုပ်လုပ်နိုင်ပါတ်။ list တွေရဲ့ length တော့ တူရပါမယ်။ map() က ပထမ list ရဲ့ $0^{th}$ element ၊ ဒုတိယ list ရဲ့ $0^{th}$ element စတဲ့ပုံစံနဲ့ ရှိတဲ့ length အတိုင်း အလုပ်လုပ်ပေးပါမယ်။

```
>>> a = [1,2,3,4]
>>> b = [17,12,11,10]
>>> c = [-1,-4,5,9]
>>> map(lambda x,y:x+y, a,b)
[18, 14, 14, 14]
>>> map(lambda x,y,z:x+y+z, a,b,c)
[17, 10, 19, 23]
>>> map(lambda x,y,z:x+y-z, a,b,c)
[19, 18, 9, 5]
```

အထက်ဖော်ပြပါ ဥပမာမှာ parameter x က သူ့အတွက် တန်ဖိုးတွေကို list a ကနေရရှိပါတယ်။ y ကတော့ list b ကနေဖြစ်ပြီး ၊ z ကတော့ lit c ကနေဖြစ်ပါတယ်။

## Filtering

Filter(function,list) function ့တော့ list ထဲက element တွေ filter(စစ်ထုတ်)လုပ်ရာမှာ သုံးပါတယ်။ first argument ကတော့ function တစ်ခုလိုပါတယ်.ၚင်း function က Boolean value ထုတ်ပေးရမှာဖြစ်ပါတယ်။ ၚင်း function ကို list ထဲမှာ ရှိတဲ့ element တိုင်းအတွက် တွက်ချက်ပေးမှဖြစ်ပြီးတော့ true ဖြစ်တဲ့ element တွေကို List အနေနဲ့ ထုတ်ပေးပါတယ်။

```
>>> fib = [0,1,1,2,3,5,8,13,21,34,55]
>>> result = filter(lambda x: x % 2, fib)
>>> print result
[1, 1, 3, 5, 13, 21, 55]
>>> result = filter(lambda x: x % 2 == 0, fib)
>>> print result
[0, 2, 8, 34]
>>>
```

## Reducing a List

Reduce(func,seq) ကတော့ seq ထဲက element တွေကိုတောက်လျှောက် တွက်ချက်ပေးပြီး value တစ်ခုပဲ ထုတ်ပေးပါတယ်။ ဥပမာ list ထဲက element တွေအားလုံးပေါင်းလဒ်လိုမျိုး

Seq=[s1,s2,s3,s4,s5,…..,sn] ကိုသုံးပြီး reduce(func,seq)ကိုခေါ်မယ်ဆိုရင်

- ပထမဆုံး element နှစ်ခုကိုအရင်တွက်မယ် func(s1,s2)
- နောက်တစ်ဆင့်မှာ ခုနရလာတဲ့ result နဲ့ တတိယ element နဲ့ ထပ်ပြီးတွက်မယ် func(func(s1,s2),s3) ဒီလိုပုံစံဖြစ်သွားပါတယ်။
- နောက်ဆုံးအကုန်ကုန်သွားတဲ့အထိလုပ်ပြီး value တစ်ခုပြန်ထုတ်ပေးပါတယ်။

Example

```
>>> reduce(lambda x,y: x+y, [47,11,42,13])
113
```

Examples of reduce()

Determining the maximum of a list of numerical values by using reduce:

```
>>> f = lambda a,b: a if (a > b) else b
>>> reduce(f, [47,11,42,102,13])
102
>>>
```

Calculating the sum of the numbers from 1 to 100:

```
>>> reduce(lambda x, y: x+y, range(1,101))
5050
```

# Chapter 31

## List Comprehension

### Introduction

List compression ဆိုတာကတော့ python မှာ List တွေဖန်တီးဖို့အတွက် သပ်ရပ်လှပတဲ့ နည်းလမ်းတစ်ခုဖြစ်ပါတယ်။

List compression ဟာ map() filter() redue() တွေအစားသုံးနိုင်ပါတယ်။

### Examples

Lambda အခန်းမှာ Celsius <=>fahrenheit လေးရေးခဲ့တယ်။ list compression နဲ့ဆိုရင် အောက်ပါပုံစံအတိုင်းရမယ်။

```
>>> Celsius = [39.2, 36.5, 37.3, 37.8]
>>> Fahrenheit = [ ((float(9)/5)*x + 32) for x in Celsius ]
>>> print Fahrenheit
[102.56, 97.700000000000003, 99.140000000000001, 100.03999999999999]
>>>
```

### Pythagorean triples အတွက်ဆိုရင်

```
>>> [(x,y,z) for x in range(1,30) for y in range(x,30) for z in range(y,30)
if x**2 + y**2 == z**2]
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (7, 24, 25), (8, 15, 17), (9, 12, 15),
(10, 24, 26), (12, 16, 20), (15, 20, 25), (20, 21, 29)]
>>>
```

Cross product of two sets:

```
>>> colours = [ "red", "green", "yellow", "blue" ]
>>> things = [ "house", "car", "tree" ]
>>> coloured_things = [ (x,y) for x in colours for y in things ]
>>> print coloured_things
[('red', 'house'), ('red', 'car'), ('red', 'tree'), ('green', 'house'),
('green', 'car'), ('green', 'tree'), ('yellow', 'house'), ('yellow',
'car'), ('yellow', 'tree'), ('blue', 'house'), ('blue', 'car'), ('blue',
'tree')]
>>>
```

Generator Comprehension

Python2.6 မှာ generator compression စတင်ပါဝင်လာတယ်။ သူကတော့ () နဲ့သုံးတဲ့ generator expression ဖြစ်ပါတယ်.။ သူရဲ့အလုပ်လုပ်ပုံက list compression နဲ့တူတယ် ဒါပေမယ့် သူက list အစား ːgenerator ကို ထုတ်ပေးတယ်။

```
>>> x = (x **2 for x in range(20))
>>> print(x)
 at 0xb7307aa4>
>>> x = list(x)
>>> print(x)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256,
289, 324, 361]
```

A more Demanding Example

sieve of Eratosthenes ကိုသုံးပြီး 1 နဲ့ 100 ကြား prime number တွေရှာမယ်။

```
>>> noprimes = [j for i in range(2, 8) for j in range(i*2, 100, i)]
>>> primes = [x for x in range(2, 100) if x not in noprimes]
>>> print primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
71, 73, 79, 83, 89, 97]
>>>
```

ဒါကိုမှပိုပြီး general ကျမယ့်ပုံစံပြောင်းမယ်။ n လောက်ရှိတဲ့ prime number တွေရှာမယ်ပေါ့။

```
>>> from math import sqrt
>>> n = 100
>>> sqrt_n = int(sqrt(n))
>>> no_primes = [j for i in range(2,sqrt_n) for j in range(i*2, n, i)]
```

no_primes ကိုကြည့်ရင် ပြဿနာတစ်ခုရှိတာတွေ့ပါမယ်။ list ထဲမှာ နှစ်ခုထပ်နေတာတွေတွေ့ပါလိမ့်မယ်။

```
>>> no_primes
[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40,
42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78,
80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 6, 9, 12, 15, 18, 21, 24, 27, 30,
33, 36, 39, 42, 45, 48, 51, 54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87,
90, 93, 96, 99, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64,
68, 72, 76, 80, 84, 88, 92, 96, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60,
65, 70, 75, 80, 85, 90, 95, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78,
84, 90, 96, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 16, 24, 32,
40, 48, 56, 64, 72, 80, 88, 96, 18, 27, 36, 45, 54, 63, 72, 81, 90, 99]
>>>
```

ဒါကိုဖြေရှင်းဖို့အတွက် နောက်အခန်းမှာပါမယ့် set compression နားလည်ရပါမယ်။

## Set Comprehension

Set comprehension က list comprehension နဲ့တူပါတယ်။ သူကတော့ set တစ်ခု return ပြန်ပေးပါတယ်။ set လုပ်ဖို့ {} တွေသုံးပါတယ်။ ခုန ပြဿနာကို ထပ်တာတွေမပါချင်ရင် set comprehension သုံးရပါမယ်။

```
>>> from math import sqrt
>>> n = 100
>>> sqrt_n = int(sqrt(n))
>>> no_primes = {j for i in range(2,sqrt_n) for j in range(i*2, n, i)}
>>> no_primes
{4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24, 25, 26, 27, 28, 30,
32, 33, 34, 35, 36, 38, 39, 40, 42, 44, 45, 46, 48, 49, 50, 51, 52, 54, 55,
56, 57, 58, 60, 62, 63, 64, 65, 66, 68, 69, 70, 72, 74, 75, 76, 77, 78, 80,
81, 82, 84, 85, 86, 87, 88, 90, 91, 92, 93, 94, 95, 96, 98, 99}
>>> primes = {i for i in range(n) if i not in no_primes}
>>> print(primes)
{0, 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97}
>>>
```

## Recursive Function to Calculate the Primes

အောက်ပါ python scriptကတော့ prime numbers တွေကို recursive function သုံးပြီးတွက် ပေးတာဖြစ်ပါတယ်။

```
from math import sqrt
def primes(n):
    if n == 0:
        return []
    elif n == 1:
        return []
    else:
        p = primes(int(sqrt(n)))
        no_p = {j for i in p for j in xrange(i*2, n+1, i)}
        p = {x for x in xrange(2, n + 1) if x not in no_p}
    return p

for i in range(1,50):
    print i, primes(i)
```

# Chapter 32

## Generators

### Introduction

Generator တွေဆိုရာကတော့ iterator တွေကိုထုတ်ပေးတဲ့အလွယ်ကူဆုံးနည်းလမ်းတွေဖြစ်ပါတယ်။ function တွေနဲ့ပုံစံတူပေမဲ့ syntactic အနေနဲ့ရော semantic အနေနဲ့ရော ကွဲပြားပါတယ်။ generator တွေမှာ return အစား yield ဆိုတာတွေရပါလိမ့်မယ်။
နောက်အရေးကြီးတာတစ်ခုကတော့ local variable တွေ execution တွေကို call တွေကြားမှာ အလိုအလျောက်လုပ် ဆောင်ပေးပါတယ်။ ဒီလိုလုပ်ရတဲ့အကြောင်းကတော့ ပုံမှန် function call တွေလို generator function တွေက function အစမှာ execution မလုပ်လို့ဖြစ်ပါတယ်။ အဲလို့အစကနေ execute လုပ်မယ့်အစား generator မှာ နောက်ဆုံး call ကထွက်သွားတဲ့ yield statement မှာဆက်လုပ်ပါတယ်။ နောက်တစ်နည်းပြောရရင် python interpreter က yield statement တစ်ခုကို generator ကနေထုတ်ပေးတဲ့iterator တစ်ခုထဲမှာတွေရင် ၎င်း နေရာနဲ့ local variable တွေကိုမှတ်ပြီးတော့ iteratorကနေ return (ပြန်ထွက်)ပါတယ်။ နောက်တစ်ကြိမ် iterator ကိုခေါ်တဲ့အခါမှာ ခုနမှတ်ထားတဲ့ yield statement ကနေပြန်စပါတယ်။ generator ထဲမှာ yield statement တွေတစ်ခုထက်ပိုနိုင်သလို loop ထဲမှာလည်း ရှိနေနိုင်ပါတယ်။ generator မှာ return statement ပါရင် python interpreter က၎င်း codeကို execute လုပ်တဲ့အခါ မှာ StopIteration exception error ပြပြီး ရပ်သွားပါလိမ့်မယ်။
ခုပြောသမျှတွေကို class based iterator နဲ့လည်းအသုံးပြုနိုင်ပါတယ်။ ဒါပေမယ့် generator ရဲ့အဓိက အားသာချက်က __iter__() နဲ့ next() method တောကို အလိုအလျောက်ပြုလုပ်ပေးခြင်း တွေပါပါတယ်။ အရမ်းကြီးတဲ့အဆုံးသတ်မရှိတဲ့ dataတွေထုတ်ပေးဖို့ရာ ကောင်းမွန်တဲ့ နည်းလမ်း တွေ generator မှာပါပါတယ်။မြို့နာမည်လေးခုထုတ်ပေးတဲ့ generator တစ်ခု example ပြထားပါတယ်။

```
def city_generator():
    yield("Konstanz")
    yield("Zurich")
    yield("Schaffhausen")
    yield("Stuttgart")
```

၎င်းgenerator ကိုသုံးပြီး iterator လုပ်နိုင်ပါတယ်။

```
>>> from city_generator import city_generator
>>> x = city_generator()
>>> print x.next()
Konstanz
>>> print x.next()
Zurich
>>> print x.next()
Schaffhausen
>>> print x.next()
Stuttgart
>>> print x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
StopIteration
>>>
```

အထက်မှပြထာသလိုပဲ iterator x ကို interactive shell မှာလုပ်ခဲ့တယ်။ next() ကိုခေါ်တိုင်း နောက်မြို့တစ်ခုထုတ်ပေးတယ်။နောက်ဆုံးမြို့ကို ပေးပြီး x.next() ကိုခေါ်ရင် "StopIteration". error ပြမယ်။

အစကနေပြန်စအောင်လို့ reset လည်းလုပ်လို့မရပါဘူး။ ဒါပေမယ့် generator နောက်တစ်ခုဖန်တီး နိုင်ပါတယ်။ x=city_generator() ကိုထပ်လုပ်တာဖြစ်ပါတယ်။ yield က return နဲ့တူတယ် ထင်ရပေမဲ့ ဒီ ဥပမာ နဲ့ဆိုရင် မတူတာတွေ့ပါလိမ့်မယ်။ yield အစား return ကိုထည့်မယ်ဆိုရင် function ပဲဖြစ်ပါလိမ့်မယ်.။ ဒါပေမယ့် function ဆိုရင် အမြဲတမ်း "Konstanz" ပဲ return ပြန်ပေး နေပါလိမ့်မယ်။

Method of Operation

Generator တွေကို interator တွေထုတ်ပေးတဲ့(generate) အတွက်ကြောင့် generator လို့ခေါ်တာဖြစ်ပါတယ်။

အလုပ်လုပ်ပုံကတော့

- Generator ကို function တွေလိုပဲ call လုပ်ပါတယ်။ သူက return ပြန်ပေးတာကတော့ iterator object ဖြစ်ပါတယ်။ ဒီအထိတော့ generator code တွေအလုပ်မလုပ်သေးပါ
- Iterator တွေကိုတော့ next method နဲ့ခေါ်ပါတယ်။ ပထမဆုံး မှာတော့ function လိုအလုပ်လုပ်ပါတယ်။yield statement ကိုမတွေ့မချင်းဖြစ်ပါတယ်။
- Yield ကတော့ expression တစ်ခုရဲ့ value ကို keyword yield နဲ့return ပြန်ပေးပါတယ်။ ဒါကတော့ function နဲ့တူပါတယ်.ဒါပေမယ့် pythonက yield ရဲ့ position တွေ့ local variable တွေကို next call အတွက် မှတ်ထားပေးပါတယ်။ နောက်တစ်ခါ call လုပ်တဲ့ အခါ yield statement နောက်ကနေပြီးဆက်ပြီး execution ဆက်လုပ်ပါတယ်။
- နောက်ဆုံးထိအကုန်ပြီးသွားတယ် သို့ value မပါတဲ့ return တစ်ခုတွေ့ရင် iterator ပြီးဆုံး ပြီဖြစ်ပါတယ်။

Fibonacci ထုတ်ပေးတဲ့ example လေးကြည့်ရအောင်။

Example seauence 0,1,1,2,3,5,8,13

Mathematical term နဲ့ဆိုရင်

$F_n = F_{n-1} + F_{n-2}$

$F_0 = 0$ and $F_1 = 1$

Python implementation

```
def fibonacci(n):
    """Fibonacci numbers generator, first n"""
    a, b, counter = 0, 1, 0
    while True:
        if (counter > n): return
        yield a
        a, b = b, a + b
        counter += 1
f = fibonacci(5)
for x in f:
    print x,
print
```

အထက်ဖော်ပြပါ generator ကို Fibonacci အတွက်သုံးနိုင်ပါတယ်။

Endless iterator example တစ်ခုထပ်ပြောပါမယ်။

```
def fibonacci():
    """Fibonacci numbers generator"""
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

f = fibonacci()

counter = 0
for x in f:
    print x,
    counter += 1
    if (counter > 10): break
print
```

Recursive Generators

Function တွေလိုပဲ generator တွေကို recusice ရေးနိုင်ပါတယ်။အောက်မှာ list တွေကို permutation လုပ်ထားပါတယ်။

Permutation ဆိုတာကတော့ ပေးထားတဲ့ အစီအစဉ် အမျိုးမျိုးစဉ်တာဖြစ်ပါတယ်။

a,b,c ကို permutation လုပ်ကြည့်ပါမယ်။

```
a b c
a c b
b a c
b c a
c a b
c b a
```

n elemetnt အတွက် number of permutation ကတော့ n! ဖြစ်ပါတယ်။

```
n! = n*(n-1)*(n-2) ... 2 * 1


def permutations(items):
    n = len(items)
    if n==0: yield []
    else:
        for i in range(len(items)):
            for cc in permutations(items[:i]+items[i+1:]):
                yield [items[i]]+cc

for p in permutations(['r','e','d']): print ''.join(p)
for p in permutations(list("game")): print ''.join(p)
```

## A Generator of Generators

Fibonacci sequence exampleရဲ့ ဒုတိယgenerator မှာ iterator တစ်ခုထုတ်ပေးပါတယ်။ theory အရဆိုရင်တော့ ဒါကိုသုံးပြီး finbonacci number အားလုံးကိုထုတ်ပေးနိုင်ပါတယ်။

```
list(fibonacci())
```

ဒါဆိုရင် ကိုယ့် computer မြန်သလောက်ရှိတဲ့ sequence တွေပြပါလိမ့်မယ်။

လက်တွေ့မှာ တော့ အဆုံးမရှိတဲ့ iterator ထဲကမှ n element ပဲလိုပါတယ်။ ဒါဆိုရင် နောက် generator တစ်မျိုးသုံးနိုင်ပါတယ်။ generator g အတွက် n element တွေဆိုရင်

```
def firstn(g, n):
        for i in range(n):
                yield g.next()
```

အောပ်ပါ script ကနေ Fibonacci sequence ရဲ့ပထမ 10 လုံးထုတ်ပေးပါတယ်။

```
#!/usr/bin/env python
def fibonacci():
    """Ein Fibonacci-Zahlen-Generator"""
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

def firstn(g, n):
        for i in range(n):
                yield g.next()

print list(firstn(fibonacci(), 10))
```

# Chapter 33

## Exception Handling

Exeception ဆိုတာကတော့ program တစ်ခုကို execute လုပ်တဲ့အခါမှာ ဖြစ်ပေါ်လာတဲ့ error တွေဖြစ်ပါတယ်။ exeception handling ဆိုတာကတော့ error တွေဖြစ်ပေါ်လာလျှင် အလိုအလျောက် ကိုင်တွယ်ဖြေရှင်းတာကိုဆိုလိုပါတယ်။

Error handling ကို error ဖြစ်ပေါ်တဲ့အခါမှာရှိတဲ့ execution state တွေကို မှတ်သားပြီးတော့ သူ့အတွက် ရေးသားထားတဲ့ code တွေ(exeception handler) နဲ့ဖြေရှင်းပါတယ်။ ဖြစ်ပေါ်လာ တဲ့ error ပေါ်မူတည်ပြီးတော့ error handler ကပြင်ဆင်ပေးနိုင်သလို သိမ်းထားတဲ့ data တွေနဲ့ ဆက်လက်လုပ်ဆောင်နိုင်ပါတယ်။

## Exception Handling in Python

Execption handling မှာ python က java နဲ့တူပါတယ်။ execption ဖြစ်နိုင်တဲ့ code ကို try block ထဲမှာထားပါတယ်။ java မှာတော့ exeception တွေကို catch နဲ့သုံးပေမယ့် python မှာတော့ except နဲ့သုံးပါတယ်။raise statement နဲ့လည်း ကိုယ်ပိုင် execption တွေလုပ်နိုင်ပါသေးတယ်။

Example- user ကို integer number ထည့်ခိုင်းမယ်။ raw_input() ကိုသုံးရင်input က string ဖြစ်မယ် ဒါကြောင်း Integer ဖြစ်အောင် cast လုပ်ပေးရမယ်။ input က integer မဟုတ်ရင် ValueError တက်မယ်.။

```
>>> n = int(raw_input("Please enter a number: "))
Please enter a number: 23.5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23.5'
```

ဒါကို execption handling ထည့်ရေးရင်

```
while True:
    try:
        n = raw_input("Please enter an integer: ")
        n = int(n)
        break
    except ValueError:
        print("No valid integer! Please try again ...")
print "Great, you successfully entered an integer!"
```

ဒါဆိုရင် integer ပေးမှပဲ loop ကို break လုပ်မယ်။

Sample output

```
$ python integer_read.py
Please enter an integer: abc
No valid integer! Please try again ...
Please enter an integer: 42.0
```

```
No valid integer! Please try again ...
Please enter an integer: 42
Great, you successfully entered an integer!
$
```

Multiple Except Clauses

အမျိုးအမျိုးသော execption တွေအတွက် try statement မှာ except clause တစ်ခုထက်ပိုနိုင်ပါတယ်။ အများဆုံးတော့ တစ်ခုပဲ execution ဖြစ်နိုင်ပါတယ်။

နောက် ဥပမာတစ်ခုအနေနဲ့ file read လုပ်ဖို့ open မယ်။file ထဲက Line တစ်ကြောင်း read ပြီးတော့ အဲ့ဒီ line ကို Integer ပြောင်းမယ်။အနည်းဆုံး execption နှစ်ခုဖြစ်နိုင်တယ်။

- an IOError
- ValueError

တစ်ခြား မသတ်မှတ်နိုင်တဲ့ execption တွေလည်းရှိမယ်။

```
import sys

try:
    f = open('integers.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as (errno, strerror):
    print "I/O error({0}): {1}".format(errno, strerror)
except ValueError:
    print "No valid integer in line."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

IOError ကို handling လုပ်တာကြည့်ရမှာဖြစ်ပါတယ်။ IOError က tuple အနေနဲ့ error number တွေ error message string တွေကိုထုတ်ပေးပါတယ်။ ဒါတွေကို variable errno နဲ့ strerror ထဲမှာ "except IOError as (errno, strerror)" လို့ assign လုပ်ခဲ့ပါတယ်။ မရှိတဲ့ file တစ်ခုကို အထက်ပါ script သုံးပြီး ဖွင့်ဖို့ကြိုးစားရင် အောက်ပါ message ရပါမယ်။

```
I/O error(2): No such file or directory
```

Integers.txt က readable မဖြစ်ရင် read လုပ်ဖို့ permission မရှိရင်တော့

```
I/O error(13): Permission denied
```

Except clause တစ်ခုကို execption တစ်ခုထက်ပိုပြီးပေးချင်ရင် tuple တွေနဲ့ နာမည်ပေးနိုင်ပါတယ်။

```
try:
    f = open('integers.txt')
    s = f.readline()
    i = int(s.strip())
except (IOError, ValueError):
    print "An I/O error or a ValueError occurred"
except:
    print "An unexpected error occurred"
    raise
```

## Clean-up Actions (try ... finally)

Try နဲ့ except ကို အတွဲလိုက်ပဲတွေခဲ့မှာဖြစ်ပါတယ်။နောက်တစ်မျိုးသုံးနိုင်ပါသေးတယ်။ try နောက်မှာ finally clause ပါနိုင်ပါတယ်။ clean-up or termination clauses လို့လည်း ခေ ခေါ်ပါတယ်။ ဘယ်အခြေအနေမှာပဲဖြစ်ဖြစ် execution လုပ်တာကြောင့်ဖြစ်ပါတယ်။ example

```
try:
    x = float(raw_input("Your number: "))
    inverse = 1.0 / x
finally:
    print("There may or may not have been an exception.")
print "The inverse: ", inverse
```

sample output

```
bernd@venus:~/tmp$ python finally.py
Your number: 34
There may or may not have been an exception.
The inverse:  0.0294117647059
bernd@venus:~/tmp$ python finally.py
Your number: Python
There may or may not have been an exception.
Traceback (most recent call last):
  File "finally.py", line 3, in <module>
    x = float(raw_input("Your number: "))
ValueError: invalid literal for float(): Python
bernd@venus:~/tmp$
```

## Combining try, except and finally

"finally" နဲ့ "except"ကို အတူသုံးနိုင်ပါတယ်။

```
try:
    x = float(raw_input("Your number: "))
    inverse = 1.0 / x
except ValueError:
    print "You should have given either an int or a float"
except ZeroDivisionError:
    print "Infinity"
finally:
    print("There may or may not have been an exception.")
```

sample output

```
bernd@venus:~/tmp$ python finally2.py
Your number: 37
There may or may not have been an exception.
bernd@venus:~/tmp$ python finally2.py
Your number: seven
You should have given either an int or a float
There may or may not have been an exception.
bernd@venus:~/tmp$ python finally2.py
Your number: 0
Infinity
There may or may not have been an exception.
bernd@venus:~/tmp$
```

else Clause

tr…except clause မှာ else clause လည်းပါနိုင်ပါတယ်။ else ကို except တွေအားလုံးနောက်မှာ ထားရပါမယ်။ try မှာ error မတက်ရင် else အလုပ်လုပ်ပေးပါမယ်။

example

```
import sys
file_name = sys.argv[1]
text = []
try:
    fh = open(file_name, 'r')
    text = fh.readlines()
    fh.close()
except IOError:
    print 'cannot open', file_name

if text:
    print text[100]
```

ဒီဥပမာ မှာ commamd line argument ကနေ file name ကိုလက်ခံပေးပါတယ်။ "exception_test.py" လို့သိမ်းထားတယ်ဆိုပါဆို့.။ဒါဆိုရင်

```
python exception_test.py integers.txt
```

ဒီလိုမလုပ်ချင်ရင် "file_name = sys.argv[1]" to "file_name = 'integers.txt'" လို့ပြောင်းလိုက် ပါ။

```
import sys
file_name = sys.argv[1]
text = []
try:
    fh = open(file_name, 'r')
except IOError:
    print 'cannot open', file_name
else:
    text = fh.readlines()
    fh.close()

if text:
    print text[100]
```

မတူတာကတော့ first case မှာဖြစ်ပါတယ်။

The assert Statement

Assert Statement ကတော့ debugging statement တွေအတွက်ဖြစ်ပါတယ်။ သတ်မှတ်ထားတဲ့ အခြေအနေတစ်ခုမမှန်မှသာလျှင် assert အတွက် exception ဖြစ်ပါတယ်။

Assert statement မသုံးပဲ python မှာအောက်ပါအတိုင်းအသုံးပြုနိုင်ပါတယ်။

```
if not <some_test>:
        raise AssertionError(<message>)
```

အောက်မှာတော့ assert statement ကိုသုံးထားပါတယ်။

```
assert <some_test>, <message>
```

အထက်ဖော်ပြပါစာကြောင်းကို <some_test> တွက်ချက်တာတွေ မှားမယ်ဆိုရင် exeception ဖြစ်ပြီးတော့ <message>ကို output ထုတ်ပေးပါမယ်။

Example:

```
>>> x = 5
>>> y = 3
>>> assert x < y, "x has to be smaller than y"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: x has to be smaller than y
>>>
```

# Chapter 34

Classes

Object Oriented Programming

ရှေ့အပိုင်းတွေမှာ Object oriented programming(OOP) အကြောင်းတွေကိုတမင်ချန်လှပ်ထားခဲ့ပါတယ်။ OOP အကြောင်းမပြောခဲ့တာ ပိုလွယ်မှာဖြစ်ပါတယ်။ ဒါပေမယ့် classes တွေရဲ့ object တွေ method တွေကိုသုံးခဲ့ကြပါတယ်။

တစ်ချို့က oop ကို modern programming paradigm တစ်ခုလို့ထင်ပေမဲ့ ၁၉၆၀ လောက်ကစတယ် ပြောရပါမယ်။ ပထမဆုံး object သုံးတဲ့ language ကတော့ simula 67 ပါ။ OOP ကိုကောင်းတယ်ပြောတဲ့သူရှိခဲ့ သလို မကောင်းဘူးပြောတဲ့သူများလည်း ရှိခဲ့ပါတယ်။ OOP C++ ရဲ့ designer/implementer ဖြစ်တဲ့ Alexander StepanovကOOP provides a mathematically-limited viewpoint and called it "almost as much of a hoax as Artificial Intelligence".လို့ပြောခဲ့ပါတယ်။

Edsger W. Dijkstra, one of the most renown professors of Computer Science, wrote: "... what society overwhelmingly asks for is snake oil. Of course, the snake oil has the most impressive names - otherwise you would be selling nothing - like "Structured Analysis and Design", "Software Engineering", "Maturity Models", "Management Information Systems", "Integrated Project Support Environments" "Object Orientation" and "Business Process Re-engineering" (the latter three being known as IPSE, OO and BPR, respectively).

Object Oritented Projection ရဲ့ အဓိက principle 4 ခု

- Encapsulation
- Data Abstraction
- Polymorphism
- Inheritance

Object oritented program တစ်ခုကို class တွေပေါ်မှာအခြေခံထားပြီးတော့ သူနဲ့တွဲဆက်အလုပ်လုပ်နေတဲ့ object များပါရှိပါတယ်။ ပုံမှန် program တွေမှာတော့ function တွေ routine တွေနဲ့ပဲအလုပ်လုပ်ပါတယ်။ OOP မှာ object တွေဟာ တစ်ခုနဲ့တစ်ခု message များလက်ခံခြင်း။ data များတွက်ချက်ပေးခြင်း။ တစ်ခြား object များဆီသို့ message ပေးပို့ခြင်းများလုပ်ဆောင်ပေးကြပါတယ်။

Analogy: The Cake Class

Class ကိုပြောရရင် cake တစ်ခုလုပ်ဖို့လိုအပ်တဲ့ receipe နဲ့ယှဉ်ကြည့်နိုင်ပါတယ်။ cake တစ်ခုဖုတ်ဖို့အတွက် receipe လိုပါတယ်။ receipe(class) နဲ့ cake(an instance or an object of this

class) ကြားခြားနားချက်ကတော့ထင်ရှားပါတယ်။ cake ကိုလုပ်ပြီးရင်စားနိုင်ပေမယ့် receipe ကိုတော့ စားလို့မရပါဘူး။cake မုန့်ဖုတ်သလိုပါပဲ oop program တစ်ခုဟာ object တွေကို class definition မှာ သတ်မှတ်ထားတဲ့အတိုင်းတည်ဆောက်ပေးပါတယ်။ class ထဲမှာ variable တွေ method တွေပါဝင်ပါတယ်။ cake မုန့်ဖုတ်မယ်ဆိုရင် ပါဝင်ပစ္စည်း(ingredient) နဲ့ လုပ်ငန်းစဉ်(instruction) တွေလိုပါမယ်။ အဲ့ဒီလိုပဲ class မှာလည်း variable တွေ method တွေလိုပါတယ်။ ၎င်း တို့ကို class variable လို့ခေါ်ပြီးတော့ method တွေအားလုံးအတွက် တူညီတဲ့တန်ဖိုးတွေရှိပါတယ်.။instance variable ကတော့သီခြား object တစ်ခုချင်းစီအတွက် မတူညီတဲ့တန်ဖိုးတွေရှိပါတယ်။ data တွေသုံးစွဲနိုင်အောင်လည်း class ထဲမှာ လိုအပ်တဲ့ method တွေသတ်မှတ်ဖို့လည်းလိုပါတယ်။

Classes, Objects, Instances

Class က data type တစ်ခုကိုသတ်မှတ်ပေးပြီးတော့ ၎င်းထဲမှာ variable ,properties,methods တွေပါဝင်ပါတယ်။

Class တွေရဲ့ instance တွေ Object တွေရှိနိုင်ပါတယ်။ instance ဆိုတာကတော့ run-time မှာ ဖန်တီးလိုက်တဲ့ class ရဲ့ object ဖြစ်ပါတယ်။ object တွေရဲ့ attribute value တွေကိုတော့ state လို့ခေါ်ပါတယ်။ object မှာ class ထဲမှာသတ်မှတ်ထားတဲ့ state နဲ့ behavior ပါဝင်ပါတယ်။

အထက်မှာပြောခဲ့သလိုပဲ class တွေမှာလည်း attribute တွေ properties တွေ နဲ့ instance တွေအတွက် method တွေပါဝင်ပါတယ်။ method ဆိုတာကတော့ function ပါပဲ ဒါပေမယ့်သူက class ကပိုင်တာဖြစ်ပါတယ်။ class ထဲမှာသတ်မှတ်ထားတာဖြစ်ပြီး instance တွေ နဲ့ class data တွေအတွက်အလုပ်လုပ်ပါတယ်။ method တွေကို class ရဲ့ instance တွေ subclass တွေကသာ အသုံးပြုနိုင်ပါတယ်။

"Account" နဲ့ "Account Holder"ဆိုပြီး class နှစ်ခုရှိမယ်။ "Account Holder" ရဲ့ data မှာ Holder Surname and Prename, Adress, Profession, and Birthday စတာတွေပါပါမယ်။ method တွေကတော့ "Change of Residence" and "Change of Profession" တွေဖြစ်ပါတယ်။ ပြီးပြည့်စုံ တဲ့ model တော့မဟုတ်ပါဘူး။ data တွေ method တွေအများကြီးလိုပါဦးမယ်။

Encapsulation of Data

OOP ရဲ့ ကောင်းမွန်တဲ့တစ်ချက်က data encapsulation ဖြစ်ပါတယ်။ encapsulation/abstractopm/ data hiding အတူတူပဲဖြစ်ပါတယ်။ encapsulation ရဲ့သဘောကတော့ object component တွေကို ရယူအသုံးပြုခြင်းကို ကန့်သတ်ပေးတာပဲဖြစ်ပါတယ်။ ဒါကြောင့် object definition ရဲ့အပြင်ကနေကြည့်ရင် object ကိုဘယ်လိုလုပ်ထားတယ်ဆိုတာမသိနိုင်ပါဘူး။ ဒါတွေကို access ရချင်ရင်တော့ special

method တွေဖြစ်တဲ့ getter/setter တွေသုံးရပါမယ်။ get() set() method တွေသုံးပြီး တော့ internal data တွေကို မတော်တဆပြောင်းလဲမိတာတွေကို ရှောင်ရှားနိုင်ပါတယ်။

ဒီလိုကာကွယ်ထားတာတွေကို ရှောင်ရှားဖို့လည်းဖြစ်နိုင်ပါတယ်။ c++ မှာ "friends" , java နဲ့ ruby မှာ reflection api ,python မှာ တော့ mangling ကိုသုံးနိုင်ပါတယ်။

Private data တွေသတ်မှတ်ဖို့သုံးတဲ့ method တွေကို စစ်ဆေးတာတွေအတွက်လည်း သုံးနိုင်ပါတယ်။ eg မှာဆို birthday ဟုတ်မဟုတ်ကိုစစ်နိုင်ပါတယ်။ အသက် ၁၈၀ ကျော်တယ်ဆိုတာ မျိုးခဲယဉ်းပါတယ်။ General Interbank Recurring Order (GIRO) နဲ့ customer တစ်ယောက်ဟာ အသက် ၁၄ နှစ်ဆိုတာလည်းမဖြစ်နိုင်ပါဘူး။

Inheritance

Class တစ်ခုက အခြား class တွေကို inherit လုပ်နိုင်ပါတယ်။ super-class တစ်ခုဆီကနေ class တစ်ခု attribute တွေ behavior တွေ inherit လုပ်နိုင်ပါတယ်။ super-class ကနေ inherit လုပ်တဲ့ class ကိုကျတော့ sub-class လို့ခေါ်ပါတယ်။ super-class ကို anscestor လို့လည်းခေါ်ပါသေးတယ်။

Class account ရဲ့ model ကိုကြည့်မယ်ဆိုရင် တကယ့် bank တွေအတွက် တော်တော်လေးကိုက်ညီ တာတွေ့ပါလိမ့်မယ်။ Bank တွေမှာ account အမျိုးမျိုးရှိပါတယ်။ eg. e.g. Savings Accounts, Giro Accounts and others။ အမျိုးအစားတွေမတူပေမဲ့လည်း ဘုံတူတဲ့ properties တွေ method တွေရှိကြပါတယ်။ဥပမာ account number,holder,balance။ ဒါအပြင် အားလုံးက deposit /withdraw တွေလုပ်နိုင်ပါတယ်။

ဒါကြောင့်အားလုံးအတွက်အခြေခံ "fundamental" account ရှိပြီးတော့ အာလုံးကသူ့ဆီကနေ inherit လုပ်ပါတယ်။ inheritance ကတော့ ရှိပြီးသား class ကိုအသုံးပြုပြီးတော့ class အသစ်တွေ တည်ဆောက်တာဖြစ်ပါတယ်။ အသစ်လုပ်လိုက်တဲ့ class ဟာ ရှိပြီးသားကို ထပ်ထည့်တာတွေ ဖြစ်နိုင်သလို ရှိတာတွေကိုမှ ကန့်သတ်တာတွေလည်းဖြစ်နိုင်ပါတယ်။

Python မှာ ဘယ်လို implement လုပ်မလဲကြည့်ရအောင်။အရှင်း ဆုံးပုံစံလေးတစ်ခုတည်ဆောက် ကြည့်ပါမယ်။

```
class Account(object):
        pass
```

ခုရေးထားတဲ့ accout class မှာ ဘာ attribute ဘာ method မှမပါပါဘူး။ ခုဒီ empty class ရဲ့ instance တွေလုပ်ကြည့်ရအောင်

```
>>> from Account import Account
>>> x = Account()
>>> print x
<Account.Account object at 0x7f364120ab90>
```

```
>>>
```

Definition of Methods

Method နဲ့ function အောက်ပါအတိုင်းကွဲပါတယ်။

- Class ကပိုင်ပြီးတော့ class ထဲမှာပဲ သတ်မှတ်ထားပါတယ်။
- Method definition ရဲ့ပထမ parameter က class instance ရဲ့ reference ဖြစ်တဲ့ "self"ဖြစ်ရပါမယ်။
- Method ကိုခေါ်ရာမှာတော့ "self" parameter မပါပါဘူး

လက်ရှိ class ကို method တွေထပ်ထည့်ကြည့်ရအောင်။ method body တော့မထည့်သေးပါဘူး။

```
class Account(object):

    def transfer(self, target, amount):
        pass

    def deposit(self, amount):
        pass

    def withdraw(self, amount):
        pass

    def balance(self):
        pass
```

Constructor

Python မှာ c++,java တို့လို explicit constructor တွေမရှိပါဘူး။ဒါပေမယ့် __init__() method ကတော့နည်းနည်းတူပါတယ်။ သူက constructor လုပ်တဲ့အလုပ် တော်တော်များများ လုပ်ပေးပါတယ်။ ဥပမာ. Class instance တစ်ခုလုပ်တဲ့အခါ ၎င်းကို အရင် execute လုပ်ပါတယ်။ နာမည်ကလည်း construntor နဲ့ဆင်ပါတယ်။ ဒါပေမယ့် constructor လို့တော့ခေါ်လို့မရပါဘူး။ ဘာကြောင့်လဲဆိုတော့ __init__ ကိုခေါ်တဲ့အချိန် မှာ instance ကလုပ်ပြီးနေပြီဖြစ်ပါတယ်။

ဒါပေမယ့် object initialize လုပ်ဖို့အတွက်တော့ __init__ method ကိုသုံးပါတယ်။ init ရေးတဲ့ ပုံစံကတော့ တစ်ခြား method တွေရေးသလိုပါပဲ။

```
def __init__(self, holder, number, balance,credit_line=1500):
        self.Holder = holder
        self.Number = number
        self.Balance = balance
        self.CreditLine = credit_line
```

## Destructor

Constructor တွေမှာပြောသလိုပဲ destructor လည်းအတူတူပါပဲ။ destructor မရှိပေမယ့် ဆင်တာရှိပါတယ်။ eg. __del__ method ၊ instance တစ်ခုကိုဖျက်ဆီးတဲ့ အခါ မှာ ၎င်း ကိုခေါ်ပါတယ်။ base class မှာ __del__() method ရှိရင် ,derived class ရဲ့ __del__() method ကနေ ခေါ်ပေးရပါမယ်။ example/

```
class Greeting:
    def __init__(self, name):
        self.name = name
    def __del__(self):
        print "Destructor started"
    def SayHello(self):
        print "Hello", self.name
```

ဒီ class ကိုသုံးရင် del က __del__() method ကို direct ခေါ်တာမဟုတ်ကြောင့် တွေ့ပါလိမ့်မည်။ x1 ကိုဖျက်တဲ့အခါမှာ destructor ကိုမခေါ်ရသေးပါဘူး။ ဘာလိုလဲဆိုတော့ del က x1 object အတွက် reference count ကို လျော့လိုက်တာဖြစ်ပါတယ်။ reference count က zero ဖြစ်မှသာ destructor ကို ခေါ်ပါတယ်။

```
>>> from hello_class import Greeting
>>> x1 = Greeting("Guido")
>>> x2 = x1
>>> del x1
>>> del x2
Destructor started
```

## Complete Listing of the Account Class

```
class Account(object):
    def __init__(self, holder, number, balance,credit_line=1500):
        self.Holder = holder
        self.Number = number
        self.Balance = balance
        self.CreditLine = credit_line

    def deposit(self, amount):
        self.Balance = amount

    def withdraw(self, amount):
        if(self.Balance - amount < -self.CreditLine):
            # coverage insufficient
            return False
        else:
            self.Balance -= amount
            return True

    def balance(self):
        return self.Balance

    def transfer(self, target, amount):
       if(self.Balance - amount < -self.CreditLine):
            # coverage insufficient
            return False
        else:
```

```
            self.Balance -= amount
            target.Balance += amount
            return True
```

Account.pyလို့သိမ်းထားမယ်ဆိုရင် ဒီ class ကို interactive python shell မှာသုံးနိုင်ပါတယ်။

```
>>> import Account
>>> k = Account.Account("Guido",345267,10009.78)
>>> k.balance()
10009.780000000001
>>> k2 = Account.Account("Sven",345289,3800.03)
>>> k2.balance()
3800.0300000000002
>>> k.transfer(k2,1000)
True
>>> k2.balance()
4800.0300000000007
>>> k.balance()
9009.7800000000007
>>>
```

ဥပမာမှာ မှားနေတာလေးတွေရှိပါတယ်။attribute တွေကို အပြင်ရနေ direct access ရနေပါတယ်။

```
>>> k.balance()
9009.7800000000007
>>> k2.Balance
4800.0300000000007
>>> k2.Balance += 25000
>>> k2.Balance
29800.029999999999
>>>
```

Data Encapsulation

Identifiter ကို "_" underscore character နဲ့ မစရင် အပြင်ကနေ ရယူအသုံးပြုနိုင်ပါတယ်။ ဆိုလိုတာကတော့ တန်ဖိုးတွေကို ဖတ်လို့ရ ပြောင်းလဲလို့ရပါတယ်။ data တွေကိုကာကွယ်ချင်ရင်တော့ member တွေကို private or protected ထားရပါမယ်။ underscore character နှစ်ခုနဲ့ အစပြုထားတဲ့ instance variable name တွေကို class အပြင်ကနေ မသုံးစွဲနိုင်ပါဘူး။ direct မရပေမယ့်လည်း private name mangling သုံးပြီး access သုံးနိုင်ပါသေးတယ်။ ဆိုလိုတာကတော့ private data __A ကို အောက်ပါအတိုင်းသုံးနိုင်ပါတယ်။ instance_name._classname__A အောက်မှာရှင်းပြထားပါတယ်။

```
>>> from mangling import Mangling
>>> x = Mangling(42)
>>> x.__A
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Mangling' object has no attribute '__A'
>>> x._Mangling__A
42
>>>
```

Identifier ကို underscore တစ်ခုနဲ့ပဲ စထားတာဆိုရင် တော့ protected member ဖြစ်ပါတယ်။ protected member တွေကို class အပြင်ကနေ public လိုပဲ ရယူအသုံးပြုနိုင်ပါတယ်။

Example.။ encapsulation.py လို့သိမ်းထားမယ်။

```
class Encapsulation(object):
    def __init__(self, a, b, c):
        self.public = a
        self._protected = b
        self.__private = c
```

အောက်ပါ interactive seeeion မှာ public,protected,private member တွေဘယ်လိုအလုပ်လုပ်လဲတွေ့နိုင်ပါတယ်။

```
>>> from encapsulation import Encapsulation
>>> x = Encapsulation(11,13,17)
>>> x.public
11
>>> x._protected
13
>>> x._protected = 23
>>> x._protected
23
>>> x.__private
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Encapsulation' object has no attribute '__private'
>>>
```

အောက်မှာဘယ်လိုအလုပ်လုပ်လဲပြောထားပါတယ်။

| Name | Notation | Behaviour |
|---|---|---|
| name | Public | Can be accessed from inside and outside |
| _name | Protected | Like a public member, but they shouldn't be directly accessed from outside. |
| __name | Private | Can't be seen and accessed from outside |

Account classရဲ့ private member တွေပါတဲ့ constructor ပါ။

```
def __init__(self, holder, number, balance,credit_line=1500):
        self.__Holder = holder
        self.__Number = number
        self.__Balance = balance
        self.__CreditLine = credit_line
```

Class and Object Variables

Object variable (instance variable )တွေတော့သုံးခဲ့ပြီ။ ၎င်း variable တွေဟာ မတူညီတဲ့ object တွေအတွက်မတူညီတဲ့ value တွေရှိကြပါတယ်။eg။ account နှစ်ခုမှာဆိုရင် acoount number မတူနိုင်သလို balance တွေလည်း တူချင်မှတူပါလိမ့်မယ်။ Object variable တွေကို object တွေ class instacce တွေစတဲ့ ဆိုင်ရာဆိုင်ရာက ပိုင်ဆိုင်ကြပါတယ်။ object တွေရဲ့ကိုယ်ပိုင် variable တွေဖြစ်ပါတယ်။ object တစ်ခုခြင်းစီ အတွက် ဖန်တီးပေးထားတာ ဖြ ဖြစ်တာကြောင့် non-static or dynamic variable or members လို့ခေါ်ကြပါတယ်။
Class variable ကတော့ object အားလုံးကမျှဝေသုံးစွဲကြပါတယ်။ သူတို့ကိုတော့ မည်သည့် object ကနေမဆို ပြောင်းလဲမှုတွေလုပ်နိုင်ပါတယ်။ object အားလုံးအတွက် တစ်ခုပဲရှိတာကြောင့် အခြား object တွေအတွက်လည်းပြောင်းလဲပါတယ်။

Static variable အတွက် example အနေနဲ့။ account စုစုပေါင်း တွက်မယ့် counter variable တစ် ခုလုပ်ပါမယ်။ ဒါကို class ရဲ့အောက်မှာ တစ်ခါထဲပြုလုပ်သွားပါမယ်။ constructor ကိုခေါ်တိုင်း variable ကို တိုးသွားပါမယ်။ account ရဲ့ object ကိုဖျက်မယ်ဆိုရင် လျော့သွားပါမယ်။

```
class Account(object):
    counter = 0
    def __init__(self, holder, number, balance,credit_line=1500):
        Account.counter += 1
        self.__Holder = holder
        self.__Number = number
        self.__Balance = balance
        self.__CreditLine = credit_line
    def __del__(self):
        Account.counter -= 1
```

Interaction session မှာ ပြသွားပါမယ်။

```
>>> from Account import Account
>>> Account.counter
0
>>> a1 = Account("Homer Simpson", 2893002, 2325.21)
>>> Account.counter
1
>>> a2 = Account("Fred Flintstone", 2894117, 755.32)
>>> Account.counter
2
>>> a3 = a2
>>> Account.counter
2
>>> a4 = Account("Bill Gates", 2895007, 5234.32)
>>> Account.counter
3
```

```
>>> del a4
>>> Account.counter
2
>>> del a3
>>> Account.counter
2
>>> del a2
>>> Account.counter
1
```

## Inheritance

Account class ကနေ inherit လုပ်နိုင်တဲ့ Counter class တစ်ခုလုပ်ပြီး inheritance ကိုစမ်းကြည့့် ပါမယ်။ ဒါဆိုရင် account သို့ အခြား class တွေထဲမှာ count လုပ်စရာမလိုတော့ပါဘူး။ eg member or employee

Class တစ်ခုဘယ် CLASS ကနေ inherit လုပ်ထားတယ်ဆိုတာ(super-class ) ကိုဖော်ပြတာလွယ်ပါတယ်။ class name နောက်က () ထဲမှာ super-class ကိုထည့့်ပေးရုံပါပဲ။

အောက်မှာ account class ကို super-class အနေနဲ့ ထားတဲ့ counter class ကိုဖော်ပြထားပါတယ်။

```
class Counter(object):
    number = 0

    def __init__(self):
        type(self).number += 1

    def __del__(self):
        type(self).number -= 1

class Account(Counter):
    def __init__(self,
                 account_holder,
                 account_number,
                 balance,
                 account_current=1500):
        Counter.__init__(self)
```

## Multiple Inheritance

Class တစ်ခုသည် inherit လုပ်ခြင်းကို class တစ်ခုထက်ပို လုပ်ဆောင်နိုင်ပါတယ်။ ဒါကို multiple inheritance လို့ခေါ်ပါတယ်။ ရေးသားတာကတော့လွယ်ပါတယ်။ super-class အားလုံးကို class နောက်က () ထဲမှာ comma ခံပြီးရေးသားနိုင်ပါတယ်။

```
class NN (class1, class2, class3 ...):
```
This class inherits from class1, class2, and so on.

# Chapter 35

## Inheritance Example

### Introduction

Web ပေါ်မှာ inheritance ပတ်သတ်တဲ့ example များစွာတွေ့နိုင်ပါတယ်။ လွယ်ကူရိုးရှင်းတာတွေ ရှိသလို အရမ်းခက်ခဲတာတွေလည်း တွေ့ နိုင်ပါတယ်။ ဒါကြောင့် အရမ်းလည်း မခက်ခဲလက်တွေ့လည်း အသုံးဝင်ပြီးတော့ inheritance ရဲ့အခြေခံ သဘောတရားတွေကို ရှင်းလင်းပြချင်ပါတယ်။

ဒီအတွက် base class နှစ်ခုလုပ်ပါမယ်။ clock အတွက်ရယ် calendar အတွက်ရယ်ဖြစ်ပါတယ်။ ဒီနှစ်ခုအပေါ်မှာ အခြေခံပြီးတော့(inherit) CalendarClock ဆိုတဲ့ class ပြုလုပ်ထားပါတယ်။

### The Clock Class

```
class Clock(object):

    def __init__(self,hours=0, minutes=0, seconds=0):
        self.__hours = hours
        self.__minutes = minutes
        self.__seconds = seconds

    def set(self,hours, minutes, seconds=0):
        self.__hours = hours
        self.__minutes = minutes
        self.__seconds = seconds

    def tick(self):
        """ Time will be advanced by one second """
        if self.__seconds == 59:
            self.__seconds = 0
            if (self.__minutes == 59):
                self.__minutes = 0
                self.__hours = 0 if self.__hours==23  else self.__hours+1
            else:
                self.__minutes += 1;
        else:
            self.__seconds += 1;

    def display(self):
        print("%d:%d:%d" % (self.__hours, self.__minutes, self.__seconds))

    def __str__(self):
        return "%2d:%2d:%2d" % (self.__hours, self.__minutes,
self.__seconds)

x = Clock()
print(x)
for i in xrange(10000):
    x.tick()
print(x)
```

## The Calendar Class

```
class Calendar(object):
    months = (31,28,31,30,31,30,31,31,30,31,30,31)

    def __init__(self, day=1, month=1, year=1900):
        self.__day = day
        self.__month = month
        self.__year = year

    def leapyear(self,y):
        if y % 4:
           # not a leap year
           return 0;
        else:
           if y % 100:
             return 1;
           else:
             if y % 400:
                return 0
             else:
                return 1;

    def set(self, day, month, year):
        self.__day = day
        self.__month = month
        self.__year = year


    def get():
        return (self, self.__day, self.__month, self.__year)
    def advance(self):
        months = Calendar.months
        max_days = months[self.__month-1]
        if self.__month == 2:
                max_days += self.leapyear(self.__year)
        if self.__day == max_days:
                self.__day = 1
                if (self.__month == 12):
                        self.__month = 1
                        self.__year += 1
                else:
                        self.__month += 1
        else:
                self.__day += 1


    def __str__(self):
       return str(self.__day)+"/"+ str(self.__month)+ "/"+ str(self.__year)

if __name__ == "__main__":
   x = Calendar()
   print(x)
   x.advance()
   print(x)
```

## The Calendar-Clock Class

```
from clock import Clock
from calendar import Calendar

class CalendarClock(Clock, Calendar):

   def __init__(self, day,month,year,hours=0, minutes=0,seconds=0):
        Calendar.__init__(self, day, month, year)
        Clock.__init__(self, hours, minutes, seconds)

   def __str__(self):
       return Calendar.__str__(self) + ", " + Clock.__str__(self)


if __name__  == "__main__":
   x = CalendarClock(24,12,57)
   print(x)
   for i in range(1000):
      x.tick()
   for i in range(1000):
      x.advance()
   print(x)
```

# Chapter 36

## Slots

### Avoiding Dynamically Created Attributes

Object တွေရဲ့ attribute တွေကို "__dict__"ဆိုတဲ့ dictionary ထဲမှာသိမ်းထားပါတယ်။ အခြား dictionary တွေလိုပဲ attribute အတွက် သုံးတဲ့ dictionary တွေမှာ element ဘယ်လောက်ပါမယ် ဆိုတဲ့ တိကျတဲ့ သတ်မှတ်ချက်မရှိပါဘူး။ ကိုယ်လိုချင်လိုထပ်ထည့်နိုင်ပါတယ်။ ဒါကြောင့်ပဲ class object တွေ ရဲ့ attribute တွေကို dynamically ထပ်ထည့်နိုင်တာဖြစ်ပါတယ်။

```
>>> class A(object):
...     pass
...
>>> a = A()
>>> a.x = 66
>>> a.y = "dynamically created attribute"
```

"a"ရဲ့ attribute တွေပါတဲ့ dictionary ကိုအောက်ပါအတိုင်း ရယူအသုံးပြုနိုင်ပါတယ်။

```
>>> a.__dict__
{'y': 'dynamically created attribute', 'x': 66}
```

ဒီလို attribute တွေကို dynamic ထည့်နိုင်တာကိုတွေ့နိုင်ပါတယ်။ ဒါပေမယ့် built-in class တွေဖြစ်တဲ့ "int" သို့ "list"တွေမှာတော့လုပ်လို့မရပါဘူး။

```
>>> x = 42
>>> x.a = "not possible to do it"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'a'
>>>
>>> lst = [34, 999, 1001]
>>> lst.a = "forget it"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'a'
```

Dictionary တွေကိုattribute တွေသိမ်းဖို့သုံးတာအဆင်ပြေပါတယ်။ ဒါပေမယ့် variable နည်းနည်းလေးရှိတဲ့ object တွေအတွက် တော့ space ဖြုန်းတီးရာရောက်ပါတယ်။ instance တွေအများကြီးလုပ်မယ်ဆိုရင်တော့ ဒီလို space consumption ဟာ အန္တရာယ်ရှိပါတယ်။ ဒီပြဿနာ အတွက်ဆိုရင် slots ကိုသုံးရပါမယ်။ object attribute တွေကို dynamic ထည့်ခွင့်ပေးမည့် အစား slots ကတော့ object ဖန်တီးပြီးရင် မပြောင်းလဲနိုင်တဲ့ static-structure နဲ့အလုပ်လုပ်ပါတယ်။

Class တစ်ခုကို design လုပ်ရာမှာ slot ကိုသုံးပြီး attribute dynamic creation ကိုကာကွယ်နိုင်ပါတယ်။ slot လုပ်ဖို့အတွက် "__slots__" ဆိုတဲ့ list တစ်ခုလုပ်ပေးရပါမယ်။ list ထဲမှာ

ကိုယ်သုံးချင်တဲ့ attribute တွေပါရပါမယ်။ အောက်ပါ example မှာ val ဆိုတဲ့ attribute တစ်ခုထဲ ပါတဲ့ slot list တစ်ခုလုပ်ထားပါတယ်။

```
class S(object):

    __slots__ = ['val']

    def __init__(self, v):
        self.val = v


x = S(42)
print(x.val)

x.new = "not possible"
```

program ကို run လိုက်မယ်ဆိုရင် dynamic attribute တွေလုပ်မရတာ တွေ့ပါလိမ့်မယ်။ 42

```
Traceback (most recent call last):
  File "slots_ex.py", line 12, in <module>
    x.new = "not possible"
AttributeError: 'S' object has no attribute 'new'
```

Slot တွေဟာ object တွေကspace comsuption လုပ်တာကို ကာကွယ်ပေးတယ်လို့ပြောခဲ့ပါတယ်။ python 3.3 နောက်မှာတော့ သိပ်ပြီးအသုံးမဂင်တော့ပါဘူး။ python 3.3 မှာ object တွေကို သိမ်းဖို့အတွက် Key-Sharing dictionary တွေကိုသုံးပါတယ်။ instance တွေရဲ့ attribute တွေဟာ တစ်ခုနဲ့ တစ်ခု မျှပေသုံးစွဲလာနိုင်ပါတယ်။ ဆိုလိုတာက key သိမ်းထားတဲ့ အပိုင်းနဲ့ သက်ဆိုင်ရာ hash တွေကိုဖြစ်ပါတယ်။ ဒါကြောင့် non-builtin type instance တွေအများ ကြီး တည်ဆောက်တဲ့ program တွေမှာ memory consumption ကိုလျော့ချပေးပါတယ်။

# Chapter 37

Classes and Class Creation

New-style vs. old-style Classes

Python2 မှာအရှေ့ခန်းတွေမှာပြောခဲ့သလိုပဲ ရှုပ်ထွေးနိုင်တာတစ်ခုရှိပါတယ်။ The coexistence of old-style and new-style classes.

Official python reference မှာအောက်ပါအတိုင်းပြောထားပါတယ်။

*"New-style classes were introduced in Python 2.2 to unify classes and types. A new-style class neither more nor less than a user-defined type. If x is an instance of a new-style class, then type(x) is the same as x.*__class__*."*

နောက်တစ်ပိုဒ်ကတော့ new-style class ကိုဘာကြောင့်လုပ်ရလဲပြောထားပါတယ်။ "to provide a unified object model with a full meta-model"လို့ပြောထားပါတယ်။ အခြား ဖြစ်တဲ့ "immediate benefits" the "ability to subclass most built-in types, or the introduction of 'descriptors', which enable computed properties."တွေလည်းပြောထားပါတယ်။

Syntactic အနည်းငယ်သာကွာပါတယ်။ new-style class ဖြစ်ဖို့အတွက် new-stycle class object တစ်ခုသို့ အခြား new-style class တစ်ခုကနေ inherit လုပ်ရပါမယ်။ python 3 မှာတော့ new-style class ပဲရှိပါတယ်။

```
# old-style class
class A:
    pass
class B(A):
    pass
a = A()
b = B()
print(type(A), type(B))
print(type(a), type(b))
```

အထက်ဖော်ပြပါ program ကို execute လုပ်ရင်အောက်ပါအတိုင်းရပါမယ်။

```
(<type 'classobj'>, <type 'classobj'>)
(<type 'instance'>, <type 'instance'>)
```

```
# new-style class
class A(object):
    pass
class B(A):
    pass
a = A()
b = B()
```

```
print(type(A), type(B))
print(type(a), type(b))
```
အထက်ဖော်ပြပါ code ကို execute လုပ်ရင်အောက်ပါအတိုင်းရပါမယ်။

```
(<type 'type'>, <type 'type'>)
(<class '__main__.A'>, <class '__main__.B'>)
```

အောက်ပါခေါင်းစဉ်ကတော့ new-style class နဲ့သတ်မှတ်ထားမှရပါမယ်။

Behind the scenes: Relationship between Class and type

Class တွေသတ်မှတ်ရင် object တွေလုပ်ရင် နောက်ကွယ်မှာဘာတွေဖြစ်နေလဲ ကြည့်ရအောင် ။

Type နဲ့ class တွေကြားဆက်နွယ်မှုတွေအရင်ကြည့်မယ်။ class တစ်ခုတည်ဆောက်တဲ့အခါ နောက်ကမှာဘာတွေဖြစ်နေလဲပေါ့။ object ကို type နဲ့ကြည့်ရင်objet ဟာ ဘယ် class ကလဲ ဆိုတာတွေ့မှာဖြစ်ပါတယ်။

```
x = [4, 5, 9]
y = "Hello"
print(type(x), type(y))
```
အောက်ပါအတိုင်း တွေ့မှာပါ။

```
(<type 'list'>, <type 'str'>)
```
Class ရဲ့ name ကိုမှ type နဲ့ကြည့်ရင် type ဆိုတဲ့ class ကိုရမှာပါ

```
print(type(list), type(str))
```
ဇ အောက်ပါအတိုင်းတွေ့မှာပါ။

```
(<type 'type'>, <type 'type'>)
```
ဒါက type(x) type(y) တွေကို type နဲ့ကြည့်တာနဲ့တူပါတယ်။

```
x = [4, 5, 9]
y = "Hello"
print(type(x), type(y))
print(type(type(x)), type(type(y)))
```
ဇ အောက်ပါအတိုင်းတွေ့မှာပါ။

```
(<type 'list'>, <type 'str'>)
(<type 'type'>, <type 'type'>)
```
User-defined class ဆိုတာအမှန်တော့ type ဆိုတဲ့ class ရဲ့ object သာဖြစ်ပါတယ်။

ဒါကြောင့် class တွေဟာ type ဆိုတဲ့ class ကနေဖန်တီးတာဖြစ်ပါတယ်။ python 3 မှာ တော့ class နဲ့ type အတူတူပါပဲ။

One argument အစာ းtype ကို parameter 3 ခုနဲ့ အသုံးပြုနိုင်ပါတယ်။

```
type(classname, superclasses, attributes_dict)
```

type ကို argument သုံးခုနဲ့ခေါ်ထားပါတယ်။ type object အသစ်တစ်ခု return ပြန်ပေးပါတယ်. Class ရဲ့ dynamic form ကိုတွေ့နိုင်ပါတယ်။

- “”classname”ကတော့ class name အတွက် string ဖြစ်ပြီးတော့ name attribute ဖြစ်လာပါတယ်။
- “superclass”ကတော့ list သို့ tuple ဖြစ်ပြီးတော့ ကျွန်တော်တို့ class ရဲ့ super-class တွေပါဝင်ပါတယ်။ list or type က base attribure တွေဖြစ်လာပါတယ်။
- “attributes_dict”ကတော့ dictionary ဖြစ်ပါတယ်။ ကျွန်တော်တို့ class ရဲ့ namespace အနေနဲ့အလုပ်လုပ်ပါတယ်။ class body ရဲ့ definition တွေပါပြီးတော့ dict attribute ဖြစ်လာပါတယ်။

Class definition တစ်ခုကိုကြည့်ကြည့်ရအောင်

```
class A(object):
    pass
x = A()
print(type(x))
```

result အနေနဲ့ အောက်ပါအတိုင်းတွေ့မှာဖြစ်ပါတယ်။

```
<class '__main__.A'>
```

ဂင်းအတွက် လည်း type ကိုသုံးနိုင်ပါတယ်။

```
A = type("A", (), {})
x = A()
print(type(x))
```

result အနေနဲ့ အောက်ပါအတိုင်းတွေ့မှာဖြစ်ပါတယ်။

```
<class '__main__.A'>
{'__doc__': None, '__module__': '__main__', '__dict__': <attribute
'__dict__' of 'A' objects>, '__weakref__': <attribute '__weakref__' of 'A'
objects>}
```

ပြောရရင် class A ကိုအောက်ပါအတိုင်းသတ်မှတ်နိုင်ပါတယ်။

```
type(classname, superclasses, attributedict)
```

“type”ကိုခေါ်တဲ့အခါမှာ type ရဲ့ call method ကိုခေါ်ပါတယ်။ call method ကနေမှ အခြား method တွေဖြစ်တဲ့ new နဲ့ init ကိုခေါ်ပါတယ်။

```
type.__new__(typeclass, classname, superclasses, attributedict)
type.__init__(cls, classname, superclasses, attributedict)
```

new method ကနေ မှ class objet အသစ်တစ်ခုလုပ်ပြီးတော့ return ပြန်ပေးပါတယ်။ ဂင်းနောက်မှာတော့ init method ကနေပြီးတော့ အသစ်ဖန်တီးလိုက်တဲ့ object ကို initialize လုပ်ပါတယ်။

```
class Robot(object):
    counter = 0
    def __init__(self, name):
        self.name = name
    def sayHello(self):
        return "Hi, I am " + self.name
```

```
def Rob_init(self, name):
    self.name = name
Robot2 = type("Robot2",
              (),
              {"counter":0,
               "__init__": Rob_init,
               "sayHello": lambda self: "Hi, I am " + self.name})
x = Robot2("Marvin")
print(x.name)
print(x.sayHello())
y = Robot("Marvin")
print(y.name)
print(y.sayHello())
print(x.__dict__)
print(y.__dict__)
```

အောက်ပါ output ကိုရပါမယ်။

```
Marvin
Hi, I am Marvin
Marvin
Hi, I am Marvin
{'name': 'Marvin'}
{'name': 'Marvin'}
```

Robot နဲ့Robot2 class နှစ်ခုဟာ syntactically အနေနဲ့ မတူပါဘူး logically အနေနဲ့ တော့ အတူတူပဲဖြစ်ပါတယ်။

ပထမဥပမာမှာ python ့လုပ်တာက ပုံမှန်အတိုင်း class လုပ်ရင်ဘာတွေလုပ်သလဲဆိုတော့။ python ကနေပြီးတော့ class Robot ရဲ့ statement တွေကို process လုပ်ပြီးတော့ robot class ရဲ့ method တွေ attribute တွေကိုရယူကာ call type ရဲ့ attributes_dict ထဲကိုထည့်ပါတယ်။ ဒါကြောင့် python က Robot2 မှာလုပ်သလိုပဲ type ကို call လုပ်ပါတယ်။

# Chapter 38

On the Road to Metaclasses

Motivation for Metaclasses

ဒီအပိုင်းမှာတော့ metaclass ဘာကြောင့်သုံးရတာလဲ ဆိုတဲ့အကြောင်းပြောချင်ပါတယ် ။ metaclass ကနေဖြေရှင်းပေး နိုင်းတဲ့ design problem တွေကို demo ပြဖို့ philosopher class တွေတည်ဆောက်သွားပါမယ်။ philosopher class တိုင်းအတွက်(Philosopher1,philosoper2,etc) method တွေ ပါဝင်ပါမယ်။ လုံးဝမကောင်းတဲ့ နည်းလမ်းကတော့ class တိုင်းမှာ တူညီတဲ့ code တွေပါနေတာပါပဲ။

```
class Philosopher1:

    def the_answer(self, *args):
        return 42

class Philosopher2:
    def the_answer(self, *args):
        return 42

class Philosopher3:
    def the_answer(self, *args):
        return 42

plato = Philosopher1()
print(plato.the_answer())
kant = Philosopher2()
# let's see what Kant has to say :-)
print(kant.the_answer())
42
42
```

"the_answer" method ကို ပွားထားတာတွေတွေ့မှာဖြစ်ပါတယ်။ error ဖြစ်နိုင်လို maintain လုပ်ဖို့လည်းခက်ပါတယ်။

သိခဲ့သလောက်နဲ့ redundant code တွေကိုဖြေရှင်းမယ်ဆိုရင်တော့ "the_answer" method ပါဝင်တဲ့ Base class တစ်ခုတည်ဆောက်မှာပါ ။ပြီးတော့ Philosoper class တွေက inherit လုပ်မှာဖြစ်ပါတယ်။

```
class Answers:
    def the_answer(self, *args):
        return 42

class Philosopher1(Answers):
    pass
class Philosopher2(Answers):
    pass
class Philosopher3(Answers):
    pass
plato = Philosopher1()
print(plato.the_answer())
kant = Philosopher2()
# let's see what Kant has to say :-)
```

```
print(kant.the_answer())
42
42
```

ဒီလိုလုပ်လိုက်ရင် philosopher class တိုင်းမှာ "the_answer"method ပါနေမှာပါ။ ၎င်း method လိုမလိုဆိုတာ ကြိုမသိနိုင်ဘူးဆိုပါစို့။ ဒါကို run-time မှာပဲဆုံးဖြတ်မယ် လို့ထားလိုက်ပါမယ်။ ဒါကိုဆုံးဖြစ်တာကို configuration file,user input ,တွက်ချက်မှုတစ်ခုပေါ်မူတည်လိမ့်မယ်။

```
# the following variable would be set as the result of a runtime
calculation:
x = input("Do you need the answer? (y/n): ")
if x=="y":
    required = True
else:
    required = False

def the_answer(self, *args):
        return 42

class Philosopher1:
    pass
if required:
    Philosopher1.the_answer = the_answer

class Philosopher2:
    pass
if required:
    Philosopher2.the_answer = the_answer

class Philosopher3:
    pass
if required:
    Philosopher3.the_answer = the_answer


plato = Philosopher1()
kant = Philosopher2()
# let's see what Plato and Kant have to say :-)
if required:
    print(kant.the_answer())
    print(plato.the_answer())
else:
    print("The silence of the philosphers")
Do you need the answer? (y/n): y
42
42
```

ဒါက ဒီပြဿနာအတွက် ဖြေရှင်းချက်တစ်ခုဆိုပေမဲ့ မကောင်းတာတွေရှိပါသေးတယ်။ ဘာလိုလဲဆိုတော့ class တိုင်းကို code အတူတူတွေထည့်ရမှာဖြစ်ပြီး ဒါကိုမေ့နိုင်ပါသေးတယ်။ ဒဲ့အပြင် method တွေအများကြီးဆိုရင် ရှုပ်ထွေးနိုင်ပါသေးတယ်။

ဒါကိုပိုကောင်းအောင်း manager function တစ်ခုလုပ်ပြီး redundant code တွေဖယ်နိုင်ပါတယ်။ manager function ကိုသုံးပြီးတော့ calss ကို conditionally augument လုပ်မှာပါ။

```
# the following variable would be set as the result of a runtime
calculation:
```

```
x = input("Do you need the answer? (y/n): ")
if x=="y":
    required = True
else:
    required = False

def the_answer(self, *args):
        return 42

# manager function
def augment_answer(cls):
    if required:
        cls.the_answer = the_answer


class Philosopher1:
    pass
augment_answer(Philosopher1)
class Philosopher2:
    pass
augment_answer(Philosopher2)
class Philosopher3:
    pass
augment_answer(Philosopher3)


plato = Philosopher1()
kant = Philosopher2()
# let's see what Plato and Kant have to say :-)
if required:
    print(kant.the_answer())
    print(plato.the_answer())
else:
    print("The silence of the philosphers")
Do you need the answer? (y/n): y
42
42
```

ပြဿနာကိုဖြေရှင်းဖို့အသုံးဝင်ပေမယ့်ကျွန်တော် တို့က manage fuction ဖြစ်တဲ့ "augument_answer" ကို ခေါ်ဖို့မေ့နိုင်ပါသေးတယ်။ code တွေကို အလိုအလျောက် execute လုပ်သင့်ပါတယ်။ class definition ပြီးလျှင် အလိုအလျောက် အလုပ်လုပ်မယ့် code အချို့လို နေပါတယ်။

```
# the following variable would be set as the result of a runtime
calculation:
x = input("Do you need the answer? (y/n): ")
if x=="y":
    required = True
else:
    required = False


def the_answer(self, *args):
        return 42

def augment_answer(cls):
    if required:
        cls.the_answer = the_answer
    # we have to return the class now:
```

```
    return cls


@augment_answer
class Philosopher1:
    pass
@augment_answer
class Philosopher2:
    pass
@augment_answer
class Philosopher3:
    pass


plato = Philosopher1()
kant = Philosopher2()


# let's see what Plato and Kant have to say :-)
if required:
    print(kant.the_answer())
    print(plato.the_answer())
else:
    print("The silence of the philosphers")
Do you need the answer? (y/n): y
42
42
```

Metaclass တွေကိုဖောပြပါပြသနာတွေအတွက်သုံးနိုင်ပါတယ်။

# Chapter 39

## Metaclasses

Metaclass တွေဆိုတာကတော့ class တစ်ခုပဲဖြစ်ပြီးတော့ ၎င်း ရဲ့ object တွေကလည်း class တွေပဲဖြစ်ပါတယ်။ ပုံမှန် classs တစ်ခုက class object behavior တွေကိုသတ်မှတ်သလိုပဲ ,metaclass ကလည်း class တွေနဲ့ ၎င်း ရဲ့ object တွေအတွက်ကို behavior တွေသတ်မှတ်ပေးပါတယ်။

Metaclass တွေကို OOP language တိုင်းက ထောက်ပံ့ပေးတာတော့မဟုတ်ပါဘူး။ အထောက်အပံ့ မပေးတဲ့ language တွေမှာ အခြားနည်းသုံးကြပါတယ်။ python ကတော့ ထောက်ပံ့ ပေးပါတယ်။

Metaclass အသုံးပြုပုံများစွာရှိပါတယ်။

- logging and profiling
- interface checking
- registering classes at creation time
- automatically adding new methods
- automatic property creation
- proxies
- automatic resource locking/synchronization.

## Defining Metaclasses

Metaclass ကအခြား class နဲ့တူပေမယ့် အခြား class တွေက "type" ကနေ inherit လုပ်ထားတာဖြစ်ပါတယ်။ နောက်မတူတာက metaclass ကိုသုံးထားတဲ့ class statement တွေဆုံးရင် metaclass တွေကို အလိုအလျောက် ခေါ်ပေးပါတယ်။ metaclas မသုံးထားရင် type() class ကိုခေါ်တာဖြစ်ပြီး သုံးထားရင်တော့ type အစား metaclass ကိုခေါ်တာဖြစ်ပါတယ်။

Metaclass လေးတစ်ခုတည်ဆောက်ကြည့်ပါမယ်။ __new__ method ထဲက argument content တွေကို print ထုတ်တာရယ် __new__ type ကိုခေါ်တဲ့အခါရလာတဲ့ result ကို return ပြန်တာရယ်ပါပါမယ်။

```
class LittleMeta(type):
    def __new__(cls, clsname, superclasses, attributedict):
        print("clsname: ", clsname)
        print("superclasses: ", superclasses)
        print("attributedict: ", attributedict)
        return type.__new__(cls, clsname, superclasses, attributedict)
```

"LittleMeta" ဆိုတဲ့ metaclass ကိုအောက်ပါ example မှာသုံးပါမယ်။

```
class S:
```

```
    pass
class A(S, metaclass=LittleMeta):
    pass
a = A()
clsname:  A
superclasses:  (<class '__main__.S'>,)
attributedict:  {'__module__': '__main__', '__qualname__': 'A'}
```

LittleMeta.__new__ ကိုခေါ်တာဖြစ်ပြီးတော့ type.__new__ ကိုမသုံးတာတွေ့ပါမယ်။

အရင် chapter က အပိုင်းကိုပြန်သွားရအောင် "EssentialAnswers"ဆိုတဲ့ metaclass တည်ဆောက်ပါမယ်။ ၎င်း က လိုအပ်တဲ့ augument_answer method ကိုအလိုအလျောက်ထည့်သွင်း ပေးနိုင်ပါတယ်။

```
x = input("Do you need the answer? (y/n): ")
if x.lower() == "y":
    required = True
else:
    required = False

def the_answer(self, *args):
        return 42

class EssentialAnswers(type):

    def __init__(cls, clsname, superclasses, attributedict):
        if required:
            cls.the_answer = the_answer


class Philosopher1(metaclass=EssentialAnswers):
    pass
class Philosopher2(metaclass=EssentialAnswers):
    pass
class Philosopher3(metaclass=EssentialAnswers):
    pass


plato = Philosopher1()
print(plato.the_answer())
kant = Philosopher2()
# let's see what Kant has to say :-)
print(kant.the_answer())
Do you need the answer? (y/n): y
42
42
```

Class definition ပြီးရင် pyton ကအောက်ပါ function ကိုခေါ်ပါတယ်။

```
type(classname, superclasses, attributes_dict)
```

metaclass ကို header အနေနဲ့ပြောထားရင် ဒါကပြဿနာမဟုတ်ပါဘူး။ ခုန example မှာ ဒီလိုလုပ်ထားတာဖြစ်ပါတယ်။ Philosopher1, Philosopher2 and Philosopher3တွေကို metaclass ဖြစ်တဲ့ EssentialAnswers နဲ့ချိတ်ထားပါတယ်။ဒါကြောင့် type အစား EssentialAnswer ကိုခေါ်တာဖြစ်ပါတယ်။

```
EssentialAnswer(classname, superclasses, attributes_dict)
```

အတိအကျပြောရရင် call လုပ်တဲ့ argument တွေကိုအောက်ပါတန်ဖိုးတွေဖြစ်ပါတယ်။

```
EssentialAnswer('Philopsopher1',
                (),
                {'__module__': '__main__', '__qualname__': 'Philosopher1'})
```

အခြား philosopher class တွေအတွက်လည်းအတူတူပဲဖြစ်ပါတယ်။

## Creating Singletons using Metaclasses

Singleton pattern ဆိုတာကတော့ design pattern တစ်ခုဖြစ်ပြီးတော့ class တစ်ခုအတွက် ကို object တစ်ခုလိုကန်သတ်ထားပါတယ်။ object တစ်ခုပဲလိုအပ်တဲ့အခါသုံးပါတယ်။

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton,
cls).__call__(*args, **kwargs)
        return cls._instances[cls]

class SingletonClass(metaclass=Singleton):
    pass
class RegularClass():
    pass
x = SingletonClass()
y = SingletonClass()
print(x == y)
x = RegularClass()
y = RegularClass()
print(x == y)
True
False
```

## Creating Singletons using Metaclasses

နောက်တစ်နည်းအနေနဲ့တော့ singleton class တစ်ခု ကို အခြား singleton class တစ်ခုကနေ inherit လုပ်ပြီး ဖန်တီးတာဖြစ်ပါတယ်။

```
class Singleton(object):
    _instance = None
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = object.__new__(cls, *args, **kwargs)
        return cls._instance

class SingletonClass(Singleton):
    pass
class RegularClass():
    pass
x = SingletonClass()
y = SingletonClass()
print(x == y)
x = RegularClass()
y = RegularClass()
print(x == y)

True
False
```

# Chapter 40

Metaclass Use Case: Count Function Calls

## Introduction

ဒီအခန်းမှာတော့ meatclass use case ကို ဥပမာနဲ့ပြောသွားမှာဖြစ်ပါတယ်။ metaclass တစ်ခုကို တည်ဆောက်မယ် ၎င်း ပြီးတော့ subclass ရဲ့ method တွေကို decorate လုပ်မှာဖြစ်ပါတယ်။ decorator ကနေ return ပြန်လာတဲ့ decorated function တွေက subclass ရဲ့ method တွေကို ဘယ်နှစ်ကြိမ် call လုပ်လဲဆိုတာကို ရေတွက်နိုင်ပါတယ်။

## Preliminary Remarks

ပြဿနာကို မဖြေရှင်းခင် class attribute တွေကို ဘယ်လို access လုပ်လဲပြန်ကြည့်ရအောင်။ list class နဲ့ပြပါမယ်။ class တစ်ခုရဲ့ non-private attribute list ကိုအောက်ပါအတိုင်းရနိုင်ပါတယ်။

```
import random
cls = "random" # name of the class as a string
all_attributes = [x for x in dir(eval(cls)) if not x.startswith("__") ]
print(all_attributes)
```

ဒါဆိုအောက်ပါအတိုင်းရပါမယ်။

```
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST',
'SystemRandom', 'TWOPI', '_BuiltinMethodType', '_MethodType', '_Sequence',
'_Set', '_acos', '_ceil', '_cos', '_e', '_exp', '_inst', '_log', '_pi',
'_random', '_sha512', '_sin', '_sqrt', '_test', '_test_generator',
'_urandom', '_warn', 'betavariate', 'choice', 'expovariate',
'gammavariate', 'gauss', 'getrandbits', 'getstate', 'lognormvariate',
'normalvariate', 'paretovariate', 'randint', 'random', 'randrange',
'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform',
'vonmisesvariate', 'weibullvariate']
```

Public method တွေကိုပဲ filter လုပ်ကြည့်ပါမယ်။

```
methods = [x for x in dir(eval(cls)) if not x.startswith("__")
                                and callable(eval(cls + "." + x))]
print(methods)
```

အောက်ပါအတိုင်းတွေ့ရမှာဖြစ်ပါတယ်။

```
['Random', 'SystemRandom', '_BuiltinMethodType', '_MethodType',
'_Sequence', '_Set', '_acos', '_ceil', '_cos', '_exp', '_log', '_sha512',
'_sin', '_sqrt', '_test', '_test_generator', '_urandom', '_warn',
'betavariate', 'choice', 'expovariate', 'gammavariate', 'gauss',
'getrandbits', 'getstate', 'lognormvariate', 'normalvariate',
'paretovariate', 'randint', 'random', 'randrange', 'sample', 'seed',
'setstate', 'shuffle', 'triangular', 'uniform', 'vonmisesvariate',
'weibullvariate']
```

Non-public attribute တွေအတွက်ဆိုရင်တော့ not ထည့်ပေးရုံပါပဲ

```
non_callable_attributes = [x for x in dir(eval(cls)) if not
x.startswith("__")
                           and not callable(eval(cls + "." + x))]
print(non_callable_attributes)
```

ဒါဆိုရင်အောက်ပါ result ရပါမယ်။

```
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'SG_MAGICCONST', 'TWOPI',
'_e', '_inst', '_pi', '_random']
```

ပုံမှ python program တွေအတွက်တော့ အောက်မှာဖော်ပြထားတဲ့ပုံစံတွေ မလိုပေမဲ့ဖြစ်နိုင်ပါသေးတယ်။

```
lst = [3,4]
list.__dict__["append"](lst, 42)
lst
```

ဆိုရင်အောက်ပါအတိုင်းရရှိပါလိမ့်မယ်။

```
[3, 4, 42]
```

A Decorator for Counting Function Calls

Metaclass design ကိုစရအောင် ။ ၎င်း subclass ထဲက method တွေကို call တွေရေတွက်ပေးမယ့် decorator နဲ့ decorate လုပ်ပါမယ်။

```
def call_counter(func):
    def helper(*args, **kwargs):
        helper.calls += 1
        return func(*args, **kwargs)
    helper.calls = 0
    helper.__name__= func.__name__
    return helper
```

ပုံမှန်သုံးသလိုပဲသုံးနိုင်ပါတယ်။

```
@call_counter
def f():
    pass
print(f.calls)
for _ in range(10):
    f()

print(f.calls)
```

အောက်ပါအတိုင်းထုတ်ပေးပါလိမ့်မယ်။

```
0
10
```

Decorator ကိုနောက်တစ်မျိုးခေါ်ပုံလည်းမှတ်ထားသင့်ပါတ်။

```
def f():
    pass
f = call_counter(f)
print(f.calls)
for _ in range(10):
    f()

print(f.calls)
def f():
    pass
f = call_counter(f)
print(f.calls)
for _ in range(10):
    f()

print(f.calls)
```

## The "Count Calls" Metaclass

Metaclass ရေးဖို့လိုအပ်တာတွေစုံသွားပြီဖြစ်ပါတယ်။ call_decorator ကို static method အနေနဲ့ ထည့်ထားပါတယ်။

```
class FuncCallCounter(type):
    """ A Metaclass which decorates all the methods of the
        subclass using call_counter as the decorator
    """

    @staticmethod
    def call_counter(func):
        """ Decorator for counting the number of function
            or method calls to the function or method func
        """
        def helper(*args, **kwargs):
            helper.calls += 1
            return func(*args, **kwargs)
        helper.calls = 0
        helper.__name__= func.__name__

        return helper


    def __new__(cls, clsname, superclasses, attributedict):
        """ Every method gets decorated with the decorator call_counter,
            which will do the actual call counting
        """
        for attr in attributedict:
            if not callable(attr) and not attr.startswith("__"):
                attributedict[attr] = cls.call_counter(attributedict[attr])

        return type.__new__(cls, clsname, superclasses, attributedict)

class A(metaclass=FuncCallCounter):

    def foo(self):
        pass
```

```
    def bar(self):
        pass
if __name__ == "__main__":
    x = A()
    print(x.foo.calls, x.bar.calls)
    x.foo()
    print(x.foo.calls, x.bar.calls)
```

This gets us the following:

```
0 0
1 0
```

# Chapter 41

Abstract Classes

Abstract class တွေကတော့ abstract method များပါဝင်တဲ့ class တွေဖြစ်ပါတယ်။ abstract method တွေကတော့(declare) ကြေငြာထားပေမယ့် implement မလုပ်ရသေးတာတွေဖြစ်ပါတယ်။ abastract class တွေကို instantiate မလုပ်တာဖြစ်နိုင်သလို subclass တွေအနေနဲ့ abastrct method တွေကို implement လုပ်ပေးဖို့လိုအပ်ပါတယ်။ python ရဲ့ abstract class တစ်ခုရဲ့ subclass ကတော့ paret class ရဲ့ abstract method တွေကို implement လုပ်စရာမလိုပါဘူး။

Example ကြည့်ရအောင်

```
class AbstractClass:

    def do_something(self):
        pass


class B(AbstractClass):
    pass
a = AbstractClass()
b = B()
```

ဒါကိုကြည့်ရင် abastract class မဟုတ်တာကိုတွေ့ပါမယ်။ ဘာလို့လဲဆိုတော့

- instantiate လည်းလုပ်ထားတယ်
- B ရဲ့ class definition မှာ do_something ကို implement လုပ်စရာမလိုပါဘူး။

ဒီ example မှာ inheritance လုပ်တာလေးပဲပါပြီး တော့ abstract class နဲ့ဘာမှမဆိုင်ပါဘူး။ python မှာကိုယ်ပိုင်အနေနဲ့ abstract class မပါပါဘူး။ဒါပေမယ့် abstract base classes တွေအတွက် abc လို့ခေါ်တဲ့ Module တော့ပါပါတယ်။

အောက်မှာ abc module ကိုသုံးပြီးတော့ abtract base class တစ်ခုတည်ဆောက်ထားပါတယ်။

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    def __init__(self, value):
        self.value = value
        super().__init__()

    @abstractmethod
    def do_something(self):
        pass
```

အခု abstract class ကိုသုံးဖို့ subclass လုပ်မယ်။ do_something method ကို implement လုပ်ဖို့လိပေမယ့် မလုပ်ရသေးတာတွေ့မှဖြစ်ပါတယ်။ဘာကြောင့်လဲဆိုတော့ ၎င်း method ကို "abstractmethod" ဆိုတဲ့ decorator နဲ့ docorate လုပ်ထားလို့ဖြစ်ပါတယ်။DoAdd42 ကို instantiate လုပ်မရပါဘူးဆိုတဲ့ exception တက်ပါလိမ့်မယ်။

```
class DoAdd42(AbstractClassExample):
    pass
x = DoAdd42(4)
```

ရလာတဲ့ result က

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-83fb8cead43d> in <module>()
      2     pass
      3
----> 4 x = DoAdd42(4)
TypeError: Can't instantiate abstract class DoAdd42 with abstract methods
do_something
```

နောက်တစ်ခုမှာမှန်အောင်ပြန်ရေးပါမယ်။ class နှစ်ခုဆောက်ပြီးတော့ abstract class ကနေ inherit လုပ်ပါမယ်။

```
class DoAdd42(AbstractClassExample):
    def do_something(self):
        return self.value + 42

class DoMul42(AbstractClassExample):

    def do_something(self):
        return self.value * 42

x = DoAdd42(10)
y = DoMul42(10)
print(x.do_something())
print(y.do_something())
52
420
```

Abstract class ကနေ derived လုပ်ထားတဲ့ class ကို instantiate လုပ်မယ်ဆိုရင် abstract method အားလုံးကို overridden လုပ်ထားမှရပါမယ်။

Abstract metchod တွေကို abstract base class ထဲမှာ implement မလုပ်နိုင်ဘူး ထင်ပါလိမ့်မယ်။ ဒါပေမယ့် လုပ်လို့ရပါတယ်။ implement လုပ်ထားရင်တောင်မှ subclass တွေထဲမှာ implementation တွေကို overridden လုပ်ပေးရပါမယ်။ အခြား ပုံမှန် inheritance တွေမှာလို abstract class တွေကနေ super() ကိုခေါ်နိုင်ပါတယ်။ ဒါကြောင့် abastract classs ထဲမှာ basic functionality တွေထည့်နိုင်ပြီးတော့ ၎င်း တို့ကို subclass တွေထဲမှာ implement လုပ်နိုင်ပါတယ်။

```
from abc import ABC, abstractmethod

class AbstractClassExample(ABC):

    @abstractmethod
    def do_something(self):
        print("Some implementation!")

class AnotherSubclass(AbstractClassExample):
    def do_something(self):
        super().do_something()
```

```
        print("The enrichment from AnotherSubclass")

x = AnotherSubclass()
x.do_something()

Some implementation!
The enrichment from AnotherSubclass
```

| Original website | http://python-course.eu/ |
|---|---|
| ဘာသာပြန်ဆိုသူ | ဟန်ဖြိုးဦး |
| ရက်စွဲ | 10/4/2017 |