সূচিপত্র

পরিচিতি	0
ইনস্টলেশন	1
ব্যাসিক কনসেপ্ট	2
সাধারণ কিছু অপারেশন	2.1
আরও কিছু নিউমেরিক অপারেশন	2.2
স্ট্রিং	2.3
ব্যাসিক ইনপুট আউটপুট	2.4
স্ট্রিং অপারেশন	2.5
টাইপ কনভার্সন	2.6
ভ্যারিয়েবল	2.7
ইনপ্লেস অপারেটর	2.8
এডিটর এর ব্যবহার	2.9
কন্ট্রোল স্ট্রাকচার	3
বুলিয়ান	3.1
if স্টেটমেন্ট	3.2
else স্টেটমেন্ট	3.3
বুলিয়ান লজিক	3.4
অপারেটর প্রেসিডেন্স	3.5
while লুপ	3.6
লিস্ট	3.7
লিস্ট অপারেশন	3.8
লিস্ট ফাংশন	3.9
রেঞ্জ	3.10
for লুপ	3.11
গুরুত্বপূর্ণ ডাটা টাইপ	4
None	4.1
ডিকশনারি	4.2
ডিকশনারি ফাংশন	4.3
টাপল	4.4
আবারও লিস্ট	4.5

- লিস্ট ও ডিকশনারি কম্প্রিহেনশন 4.6
- ফাংশন ও মডিউল 5
  - কোডের পুনব্যবহার 5.1
  - ফাংশন 5.2
  - ফাংশন আর্গুমেন্ট 5.3
  - ফাংশন রিটার্ন 5.4
  - কমেন্ট ও ডক স্ট্রিং 5.5
  - অবজেক্ট হিসেবে ফাংশন 5.6
  - মডিউল 5.7
  - স্ট্যান্ডার্ড লাইব্রেরী 5.8
  - pip 5.9
- ফাইল ও এক্সেপশন 6
  - এক্সেপশন 6.1
  - এক্সেপশন হ্যান্ডেলিং 6.2
  - finally 6.3
  - এক্সেপশন Raise 6.4
  - Assertions 6.5
  - ফাইল খোলা 6.6
  - ফাইল পড়া 6.7
  - ফাইলে লেখা 6.8
  - ফাইল নিয়ে সঠিক কাজ 6.9
- ফাংশনাল প্রোগ্রামিং 7
  - ভূমিকা 7.1
  - ল্যামডা 7.2
  - ম্যাপ ও ফিল্টার 7.3
  - জেনারেটর 7.4
  - ডেকোরেটর 7.5
  - রিকারসন 7.6
  - সেট 7.7
  - itertools 7.8
- অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং 8
  - ক্লাস 8.1
  - ইনহেরিটেন্স 8.2
  - ম্যাজিক মেথড 8.3

অপারেটর অভারলোডিং 8.4
অবজেক্ট লাইফ সাইকেল 8.5
ডাটা হাইডিং 8.6
ক্লাস মেথড ও স্ট্যাটিক মেথড 8.7
প্রোপার্টিস 8.8
রেগুলার এক্সপ্রেশন 9
পরিচিতি 9.1
মেটা ক্যারেক্টার 9.2
ক্যারেক্টার ক্লাস 9.3
গ্রুপ 9.4
স্পেশাল সিকুয়েন্স 9.5
অতিরিক্ত কিছু বিষয় 10
পাইথনিকনেস 10.1
PEP 10.2
__main__ 10.3
-*- coding: utf-8 -*- 10.4
#! /usr/bin/env python 10.5
CPython 10.6
ডকুমেন্টেশন পড়া 10.7
প্যাকেজিং 11

বাংলায় পাইথন

Like Share 10K people like this. Sign Up to see what your friends like.

কোর্স পরিচালনায় **(Lead Author)**

Nuhil Mehdy

স্বয়ংক্রিয় কন্ট্রিবিউটরের তালিকা
(প্রথম ৫ জন)

- [139] Nuhil Mehdy
- [025] Abu Ashraf Masnun
- [020] Syed Sirajul Islam Anik
- [018] Manash Kumar Mandal
- [006] Ahmed shamim

প্রারম্ভিকা

পাইথন একটি ডায়নামিক প্রোগ্রামিং ল্যাঙ্গুয়েজ যেটি জয় করেছে বহু ডেভেলপারের হৃদয় । এর মধ্যে আছে গুগল, ড্রপবক্স, ইন্সটাগ্রাম, মোজিলা সহ অনেক বড় বড় প্রতিষ্ঠানের হাজারো প্রকৌশলী । পাইথন এমন একটি ভাষা যার গঠন শৈলী অনন্য এবং প্রকাশভঙ্গি অসাধারণ । চমৎকার এই ল্যাঙ্গুয়েজটি তাই আজ ছড়িয়ে পড়েছে নানা দিকে - ওয়েব, ডেস্কটপ, মোবাইল, সিস্টেম এ্যাডমিনিস্ট্রেশন, সাইন্টিফিক কম্পিউটিং কিংবা মেশিন লার্নিং - সবত্রই পাইথনের দৃপ্ত পদচারণা।

আরও নির্দিষ্ট করে বলতে গেলে - Django, Flask, Tornado ইত্যাদি ফ্রেমওয়ার্ক এর মাধ্যমে ওয়েব অ্যাপ্লিকেশন ডেভেলপমেন্ট করতে চাইলে পাইথন জানা অবশ্যই গুরুত্বপূর্ণ। আবার ডেস্কটপ বা গ্রাফিক্যাল ইউজার ইন্টারফেইস সমৃদ্ধ সফটওয়্যার ডেভেলপমেন্টের জন্য পাইথন প্রোগ্রামিং এর জ্ঞানকে ব্যবহার করা যাবে PyQT এর মত টুলকিট বা Tkinter এর মত প্যাকেজ এর সাথে। আরও আছে Kivy এর মত লাইব্রেরী।

বর্তমানে বহুল আলোচিত এবং ভবিষ্যতের প্রযুক্তির ভিত্তি ডাটা সায়েন্স এবং মেশিন লার্নিং, সর্বোপরি আর্টিফিশিয়াল ইন্টেলিজেন্স নিয়ে কাজ করতে চাইলে পাইথন হতে পারে নির্দ্বিধায় প্রথম পছন্দের প্ল্যাটফর্ম। কারণ, scikit-learn এর মত মেশিন লার্নিং লাইব্রেরী, Pandas এর মত ডাটা ফ্রেম লাইব্রেরী, Numpy এর মত ক্যালকুলেশন লাইব্রেরী যেগুলো এক কথায় অনন্য- এসবই আছে পাইথনের জন্য।

সিরিয়াস লোকজন ইন্টারনেট অফ থিংস নিয়ে কাজ করতে চাইলেও রাস্পবেরি-পাই, বা এরকম হার্ডওয়্যার প্ল্যাটফর্ম গুলোর সাথে পাইথনের কম্বিনেশন হতে পারে চমৎকার। আছে RPi.GPIO. আর মজার লোকজনের গেম ডেভেলপমেন্ট এর জন্য আছে PyGame.

এরকম আরও অসংখ্য প্ল্যাটফর্মে পাইথনের দৃপ্ত পদচারণা বেড়েই চলেছে আর তাই বাংলাদেশের ডেভেলপারদের মধ্যে এই ভাষাটি ছড়িয়ে দিতে আমাদের এই ক্ষুদ্র প্রয়াস।

বাংলাদেশী পাইথন ইউজার গ্রুপ

বাংলাদেশী পাইথন ডেভেলপারদের মিলনকেন্দ্র এই ফেইসবুক গ্রুপটি । এটি বাংলাদেশের সবচেয়ে বড় পাইথন ইউজার গ্রুপ । এই গ্রুপের সদস্যরা বাংলাদেশে পাইথন প্রসারে প্রতিনিয়ত অবদান রেখে চলেছেন ।

পাইথন বাংলাদেশের জন্ম হয় ফেইসবুকের বাইরে । মূল ওয়েবসাইটের সাথে ফেইসবুক গ্রুপটির নাম নিয়ে যাতে কনফিউশন তৈরি না হয় তাই ফেইসবুক গ্রুপটির নাম পরবর্তীকালে পরিবর্তন করে রাখা হয় - "পাইচার্মার্স" । বর্তমানে গ্রুপটির নাম "Python Bangladesh" এবং অফিসিয়াল ওয়েব সাইটের ঠিকানা: http://pybd.org

ওপেন সোর্স

এই বইটি মূলত স্বেচ্ছাশ্রমে লেখা এবং বইটি সম্পূর্ন ওপেন সোর্স । এখানে তাই আপনিও অবদান রাখতে পারেন লেখক হিসেবে । আপনার কন্ট্রিবিউশান গৃহীত হলে অবদানকারীদের তালিকায় আপনার নাম স্বয়ংক্রিয়ভাবে যুক্ত হয়ে যাবে।

এটি মূলত একটি গিটহাব রিপোজিটোরি যেখানে এই বইয়ের আর্টিকেল গুলো মার্কডাউন ফরম্যাটে লেখা হচ্ছে । রিপোজিটরিটি ফর্ক করে পুল রিকুয়েস্ট পাঠানোর মাধ্যমে আপনারাও অবদান রাখতে পারেন । বিস্তারিত দেখতে পারেন এই ভিডিওতে Video

> বর্তমানে বইটির কন্টেন্ট বিভিন্ন কন্ট্রিবিউটর এবং নানা রকম সোর্স থেকে সংগৃহীত এবং সংকলিত।

Like 280 Share
gitter join chat

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

ইন্সটলেশন

আপনি যদি লিনাক্স বা ম্যাক ব্যবহারকারী হন তবে আপনার কম্পিউটারে পাইথন দেওয়াই থাকে। এই কোর্স লেখা পর্যন্ত (জুলাই ২০১৬) এই মেজর দুটি অপারেটিং সিস্টেমের সাথে যে পাইথন বিল্ট ইন অবস্থায় ডিফল্ট হিসেবে থাকে তার ভার্সন হচ্ছে `Python 2.7.x` . কিন্তু, এই কোর্সটি লেখা হচ্ছে `Python 3.5.x` এর উপর ভিত্তি করে। আসলে পাইথন ২ এবং ৩ ভার্সনের মধ্যে সিনট্যাক্স এবং ফিচার সম্পর্কিত বেশ কিছু মাঝারি মানের পরিবর্তন আছে। পাইথনের অফিসিয়াল সাইটে বর্তমানে পাইথন ৩ কেই বেশি ফোকাস করা হয়ে থাকে এবং তারা স্পষ্টই বলে দিয়েছে যে পাইথনের বর্তমান এবং ভবিষ্যৎ হচ্ছে পাইথন ৩

> Python 2.x is legacy, Python 3.x is the present and future of the language

পাইথন ২ আর পাইথন ৩ এর পার্থক্য কি?
পাইথনের এই দুটি প্রধান ভার্সনের মধ্যেকার পার্থক্য এবং আরও বিস্তারিত জানতে পড়া যেতে পারে অফিসিয়াল এই পোস্টটি

ইন্সটলেশন

আমরা নিচে কিছু মেজর অপারেটিং সিস্টেমে পাইথন ৩ এর লেটেস্ট ভার্সন ইন্সটলেশনের ধাপ গুলো সম্বন্ধে জানবো। আগেই বলা হয়েছে, লিনাক্স বা ম্যাকে পাইথনের ২ ভার্সন বিল্ট-ইন অবস্থায় থাকে। তাই সরাসরি এই পাইথনের ইন্টারপ্রেটারকে চালু করতে হলে টার্মিনাল ওপেন করে কমান্ড লিখতে হবে,

```
python
```

এবং এন্টার চাপলেই পাইথন ২ এর ইন্টারপ্রেটার চালু হবে। কিন্তু আমরা এই ভার্সন নিয়ে যেহেতু কাজ করবো না তাই নিচের লেটেস্ট ভার্সন ইন্সটলেশনের দিকে মনোযোগ দেই।

লিনাক্স (উবুন্টু)
উবুন্টুর লেটেস্ট ভার্সনে `Python 3` কেও ইন্সটল্ড অবস্থায় দেখা যায় (যেমন Python 3.4.2) কিন্তু ডিফল্ট হিসেবে সেট করা থাকে না। অর্থাৎ, এই ভার্সনের ইন্টারপ্রেটার চালু করতে টার্মিনালে লিখতে হতে পারে `python3` এবং এন্টার চাপতে হবে।

> দুটি পাইথনের আলাদা আলাদা বাইনারি আলাদা নামে সেইভ থাকে এবং এদের পাথও দেখা যেতে পারে। টার্মিনালে যথাক্রমে `which python` এবং `which python3` কমান্ড ইস্যু করলে যথাক্রমে `/usr/bin/python` এবং `/usr/local/bin/python3` দেখা যাবে। অর্থাৎ ডিফল্ট পাইথন এবং পাইথন 3.4 এর পাথ আলাদা।

যাই হোক, আমরা যদি আরও লেটেস্ট ভার্সনটিকে ইন্সটল করতে চাই তাহলে সরাসরি এই লিঙ্ক থেকে পাইথন 3.5.2 এর Gzipped source tarball ডাউনলোড করে সেটিকে Extract করে নিতে হবে। এতে করে কম্পিউটারে Python-3.5.2 নামের একটি ফোল্ডার তৈরি হবে।

এবার, টার্মিনাল ওপেন করে `cd` কমান্ড ব্যবহার করে ওই ফোল্ডারের লোকেশনে যেতে হবে। যেমন,

```
$ cd ~/Downloads/Python-3.5.2
```

এরপর নিচের কমান্ডটি দিতে হবে,

```
./configure
```

এখন নিচের কমান্ডটি দিন,

```
make
```

এরপর,

```
sudo make install
```

সব কিছু ভালোয় ভালোয় হয়ে গেলে টার্মিনাল ওপেন করে কমান্ড দিন,

```
python3.5
```

নিচের মত আউটপুট আসবে,

```
Python 3.5.2 (default, Jul 22 2016, 18:23:14)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

অর্থাৎ Python 3.5.2 এর কনসোল বা REPL চালু হয়ে গেছে :)

> এই নতুন পাইথনের লোকেশন জানতে `which python3.5` কমান্ড দিয়ে দেখতে পারেন যার আউটপুট আসতে পারে `/usr/local/bin/python3.5`

ম্যাক ওএসএক্স

লিনাক্সের মত ম্যাকেও পাইথন ২ বিল্ট ইন অবস্থায় থাকে। পাইথনের লেটেস্ট ভার্সনটির .pkg ফরম্যাট ডাউনলোড করতে হবে এখানে থেকে.

এরপর ডাউনলোড করা ফাইলে ডাবল ক্লিক করে এবং স্ক্রিনে আগত তথ্য গুলো দেখে দেখে খুব সহজেই গ্রাফিক্যাল মুডে পাইথন ইন্সটল করা যায়।

ইন্সটলেশন কমপ্লিট হলে নতুন পাইথনের পাথ কে সিস্টেমের PATH এনভায়রনমেন্ট ভ্যারিয়েবলে যুক্ত করে নিতে হবে। এ জন্য আপনার ব্যবহৃত শেল প্রোগ্রামের উপর ভিত্তি করে `~/.profile` , `.zshrc` , অথবা `~/.bash_profile` ফাইলকে এডিট করে নিচের লাইনটি জুড়ে দিন।

```
export PATH=$PATH:/Library/Frameworks/Python.framework/Versions/3.5/bin/python3
```

এখন নতুন একটি টার্মিনাল উইন্ডো ওপেন করে কমান্ড দিন,

```
python3
```

নিচের মত আউটপুট তথা REPL চালু হলে ধরে নেয়া যায় পাইথনের লেটেস্ট ভার্সন ইন্সটল হয়েছে,

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  5 2015, 21:12:44)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

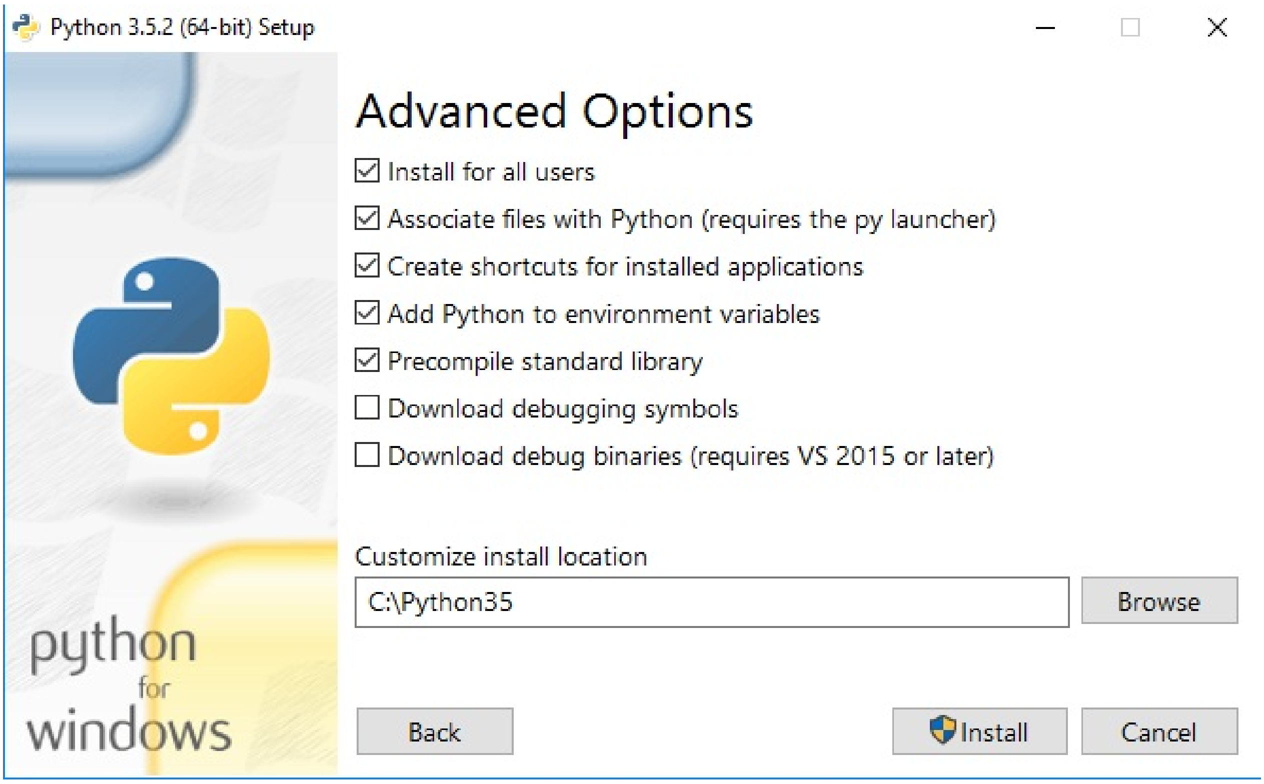
ম্যাকে পাইথন ইন্সটল করার সাথে সাথে একটি IDLE (Integrated Development Environment) -ও ইন্সটল হয়ে যায় যেটা আসলে টার্মিনালের পাইথন REPL (read–eval–print loop) এর মতই কাজ করে কিন্তু বিশেষভাবে পাইথন প্রোগ্রামিং এর জন্যই তৈরি। অ্যাপ লিস্ট থেকে এই নতুন ইন্সটল হওয়া REPL কে খুঁজে চালু করা যাবে।

সাবধানতা
যেহেতু লিনাক্স ও ম্যাকে পাইথন ২ এর বাইনারি বিল্ট-ইন থাকে এবং আলাদাভাবে ইন্সটল করা পাইথন এর বাইনারির নাম সাধারণত `python3.4` বা `python3.5` হয়ে থাকে; তো অনেকেই নতুন ইন্সটল করা পাইথন বাইনারির নাম বদলে বা সিম্বোলিক লিঙ্ক তৈরি করে `python` করে থাকেন যাতে করে টার্মিনালে `python` কমান্ড এক্সিকিউট করলেই পাইথন ৩ এর ইন্টারপ্রেটার চালু হয়। এই কাজটি করা একদম উচিৎ না। কারণ লিনাক্স বা ম্যাকে কিছু টুলস এবং প্রোগ্রাম থাকে যেগুলো ওই সিস্টেমের পাইথন এর উপরেই নির্ভর করে। এখন যখন আপনি পাইথন ৩ এর নাম বদলে শুধু পাইথন করে দিবেন, তারপর থেকে ওই সিস্টেম প্রোগ্রাম গুলো হয়তো সঠিক ভাবে কাজ করবে না। কারন তারা পাইথন ২ এর ইন্টারপ্রেটার কে চেনে পাইথন নামে।

> এসব ছোট জটিলতা সমাধান করা যায় ভার্চুয়াল এনভায়রনমেন্ট তৈরির মাধ্যমে যা কোর্সের শেষের দিকে আলোচনা করা হবে।

উইন্ডোজ
এই অপারেটিং সিস্টেমে বিল্ট-ইন পাইথন না থাকায় অবশ্যই আলাদা ভাবে ইন্সটল করে নিতে হবে। প্রথমে এখান থেকে ৬৪ বিট উইন্ডোজের জন্য অথবা এখান থেকে ৩২ বিট উইন্ডোজের জন্য ইন্সটলার ডাউনলোড করে নিন। ম্যাক এর ইন্সটলারের মতই উইন্ডোজ এর জন্য ইন্সটলারটিও গ্রাফিক্যাল ইন্টারফেস ভিত্তিক অর্থাৎ, মাউস এর কয়েকটি ক্লিক দিয়েই পাইথন ইন্সটল করে নিতে পারেন।

ইন্সটলারটি ওপেন হলে 'customize installation' সিলেক্ট করুন। তারপরে 'Optional features' স্ক্রিনে সবগুলো চেকবক্সই সিলেক্টেড রেখে দিতে পারেন। তারপরে 'Advanced option' স্ক্রিন থেকে প্রয়োজনীয় চেকবক্সগুলো সিলেক্ট করে দিন (বিশেষ করে 'Install for all users', 'Add python to environment variables' এবং 'Precompile standard library')।



এখান থেকে আপনি পাইথনের ইন্সটলেশন লোকেশনও চেঞ্জ করে দিতে পারেন। সাধারণত সবাই ইন্সটলেশন লোকেশন হিসেবে `C:\Python3x` ব্যবহার করে থাকে। এরপরে 'Install' বাটন চেপে ইন্সটলেশন কমপ্লিট করুন।

এই ইন্সটলেশনেও একটি গ্রাফিক্যাল পাইথন কনসোল প্রোগ্রাম ইন্সটল হয়ে যায় যাকে আমরা IDLE বলছি। Start মেনু থেকে All Programs এর মধ্যে Python 3.5 নামক ফোল্ডারের মধ্যে IDLE নামের প্রোগ্রামটি থাকবে যেখান থেকে একে চালু করা যেতে পারে।

যদি আপনি উপরের মত করে ইন্সটলেশনে 'Add python to environment variables' অপশন সিলেক্ট করে থাকেন তাহলে আপনি উইন্ডোজের ডিফল্ট কমান্ড প্রম্পট প্রোগ্রামের মধ্যেই পাইথন ইন্টারপ্রেটার ব্যবহার করতে পারবেন (লিনাক্স বা ম্যাকের টার্মিনালের মত করে)।

আর যদি না করে থাকেন তাহলে পাইথন ডিরেক্টরীকে সিস্টেম পাথে যোগ করে নিন । অর্থাৎ `C:\Python3x` (ধরে নিচ্ছি আপনার পাইথন ইন্সটলেশন সি ড্রাইভের মধ্যেই করেছেন) এই লোকেশনটিকে আপনার `PATH` ভ্যারিয়েবলে যোগ করে নিন।

নীচের মত করেঃ

- `My Computer` এর উপর রাইট ক্লিক করে `Properties` এ যান ।
- বাম পাশে `Advanced System Settings` এ ক্লিক করূন ।
- নিচের দিকে থাকা `Environment Variables` এ ক্লিক করুন ।
- `System Variables` এর ভিতরে `PATH` এন্ট্রি খুজে বের করে `Edit` বাটন চাপুন ।
- এবার এর শেষে `C:\Python3x;` লিখে `OK` করে বের হয়ে আসুন ।
- কমান্ড প্রম্পট চালু করুন (cmd.exe) । টাইপ করুনঃ `python` । এন্টার চাপুন ।

কমান্ড প্রম্পট এর কালো পর্দায় নিচের মত লেখা দেখাবেঃ

```
Python 3.5.1 (v3.5.1:xxxxxxxx, Sep 13 2015, 15:10:54) [MSC v.1900 32 bit (Intel)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

এরকম দেখালে বোঝা গেল আমরা পাইথন ইন্সটলেশন শেষে এটাকে রান করাতে পেরেছি কমান্ড লাইনে।

প্রথম প্রোগ্রাম
উপরের যেকোনো একটি মাধ্যমে যদি কোন ভাবে আপনি পাইথন কনসোল ওপেন করে থাকেন তাহলে নিচের লাইনটি সেখানে লিখুন এবং এন্টার চাপুন,

```
>>> print('Hello world!')
```

তাহলে তারপরের লাইনেই আউটপুট পাবেন নিচের মত,

```
Hello world!
```

> `>>>` চিহ্নটির মানে হচ্ছে পাইথন ইন্টারপ্রেটার আপনার কাছে পাইথন স্টেটমেন্ট নেয়ার জন্য প্রস্তুত এবং এখানেই আপনি লিখতে পারবেন।

এই সেকশনে থাকছে

- সাধারণ কিছু অপারেশন
- আরও কিছু নিউমেরিক অপারেশন
- স্ট্রিং
- ব্যাসিক ইনপুট আউটপুট
- স্ট্রিং অপারেশন
- টাইপ কনভার্সন
- ভ্যারিয়েবল
- ইনপ্লেস অপারেটর
- এডিটর এর ব্যবহার

ব্যাসিক অপারেশন

পাইথনের কনসোলে সহজেই ম্যাথেম্যাটিক্যাল ক্যালকুলেশন করা যায়। তাই আবার খুলে ফেলুন পাইথন ইন্টারপ্রেটার। অর্থাৎ, নিচের যেকোনো একটিঃ

১) পাইথন ইন্সটলেশনের সাথে আশা IDLE
২) লিনাক্স বা ম্যাক হলে Terminal ওপেন করুন এবং টাইপ করুন python3
৩) উইন্ডোজ হলে Command Prompt চালু করুন এবং টাইপ করুন python

কনসোলে নিচের মত ম্যাথেম্যাটিক্যাল কমান্ড লিখে সহজেই সেগুলোর রেজাল্ট পাওয়া যায় -

```
>>> 2 + 6
8
>>> 5 + 4 - 3
6
```

যোগ বিয়োগের মতই গুন ভাগের কাজও এখানে সহজেই করা যায়। ব্রাকেট ব্যবহার করে নির্ধারণ করে দেয়া যায় যে, কোন পার্ট টুকুর অপারেশন আগে করা হবে।

```
>>> 2 * (2 + 2)
8
>>> 20/2
10.0
```

একটি সিঙ্গেল `/` ব্যবহার করে ভাগ করলে রেজাল্ট আসে `float` টাইপের ডেসিম্যাল।

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

নাম্বারের আগে মাইনাস (-) সাইন দিয়ে নেগেটিভ নাম্বার নির্ধারণ করে দেয়া হয়।

সাধারণ গণিতের মতই পাইথনে কোন সংখ্যাকে শূন্য দিয়ে ভাগ করতে গেলে এরর আসবে,

```
>>> 44/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Float

যে নাম্বার গুলো Integer টাইপের নয় সেগুলোকে পাইথনে রিপ্রেজেন্ট করার জন্য float ব্যবহার করা হয়। যেমনঃ 1.0, -5.15 ইত্যাদি। যেকোনো সংখ্যার মধ্যে একটি দশমিক চিহ্ন ব্যবহার করা মানেই হল সেটি একটি float টাইপের ডাটা হয়ে যায়। অথবা পাইথনে যেকোনো দুটি ইন্টিজার টাইপের সংখ্যাকে ভাগ করলেই একটি float টাইপের রেজাল্ট পাওয়া যায়। যেমন,

```
>>> 10/5
2.0
```

একটা কথা মনে রাখা জরুরি - মানুষের মত কম্পিউটারও শতভাগ সঠিকভাবে float টাইপের ডাটা স্টোর করতে পারে না। যেমন 1/3 এর ফলাফল হচ্ছে 0.333333333 (চলতেই থাকে)। এরকম অবস্থা কিছু অনাকাঙ্ক্ষিত ত্রুটির কারণ হয়ে দাঁড়াতে পারে।

```
>>> 8 / 2
4.0
>>> 6 * 7.0
42.0
>>> 4 + 1.65
5.65
```

আরও কিছু নিউমেরিক অপারেশন

যোগ, বিয়োগ, গুন ভাগ বাদেও পাইথনে এক্সপনেন্সিয়েশন এর সাপোর্ট আছে যাকে আমরা একটি সংখ্যার উপর আরেকটা সংখ্যার পাওয়ার বলে থাকি। দুটো ** চিহ্ন দিয়ে এই অপারেশন করা হয়। যেমন -

```
>>> 2 ** 3
8
>>> 3 ** 3
27
```

শুধুমাত্র ভাগফল (quotient) নির্ণয়ের জন্য `floor division` এবং ভাগশেষ নির্ণয়ের জন্য `modulo operator` ব্যবহার করা হয়। দুটো ফরওয়ার্ড স্ল্যাস `//` ব্যবহার করে `floor division` করা হয় আর `%` সিম্বল দিয়ে `modulo operator` এর কাজ করা হয়। নিচের উদাহরণটি দেখি -

```
>>> 10 // 3
3
>>> 10 % 3
1
```

এখানে ৩ দিয়ে ১০ কে ভাগ করলে পূর্ণ ভাগফল আসে ৩ এবং ১০ কে ৩ দিয়ে ভাগ করলে ভাগশেষ থাকে ১

স্ট্রিং

পাইথনে খুবই গুরুত্বপূর্ণ ডেটা টাইপ হলো স্ট্রিং । একগুচ্ছ ক্যারেক্টার বা কিছু ওয়ার্ডের সিকুয়েন্সকে সাধারণত স্ট্রিং বলা হয়ে থাকে। পাইথনে যে কোন সেনটেন্সকেই স্ট্রিং হিসেবে ব্যবহার করা যায় সিঙ্গেল(' '), ডাবল(" ") কিংবা ট্রিপল(""" """) কোটেশন এর মাধ্যমে। আমাদের পাইথন কনসোলে যদি নিচের মত করে বাক্য লিখে এন্টার চাপি তাহলে আউটপুটে সেই বাক্যকে দেখতে পারবো।

```
>>> "We love python!"
'We love python!'
>>> 'The most popular general purpose programming language'
'The most popular general purpose programming language'
```

> লক্ষণীয়, ইনপুট দেয়ার সময় ডাবল বা সিঙ্গেল কোটেশন যাই ব্যবহার করা হোক না কেন, আউটপুটের সময় সিঙ্গেল কোট দিয়ে সেই স্ট্রিং কে দেখায়।

কিছু ক্যারেক্টারকে সরাসরি একটি স্ট্রিং এর মধ্যে ব্যবহার করা যায় না। যেমন, ডাবল কোট দিয়ে নির্দেশ করা একটি স্ট্রিং তথা বাক্যের মধ্যে ডাবল কোট থাকতে পারে না। এতে করে পাইথন এরর দিবে। এক্ষেত্রে এরকম ক্যারেক্টার গুলোর সামনে একটি ব্যাকস্ল্যাস (`\`) চিহ্ন দিয়ে এস্কেপ করা হয়ে থাকে। যেমন,

```
>>> 'Brian\'s mother: He\'s not the Messiah. He\'s a very naughty boy!'
'Brian's mother: He's not the Messiah. He's a very naughty boy!'
```

নিউ লাইন ক্যারেক্টার (`\n`), ব্যাকস্ল্যাস ক্যারেক্টার (`\`), ট্যাব, ইউনিকোড ক্যারেক্টার - এদেরকেও এস্কেপ করে স্ট্রিং এর মধ্যে ব্যবহার করতে হয়।

পাইথনে নিউলাইন ক্যারেক্টারকে ম্যানুয়ালি লেখার দরকার পরে না যদি একাধিক লাইন সম্বলিত সেই স্ট্রিং বা বাক্যকে তিনটি করে কোটেশন এর মধ্যে ডিফাইন করা হয়। নিচের উদাহরণটি দেখি,

```
>>> """Me: Hi, there!
... She: Yes, please!"""
'Me: Hi, there!\nShe: Yes, please!'
>>>
```

উপরে, দুই লাইন ওয়ালা একটি স্ট্রিংকে ইনপুট হিসেবে দিয়েছি এবং আউটপুটে দেখা যাচ্ছে সে স্ট্রিং এর মধ্যে যেখানে নতুন লাইন দরকার সেখানে পাইথন স্বয়ংক্রিয় ভাবে `\n` ক্যারেক্টার বসিয়ে দিয়েছে।

স্পেশাল ক্যারেক্টার এবং এস্কেইপ সিকুয়েন্স
কিছু প্রচলিত এস্কেইপ সিকুয়েন্স নিচে দেওয়া হলো -

সিকুয়েন্স	পরিচিতি
`\\`	একটা ব্যাকস্ল্যাশ
`\'`	সিঙ্গল কোট (')
`\"`	ডাবল কোট (")
`\a`	বেল
`\b`	ব্যাকস্পেইস
`\f`	ফর্মফিড
`\n`	লাইন ব্রেক
`\N{name}`	ইউনিকোড ক্যারেক্টার এর নাম
`\r ASCII`	ক্যারিজ রিটার্ন (ম্যাক ওস এক্স এ নিউ লাইন ক্যারেক্টার)
`\t`	ট্যাব
`\uxxxx`	১৬ বিট হেক্সাডেসিম্যাল ভ্যালু সম্বলিত ইউনিকোড ক্যারেক্টার
`\Uxxxxxxxx`	৩২ বিট হেক্সাডেসিম্যাল ভ্যালু বিশিষ্ট ইউনিকোড ক্যারেক্টার
`\v`	ভার্টিক্যাল ট্যাব
`\ooo`	`ooo` অক্টাল ভ্যালু বিশিষ্ট ক্যারেক্টার
`\xhh`	`hh` হেক্সাডেসিম্যাল ভ্যালুওয়ালা ক্যারেক্টার

(এই টেবিল টি জেড শ এর লার্ন পাইথন দ্যা হার্ড ওয়ে বইটি থেকে অনুবাদকৃত)

ব্যাসিক ইনপুট আউটপুট

পাইথনে ডাটা ইনপুট এবং আউটপুট এর জন্য আমরা স্পেশাল কিছু ফাংশন ব্যবহার করে থাকি। ইনপুট এবং আউটপুট এর বেসিক ফাংশন গুলো হল :

- `input()`
- `print()`

ইনপুট ফাংশন **`input()`**

এই ফাংশনকে আমরা সাধারণত ইউজার এর কাছ থেকে ইনপুট নেয়ার জন্য ব্যবহার করে থাকি। উদাহরণ দিলে ব্যাপারটা আরো পরিষ্কার হবে আশা করি।

```
>>> input("Give me your country name: ")
Give me your country name: Bangladesh
'Bangladesh'
```

এ ক্ষেত্রে পাইথন ইন্টারপ্রেটার যখন কোডটা এক্সিকিউট করবে তখন ইন্টারপ্রেটার ইউজার এর ইনপুট এর জন্য অপেক্ষা করবে এবং ততক্ষণ পর্যন্ত ইনপুট এর ডাটা গ্রহণ করবে যতক্ষণ না ইউজার `Enter` বাটন প্রেস করে অথবা ইনপুটে আরেকটি নিউলাইন ক্যারেক্টার আসে। যদি কনসোলে উপরের স্টেটমেন্টটি এক্সিকিউট করা হয় তাহলে এন্টার চাপলে নিচের লাইনে `'Bangladesh'` আউটপুট হিসেবে আসছে যা ইউজারের কাছ থেকে ইনপুট নেয়া হয়েছিল।

আউটপুট ফাংশন **`print()`**

`print()` ফাংশন কেবলমাত্র তাই আউটপুট দেয় যা এর আর্গুমেন্ট হিসেবে দেয়া হয়। কয়েকটি উদাহরণ দেখলে আউটপুট এর ব্যাপারটি আরো সহজ হবে।

```
>>> print('Hello World!')
Hello World!
```

উপরের প্রোগ্রামের আউটপুট হবে যথারীতি `Hello World!` ।

```
>>> print(5+5)
10
```

```
>>> print('Name: Bangladeash\nPopulation: 156.6M')
Name: Bangladeash
Population: 156.6M
```

স্ট্রিং অপারেশনস

কনক্যাটেনেশন (Concatenation)

ইন্টিজার বা ফ্লটের মত, স্ট্রিংকেও যোগ করা যায় যাকে কনক্যাটেনেশন বলা হয়।

```
>>> "Spam" + 'eggs'
'Spameggs'
```

```
>>> print("First string" + ", " + "second string")
First string, second string
```

তাই বলে কোন নাম্বারের সাথে স্ট্রিং যোগ করা যাবে না,

```
>>> "2" + "2"
'22'
>>> 1 + '2' + 3 + '4'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

রিপিটেশন (Repetition)

যোগের মত স্ট্রিং নিয়ে গুনও করা যায়, একে রিপিটেশন বলে। তবে এই গুন হতে হবে একটি স্ট্রিং এর সাথে একটি ইন্টিজার নাম্বারের। স্ট্রিং এবং স্ট্রিং এর মধ্যে নয় অথবা ফ্লট টাইপের ডাটার সাথে নয়।

উদাহরণ,

```
>>> print("spam" * 3)
spamspamspam

>>> 4 * '2'
'2222'

>>> '17' * '87'
TypeError: can't multiply sequence by non-int of type 'str'

>>> 'pythonisfun' * 7.0
TypeError: can't multiply sequence by non-int of type 'float'
```

স্ট্রিং ফরম্যাটিং

নন স্ট্রিং ডাটার সাথে স্ট্রিং টাইপের ডাটাকে যুক্ত করে সুন্দর স্ট্রিং আউটপুট তৈরি করতে `format` মেথড ব্যবহার করা হয়। এর মাধ্যমে একটি স্ট্রিং এর মধ্যে থাকা কিছু আর্গুমেন্টকে রিপ্লেস বা সাবস্টিটিউট করা যায়। `format`

মেথডের মধ্যের প্রত্যেকটি আর্গুমেন্ট দিয়ে এর সামনে থাকা স্ট্রিং এর মধ্যের প্লেস হোল্ডার গুলোকে রিপ্লেস করা হয়। প্লেস হোল্ডার গুলো `{}` এর সাথে ইনডেক্স বা নাম ব্যবহার করে ডিফাইন করা হয়। একটি উদাহরণ দেখলেই বিষয়টি পরিষ্কার হয়ে যাবে -

```
msg = "My self score on PHP: {0}, Python: {1}, Java: {2}, Swift: {3}". format(6, 6.5, 5,

print(msg)
```

আউটপুট,

```
My self score on PHP: 6, Python: 6.5, Java: 5, Swift: 6
```

ফরম্যাটিংয়ের সময় ইন্ডেক্সগুলো 0, 1, 2.... এইভাবে সিরিয়ালি দিতে হবে ব্যাপারটা কিন্তু এমন না। ইচ্ছে করলেই এগুলো আগে পরে কিংবা একাধিকবার করেও দেয়া যায়।

```
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')
'abracadabra'
```

`format` মেথডের মধ্যে নাম ওয়ালা আর্গুমেন্ট পাঠিয়ে এবং স্ট্রিং এর মধ্যের প্লেস হোল্ডার গুলোতে সেই নামে সেগুলোকে ব্যবহার করেও কাজ করা যায় -

```
message = "If x = {x} and y = {y}, then x+y = {z}".format(x = 20, y = 300, z = 20+300)

print(message)
```

আউটপুট,

```
If x = 20 and y = 300, then x+y = 320
```

কিছু গুরুত্বপূর্ণ ফাংশন

নিচে স্ট্রিং নিয়ে কাজ করার জন্য বেশ কিছু গুরুত্বপূর্ণ এবং উপকারী ফাংশনের উদাহরণ দেয়া হল -

```
print(", ".join(["apple", "orange", "pineapple"]))
#prints "apple, orange, pineapple"
```

`join` মেথড একটি স্ট্রিং ওয়ালা লিস্টের (লিস্ট নিয়ে পরবর্তীতে আলোচনা করা হয়েছে) স্ট্রিং গুলোকে একত্রিত করে কিন্তু মাঝখানে নির্ধারিত একটি সেপারেটর ব্যবহার করে। যেমন উপরের উদাহরণে, `apple` , `orange` , `pineapple` এই তিনটি ভ্যালুকে একত্রিত করা হয়েছে কিন্তু তাদের মধ্যে কমা `,` সেপারেটর ব্যবহার করে।

```
print("Hello ME".replace("ME", "world"))
#prints "Hello world"
```

`replace` মেথডের মাধ্যমে একটি সাব স্ট্রিং কে খুঁজে সেখানে অন্য কিছু রিপ্লেস করা যায়। যেমন উপরের উদাহরণে - `ME` রিপ্লেস করে `world` বসানো হয়েছে।

```
print("This is a sentence.".startswith("This"))
# prints "True"

print("This is a sentence.".endswith("sentence."))
# prints "True"
```

`startswith` , `endswith` মেথডের মাধ্যমে কোন একটি ব্যাক্যর শুরু বা শেষ নির্দিষ্ট কোন সাবস্ট্রিং দিয়ে হয়েছে কিনা তা চেক করা যায়।

```
print("This is a sentence.".upper())
# prints "THIS IS A SENTENCE."

print("AN ALL CAPS SENTENCE".lower())
#prints "an all caps sentence"
```

`upper()` মেথড স্ট্রিংয়ের সবগুলো ক্যারেক্টারকে `uppercase` এ পরিবর্তিত করে। একইভাবে `lower()` মেথড ট্রিংয়ের সবগুলো ক্যারেক্টারকে `lowercase` এ পরিবর্তিত করে।

```
print("a, e, i, o, u".split(", "))
#prints "['a', 'e', 'i', 'o', 'u']"
```

`split` মেথড হচ্ছে `join` মেথডের উল্টো। অর্থাৎ একটি বাক্যেকে নির্দিষ্ট কোন সেপারেটর এর সাপেক্ষে ভেঙ্গে একটি লিস্ট তৈরি করা যায় এই মেথডের মাধ্যমে। সেটাই দেখানো হয়েছে উপরের উদাহরণে।

ডাটাটাইপ কনভার্সন

ডাটাটাইপ কনভার্সন বলতে ভ্যারিয়েবল কে এক টাইপ থেকে অন্য টাইপ এ কনভার্ট করা বুঝায়। একে টাইপ কাস্টিং ও বলা হয়ে থাকে। পাইথনে টাইপ কাস্টিং এর জন্যে কিছু বিল্টইন ফাংশন বানানো আছে। আমরা চাইলে সহজেই সেগুলো ব্যবহার করতে পারি। এখন পর্যন্ত আমরা integers, floats, এবং strings ডাটাটাইপ সম্পর্কে জেনেছি। এই টাইপে কনভার্ট করার জন্য ফাংশন গুলো যথাক্রমে হচ্ছে - `int()` , `float` , `str()` .

ইন্টেজার এ কনভার্সন
স্ট্রিং অথবা ফ্লোট থেকে ইন্টেজার এ কনভার্ট করার জন্য `int()` ফাংশন ব্যাবহার করা হয়।

```
# String to Integer Conversion
>>> int("123")
123

# float to Integer Conversion
>>> int(12.3)
12
```

বিঃ দ্রঃ স্ট্রিং থেকে ইন্টেজার এ কনভার্ট এর সময় খেয়াল রাখতে হবে স্ট্রিং এ যাতে কোনো ননিনিউমেরিক ক্যারেকটার না থাকে।

```
>>> int("123a")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '123a'
```

ফ্লোট এ কনভার্সন
স্ট্রিং অথবা ইন্টেজার থেকে ফ্লোট এ কনভার্ট করার জন্য `float()` ফাংশন ব্যাবহার করা হয়।

```
# String to float Conversion
>>> float("123.456")
123.456

# Integer to float Conversion
>>> float(123)
123.0
```

বিঃ দ্রঃ এক্ষেত্রেও স্ট্রিং থেকে ফ্লোট এ কনভার্ট এর সময় খেয়াল রাখতে হবে স্ট্রিং এ যাতে কোনো ননিনিউমেরিক ক্যারেকটার না থাকে এবং একাধিক দশমিক পয়েন্ট না থাকে।

এবার একটু ভাবুনতো স্ট্রিং এর ভেতর যদি দশমিকযুক্ত সংখ্যা থাকে এবং তা ইন্টেজার এ কনভার্ট করার প্রয়োজন হয় তাহলে কি `int()` ফাংশন ব্যাবহার করলেই হবে ? উত্তর হবে না । সেক্ষেত্রে স্ট্রিং কে প্রথমে ফ্লোট এ এবং ফ্লোটকে ইন্টেজার এ কনভার্ট করতে হবে।

```
>>> float("123.456")
123.456
>>> int(123.456)
123
```

```
>>> int("123.456")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '123.456'
```

স্ট্রিং এ কনভার্সন

যে কোন ভ্যারিয়েবল স্ট্রিং -এ কনভার্ট করার জন্য কোনো প্রকার বিধিনিষেধ ছাড়াই `str()` ফাংশন ব্যবহার করবো।

```
>>> str(123)
'123'
```

আমরা যখন `print()` ফাংশন এর ভেতর একাধিক ভ্যারিয়েবল লিখি তখন স্ট্রিং কনভার্সন ব্যবহার করতে হয়।

```
>>> print("Float = " + str(10.5) + " Integer = " + str(50))
Float = 10.5 Integer = 50
```

ভ্যারিয়েবল

ভ্যারিয়েবল হচ্ছে কম্পিউটার মেমোরিতে তৈরি হওয়া ছোট ছোট বাক্সের মতো যার ভেতর যে কোন কিছু জমা করে রাখা যায়। যখন আমরা ভ্যারিয়েবল ডিক্লেয়ার করি তখন কম্পিউটার সেই ভ্যারিয়েবলের জন্য কিছু নির্দিষ্ট মেমোরি নির্ধারন করে দেয়। প্রতিটি ভ্যারিয়েবল এর মেমোরি অ্যাড্রেস ইউনিক হয়। প্রোগ্রামের প্রয়োজনে ওই ভ্যারিয়েবল তথা নাম সম্পন্ন মেমোরি লোকেশনে ভ্যালু জমা করে রাখা যায়। আবার প্রয়োজনের সময় সেই নাম ব্যবহার করে ওই লোকেশনের ভ্যালুকে অ্যাক্সেস করা যায় এবং কাজে লাগানো যায়।

একটি ভ্যারিয়েবলের মধ্যে কোন ভ্যালু জমা রাখার জন্য একটি সমান (=) চিহ্ন ব্যবহার করা হয়। এর আগ পর্যন্ত চ্যাপ্টার গুলোতে আমরা কোন ভ্যারিয়েবল ব্যবহার করি নি। তাই যখনই আমরা পাইথন কনসোলে কোন নাম্বার, টেক্সট অথবা স্টেটমেন্ট লিখে এন্টার কি প্রেস করেছি তখনি সেটার আউটপুট পরবর্তী লাইনে দেখিয়েছে। কিন্তু যদি আমরা কোন ভ্যালু কোন একটা ভ্যারিয়েবলে স্টোর করি (সমান চিহ্ন দিয়ে) এবং এন্টার প্রেস করি তখন কিন্তু পরের লাইনে আউটপুট আসবে না। বরং সমান চিহ্নের ডান পাশের ভ্যালুটি সমান চিহ্নের বাম পাশের ভ্যারিয়েবলে জমা হয়ে যাবে যেটাকে আমরা পরবর্তী স্টেটমেন্টে নাম উল্লেখপূর্বক ব্যবহার করতে পারবো।

অ্যাসাইনমেন্ট

```
>>> x = 7
>>> print(x)
7
>>> print(x + 3)
10
>>> print(x)
7
```

উপরের উদাহরণে, প্রথমে একটি ভ্যারিয়েবল `x` এর মধ্যে `7` (একটি ইন্টিজার নাম্বার) কে জমা রাখা হয়েছে। এরপরের লাইনে `print()` ফাংশনের আর্গুমেন্ট হিসেবে সেই `x` কেই পাঠানো হয়েছে। আর আমরা আগেই জেনেছি যে, প্রিন্ট ফাংশনের আর্গুমেন্ট হিসেবে কিছু পাঠালে তা প্রিন্ট হয়। তাই, `x` তথা `7` নাম্বারটি স্ক্রিনে প্রিন্ট হয়েছে। একই রকম কাজ করা হয়েছে `print(x + 3)` লাইনে। এখানে মূলত `print(7 + 3)` এই স্টেটমেন্টটি এক্সিকিউট হয়েছে, কারন `x` এর মান তো `7`. আর তার সাথে `3` যোগ হয়ে `10` নাম্বারটিই প্রিন্ট ফাংশনের আর্গুমেন্ট হিসেবে পাঠানো হয়েছে যা স্ক্রিনে প্রিন্ট হয়েছে।

> শেষের প্রিন্ট স্টেটমেন্টও আমরা `x` এর মান তথা `7` কে প্রিন্ট করতে পেরেছি ভ্যারিয়েবলের নাম দিয়েই। তাই বলা যায়, একটি ভ্যারিয়েবল গোটা প্রোগ্রাম জুড়ে এর মান স্টোর করে রাখে।

রি-অ্যাসাইনমেন্ট
এবার আমরা নিচের উদাহরণটি দেখি,

```
>>> x = 10.5
>>> print(x)
10.5
>>> x = "Hello There"
>>> print(x)
Hello There
```

এখানে প্রথমে `x` এর মান হিসেবে একটি ফ্লট জমা রাখা হয়েছে এবং সাধারণ ভাবেই প্রিন্ট করে তার ভ্যালু পাওয়া গেছে। কিন্তু পরের লাইনে সেই একই `x` এর মধ্যে একটি স্ট্রিং জমা রাখা হয়েছে এবং সেটিকেও পরবর্তী প্রিন্ট ফাংশনের মাধ্যমে অ্যাক্সেস করে স্ক্রিনে প্রিন্ট করা গেছে। একে ভ্যালু রি-অ্যাসাইনমেন্ট বলা হয়। অর্থাৎ, একটি ভ্যারিয়েবলের মধ্যে একাধিক বার নতুন নতুন ভ্যালু জমা রাখা যায় এবং সর্বশেষ স্টোর করা ভ্যালুটিই ওই ভ্যারিয়েবলের মধ্যে জমা থাকে (আগের ভ্যালুটি মুছে যায়)।

> পাইথনে ভ্যারিয়েবলের কোন নির্দিষ্ট ডাটা টাইপ নেই। তাই একই ভ্যারিয়েবলে প্রথমে একটি নাম্বার এবং পরবর্তীতে সেটাতে একটি স্ট্রিং জমা রাখা গেছে।

খেয়াল রাখতে হবে যে, আমরা প্রথমবার যখন কোনো একটা ভ্যারিয়েবলে কোনো ভ্যালু এসাইন করি তখন সেই ভ্যারিয়েবলটা initialize হয়। পরবর্তীতে আমরা ঐ ভ্যারিয়েবলটাতেই আবার বিভিন্ন ভ্যালু রেখে কাজ করতে পারবো। কিন্তু যদি আমরা ভ্যালু এসাইন করে initialize করিনি এই রকম কোনো ভ্যারিয়েবল নিয়ে কাজ করার চেষ্টা করি তাহলে এরর দিবে।

```
>>> variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'variable' is not defined
```

নামকরণ এর নিয়ম

পাইথনে ভ্যারিয়েবল লেখার সময় কিছু নিয়ম মেনে ভ্যারিয়েবল ডিফাইন করা হয়।

- ভ্যারিয়েবলের নাম অবশ্যই এক শব্দের হতে হবে। অর্থাৎ এরকম ভাবে ভ্যারিয়েবল লেখা যাবে নাঃ `my variable = 10`
- প্রথম অক্ষর অবশ্যই একটি alphabetic letter(uppercase or lowercase) অথবা underscore (`_`) হতে হবে । যেমনঃ `nafis` , `a` , `b` , `_variable` লেখা যাবে ভ্যারিয়েবল হিসেবে কিন্তু `1name` , `@nafis` , `7a` , `%b` এই ভাবে লেখা যাবে না । প্রথম অক্ষর ছাড়া পরে letter, underscore, number ব্যবহার করা যাবে। যেমনঃ `variable1` , `my_variable` , `not_Very_Good_Name10` যদিও ভ্যারিয়েবলের শুরুতে underscore ব্যবহার করা যায়, কিন্তু পাইথনের কনভেনশন হচ্ছে ভ্যারিয়েবলের নাম সবসময় lowercase letter দিয়ে শুরু করা।
- পাইথন Case Sensitive অর্থাৎ `a = 4` এবং `A = 4` একই ভ্যারিয়েবল না।
- পাইথনের কিছু reserved কী-ওয়ার্ড আছে, এগুলো ব্যবহার করা যাবে না । যেমনঃ `if` , `else` , `elif` , `for` , `while` , `break` , `continue` , `except` , `as` , `in` , `is` , `True` , `False` , `yield` , `None` , `def` , `del` , `class` ইত্যাদি।

উদাহরণ,

```
>>> this_is_a_normal_name = 7

>>> 123abc = 7
SyntaxError: invalid syntax

>>> spaces are not allowed
SyntaxError: invalid syntax
```

রিভিউ

আমরা আগের চ্যাপ্টারে ইনপুট, আউটপুট ফাংশন এর ব্যবহার দেখেছি। এই চ্যাপ্টারে দেখলাম ভ্যারিয়েবল। নিচের উদাহরণে আমরা দেখবো কিভাবে সেই ফাংশন ব্যবহার করে এবং একটি ভ্যারিয়েবল ব্যবহার করে ইউজার এর ইনপুট মেমোরিতে জমা রাখা যায় এবং পরবর্তীতে ব্যবহার করা যায়,

```
>>> user_input = input("Enter your birth year: ")
Enter your birth year: 1987
>>> age = 2016 - int(user_input)
>>> print("You are " + str(age) + " years old!")
You are 29 years old!
```

এখানে ইনপুট ফাংশন, ভ্যারিয়েবল, ব্যাসিক অপারেশন, টাইপ কনভার্সন এবং আউটপুট ফাংশনের ব্যবহার করা হয়েছে। খুব সহজ মনে হচ্ছে, তাই নয় কি?

পাইথন যখনই `input()` ফাংশন এক্সিকিউট করছে তখন সে স্ক্রিনে ম্যাসেজটি প্রিন্ট করে ইউজারের ইনপুটের জন্য অপেক্ষা করছে। ইউজার ইনপুট দিলে তথা এন্টার চাপলে তার দেয়া ইনপুটটি `user_input` ভ্যারিয়েবলে জমা থাকছে। এরপর আমরা `2016` থেকে সেই ভ্যালুকে বিয়োগ করার সময় `int()` ফাংশন ব্যবহার করে `user_input` এর ভ্যালুকে ইন্টিজার নাম্বারে কনভার্ট করে নিয়েছি (কারণ আমরা জানি ইউজার ইনপুট স্ট্রিং হিসেবে জমা থাকে অন্যদিকে একটি নাম্বার থেকে একটি স্ট্রিং এর বিয়োগ সম্ভব না)। বিয়োগের ফল `age` ভ্যারিয়েবলে জমা রাখছি। এরপর `print()` ফাংশনের মধ্যে আমরা প্রথমে `age` কে স্ট্রিং -এ কনভার্ট করেছি এবং অন্য দুটি স্ট্রিং এর সাথে কনক্যাটেনেশন (যোগ) করেছি।

ইন প্লেইস অপারেটর

ইনপ্লেইস অপারেশন আসলে কি? আসুন একটা ছোট্ট উদাহরন দেখে নেই -

```
a = 3
a += 2
```

এখানে `+=` টাই ইনপ্লেইস অপারেটর । এখানে আমরা প্রথমে `a` এর সাথে 2 যোগ করি এবং সেই ভ্যালুটা দিয়ে `a` এর আগের মানটা আপডেইট করে দেই । অর্থাৎ তৃতীয় লাইনে যদি আমরা লিখি, `print(a)` তাহলে এর আউটপুট আসবে `5`

এই ব্যাপারটি অন্যান্য অপারেটর গুলোর ক্ষেত্রেও প্রযোজ্য। কমন ইনপ্লেইস অপারেটর:

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`

শুধুমাত্র নাম্বার বাদেও অন্যান্য টাইপের ক্ষেত্রেও ইন প্লেইস অপারেটর ব্যবহার করা যায় যেমন, স্ট্রিং এর ক্ষেত্রে,

```
language = "Python"
language += "3"
print(language)
```

আউটপুট,

```
Python3
```

সংকলন - আবু আশরাফ মাসনুন

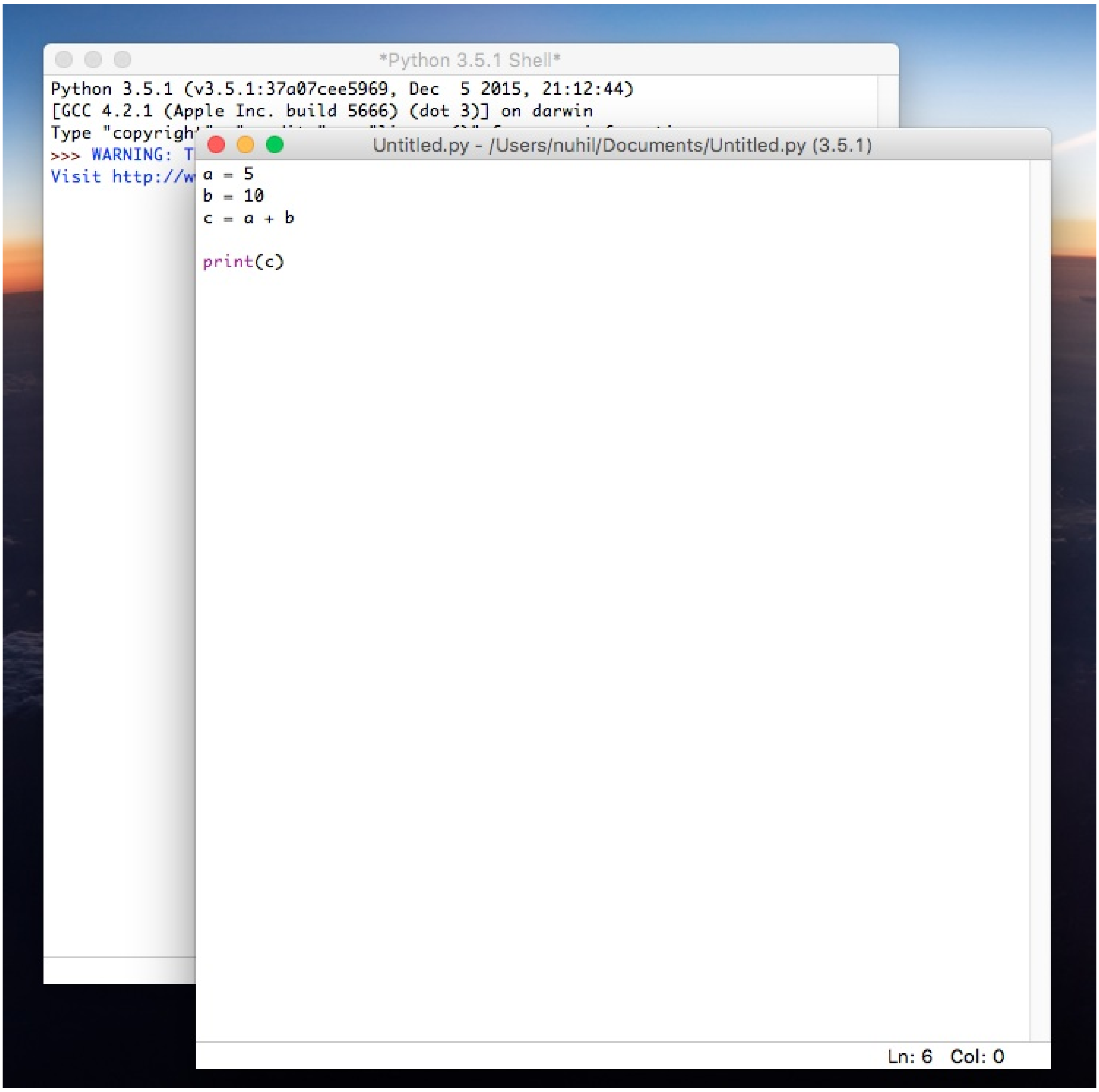
এডিটর এর ব্যবহার

এখন পর্যন্ত আমরা পাইথনের কনসোলে স্টেটমেন্ট লিখে তা পাইথনের ইন্টারপ্রেটারের মাধ্যমে এক্সিকিউট করে আউটপুট দেখেছি। এই পদ্ধতির অসুবিধা হচ্ছে প্রতিবার একটি করে লাইন লেখা যায় এবং এক্সিকিউট করা যায়। কিন্তু বাস্তব একটি প্রোগ্রাম লেখার সময় সেটি অনেক লাইন তথা ফাংশন, কন্ট্রোল, লজিক মিলে হতে পারে। এ জন্য পাইথন প্রোগ্রামকে একটি ফাইলে লিখে সেই ফাইলকে `.py` এক্সটেনশনে সেইভ করে পরবর্তীতে একই পাইথন ইন্টারপ্রেটারের মাধ্যমে রান করানো যায়। এর জন্য দরকার ভালো একটি এডিটর বা আইডিই।

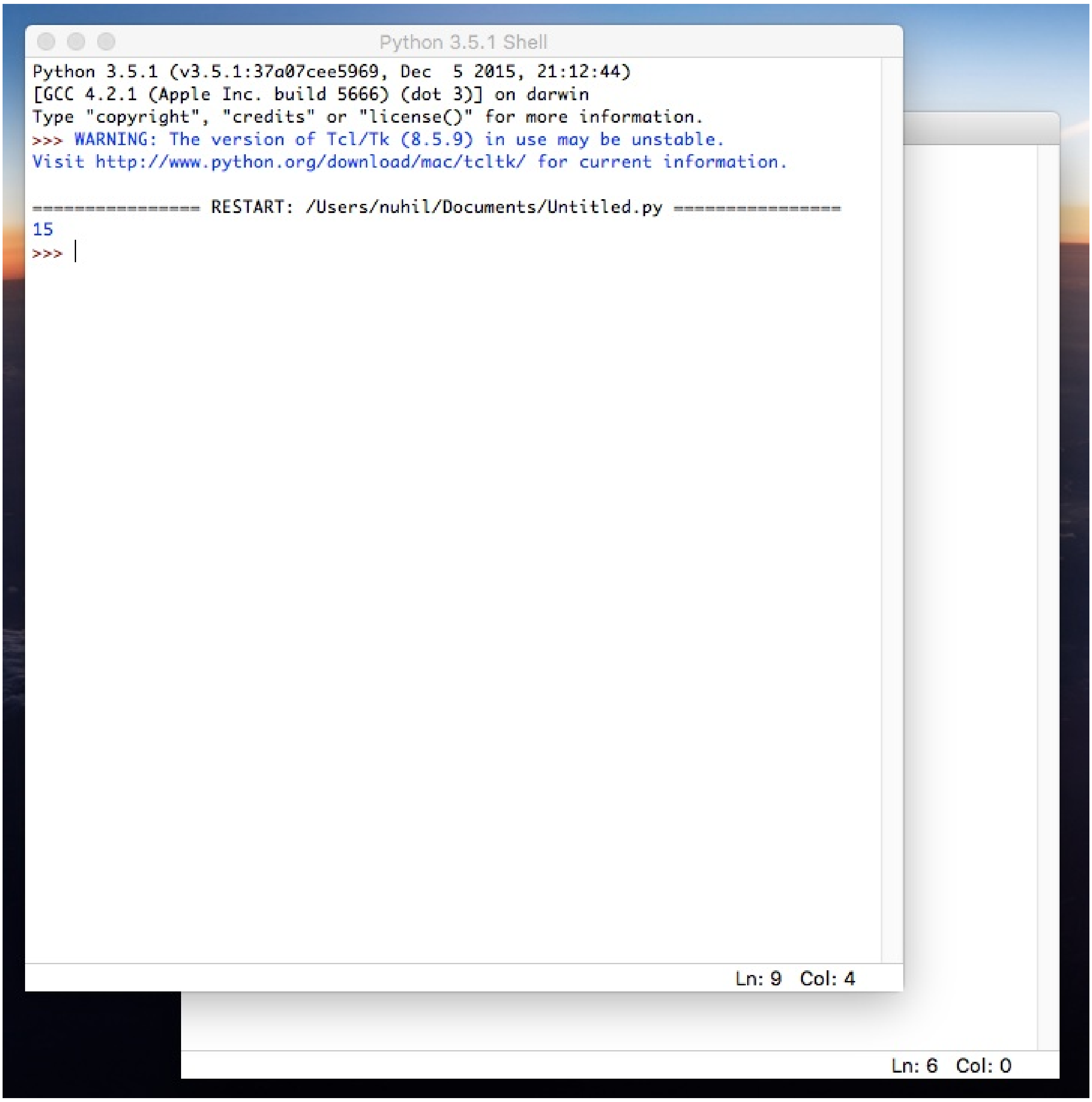
IDLE

আগেই বলা হয়েছে, পাইথনের অফিসিয়াল ইন্সটলেশনের সাথে একটি গ্রাফিক্যাল টুল IDLE ইন্সটল হয়। এটির মাধ্যমেও পাইথন সোর্স কোড ওয়ালা ফাইলকে রান করানো যায়। এর জন্য যা করতে হবে,

- IDLE প্রোগ্রামটি চালু করতে হবে
- File মেনু থেকে New File সিলেক্ট করে নতুন ফাইলে পাইথন কোড/প্রোগ্রাম লিখতে হবে



- ফাইলটি সেইভ করতে হবে
- Run মেনু থেকে Run Module ক্লিক করতে হবে তাহলে নিচের মত আউটপুট স্ক্রিন তৈরি হবে



IDE

পাইথনের অফিসিয়াল IDLE ব্যবহার করা ছাড়াও অনেক পপুলার এবং ইনটেলিজেন্ট IDE তথা Integrated Development Environment আছে। তার মধ্যে উল্লেখযোগ্য হচ্ছে PyCharm. এতে কোড লেখা, ডিবাগ করা, টেস্ট করা, প্যাকেজ ম্যানেজ করা, প্রজেক্ট স্পেসিফিক ইন্টারপ্রেটার সিলেক্ট করে দেয়া, ভার্চুয়াল এনভায়রনমেন্ট তৈরি করা ছাড়াও বড় বড় পাইথন প্রজেক্ট খুব সহজে হ্যান্ডেল করা যায়। PyCharm এর ফ্রি এডিশন (Community Edition) ডাউনলোড করা যাবে এখান থেকে।

এরকম IDE এর ব্যবহার জানতে গুগল অথবা অভিজ্ঞদের সাহায্য নেয়া যেতে পারে অথবা PyCharm এর সাইটেই ব্যাসিক ইউসেজ দেয়া আছে।

এই সেকশনে থাকছে

- বুলিয়ান
- if স্টেটমেন্ট
- else স্টেটমেন্ট
- বুলিয়ান লজিক
- অপারেটর প্রেসিডেন্স
- while লুপ
- লিস্ট
- লিস্ট অপারেশন
- লিস্ট ফাংশন
- রেঞ্জ
- for লুপ

বুলিয়ান

বুলিয়ান হলো এক প্রকারের ডাটাটাইপ যার মান সবসময় কোন কিছু সত্য অথবা মিথ্যা বুঝায়। সত্য ও মিথ্যাকে যথাক্রমে 1 ও 0 দ্বারা প্রকাশ করা হয়। এটি ইন্টেজার এর একটি সাবক্লাস। বুলিয়ান ধারনার প্রবক্তা জর্জ বুল। তার বই 'দা ম্যাথমেটিকাল এনালাইসিস অফ লজিক(১৮৪৭)' থেকে সর্বপ্রথম এ সম্পর্কে ধারনা পাওয়া যায়।

পাইথনে এই Boolean টাইপটির দুটি ভ্যালু আছে `True` এবং `False`

বুলিয়ান এক্সপ্রেশন
বুলিয়ান এক্সপ্রেশন হলো এমন কিছু এক্সপ্রেশন যেগুলো সত্য অথবা মিথ্যা মান রিটার্ন করে। একাধিক বুলিয়ান এক্সপ্রেশন মিলেও একটি বুলিয়ান এক্সপ্রেশন বানানো যায়।

বুলিয়ান অপারেটর
বুলিয়ান টাইপের তিনটি বেসিক অপারেটর আছে। এরা হলো `AND` , `OR` , `NOT` । `AND` এর বেলায় যদি সবগুলো ভ্যারিয়েবল এর মান সত্য হয় তবে এক্সপ্রেশন টি সত্য হয় অন্যথায় এক্সপ্রেশন টি মিথ্যা হয়। `OR` এর বেলায় যদি কমপক্ষে একটি ভ্যারিয়েবল এর মান সত্য হয় তবে এক্সপ্রেশন টি সত্য হয় অন্যথায় এক্সপ্রেশন টি মিথ্যা হয়। `NOT` একটি ইউনারি অপারেটর। এটি সাধারনত কোনো ভ্যারিয়েবল অথবা এক্সপ্রেশন এর বিপরীত ভ্যালু রিটার্ন করে।

ট্রুথ টেবিল
নিচে ট্রুথ টেবিল এর মাদ্ধমে বিষয়গুলো তুলে ধরা হলোঃ-

A	B	A AND B	A OR B	NOT A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

এই বেসিক এপারেটর ছাড়াও আরো কিছু অপারেটর আছে যেগুলো এই তিনটির সনন্বয়ে গঠন করা হয়েছে। যেমনঃ `XOR` , `XAND` , `NAND` , `NOR` ইত্যাদি। এ নিয়ে সামনের কোন এক চ্যাপ্টারে আবারো আলোচনা হবে।

পাইথনে কিছু উদাহরণ
পাইথনে দুটো এলিমেন্ট এর মধ্যে তুলনা করে অথবা সরাসরি ভ্যালু অ্যাসাইন করে বুলিয়ান ভ্যারিয়েবল তৈরি করা যায়। যেমন,

```
>>> my_boolean = True
>>> my_boolean
True

>>> 2 == 3
False
>>> "hello" == "hello"
True
```

আরও কিছু তুলনাকারী অপারেটর ব্যবহারের সময়,

```
>>> 1 != 1 # দেখা হচ্ছে ১ নট ইকুয়াল ১ কিনা। যেটা আসলে মিথ্যা। বাস্তবে ১ ইকুয়াল ১
False
>>> "eleven" != "seven" # এখানে eleven আর seven ইকুয়াল নয়। তাই এটা সত্য
True
>>> 2 != 10 # ২ কিন্তু ১০ এর সমান নয় যেটা যাচাই করা হচ্ছে। তাই যাচাই এর মান সত্য
True
```

```
>>> 7 > 5
True
>>> 10 < 10
False
>>> 7 <= 8
True
>>> 9 >= 9.0
True
```

if স্টেটমেন্ট

পাইথনে if স্টেটমেন্ট ব্যবহার করে নির্দিষ্ট একটি কন্ডিশনের উপর ভিত্তি করে কিছু স্টেটমেন্ট বা কোড ব্লককে রান করানো যায়। যদি একটি কন্ডিশন বা এক্সপ্রেশন সত্য হয় তাহলে এর আওতাভুক্ত স্টেটমেন্ট রান হয়।

উদাহরণ,

```
if 10 > 5:
    print("10 greater than 5") # এই স্টেটমেন্টটি if কন্ডিশনের এর আওতাভুক্ত
    print("IF scope finished")    # এই স্টেটমেন্টটিও if কন্ডিশনের এর আওতাভুক্ত

print("Program ended") # এই স্টেটমেন্টটি if কন্ডিশনের এর আওতাভুক্ত নয়
```

আউটপুট,

```
10 greater than 5
IF scope finished
Program ended
```

উপরের প্রোগ্রামে if কন্ডিশন দিয়ে চেক করা হচ্ছে 5 এর চেয়ে 10 বড় কিনা। বড় হলে এর ভিতরের দুই লাইন এক্সিকিউট হচ্ছে। যেহেতু এটা সত্য কন্ডিশন। তাই `10 greater than 5` এবং `IF scope finished` লাইন প্রিন্ট হচ্ছে। আবার, `Program ended` লাইনটি এসব কন্ডিশন এর বাইরের একটি সাধারণ স্টেটমেন্ট। আর তাই এই লাইনটিও সাধারণভাবেই প্রিন্ট হচ্ছে।

আর হ্যাঁ, এই যে বলা হচ্ছে if স্টেটমেন্টের আওতাভুক্ত বা আওতাভুক্ত নয়, এটি নির্ধারণ হয় `indentation` তথা স্টেটমেন্টের সামনে যথাযথ হোয়াইট স্পেস ব্যবহার করে। উপরের উদাহরণে `if 10 > 5:` হচ্ছে কন্ডিশন বা এক্সপ্রেশন। এর নিচের দুই লাইন কিন্তু if এর আওতাভুক্ত কারন এর সামনে `indentation` বা স্পেস ব্যবহার করে ডান দিকে সরিয়ে নেয়া হয়েছে। অন্যান্য প্রোগ্রামিং ল্যাঙ্গুয়েজে এই কাজটি করা হয়ে থাকে `{ }` ব্যবহার করে। অর্থাৎ নিচের মত,

```
if (condition) {
statement
}
```

পাইথনে `indentation` বাধ্যতামূলক অন্যথায় এরর তৈরি হবে।

> উপরের উদাহরণের প্রোগ্রামটির মধ্যে একাধিক `indentation` এর একাধিক স্টেটমেন্ট লাইন থাকায় পাইথন কনসোলে রান করা যাবে না। স্টেটমেন্ট গুলো নিয়ে একটি পাইথন ফাইল তৈরি করে অতঃপর রান করা যাবে অথবা IDE তে রান করা যাবে।

নেস্টেড if

```
num = 12
if num > 5:
    print("Bigger than 5")
    if num <= 47:
        print("Between 6 and 47")
```

আউটপুট,

```
Bigger than 5
Between 6 and 47
```

উপরের উদাহরণে, প্রথমে একটি if এক্সপ্রেশন আছে এবং এর আওতাভুক্ত স্টেটমেন্ট এর মধ্যে আরও একটি if এক্সপ্রেশন আছে। প্রথম if কন্ডিশন সত্য হলে যথাযথ ভাবে এর ভিতরের স্টেটমেন্ট রান হয় এবং সেখানে সাধারণ স্টেটমেন্টের মত করেই একটি if এক্সপ্রেশন আছে যেটাও রান হয় তথা এটির সত্যতা নির্ণয় করে পরবর্তী স্টেটমেন্ট রান করে।

else স্টেটমেন্ট

আগের চ্যাপ্টারে আমরা দেখেছি কিভাবে একটি if কন্ডিশন সত্য হলে তার আওতাভুক্ত কোড ব্লকটি রান হয়। else বস্তুত if এর সাথেই সম্পর্কিত। অর্থাৎ, যদি উল্লেখিত if কন্ডিশনটি সত্য না হয় তাহলে else এর আওতাভুক্ত কোডব্লক রান বা এক্সিকিউট হয়।

```
x = 4
if x == 5:
    print("Its 5")
else:
    print("Its not 5")
```

আউটপুট,

```
Its not 5
```

if else চেইন

একটি if স্কোপের মধ্যে যেহেতু যেকোনো কোডই থাকতে পারে সেহেতু এর মধ্যে আরও এক বা একাধিক if বা else লজিক অবস্থান করতেই পারে। যেমন নিচের উদাহরণে, প্রথমেই একটি if দিয়ে চেক করা হচ্ছে যে `num` এর ভ্যালু `5` কিনা। যদি না হয় তাহলে প্রোগ্রাম কন্ট্রোল আরেকটি ধাপে চলে যাচ্ছে যেখানে আরও একটি if দিয়ে চেক করা হচ্ছে `num` এর ভ্যালু `11` কিনা। নাহলে তার সাথে সম্পর্কিত একটি else ব্লকে চলে যাচ্ছে এবং তার মধ্যে থাকা একটি if দিয়ে আবারো চেক করা হচ্ছে `num` এর মান `7` কিনা এবং এই কন্ডিশনটি সত্য হওয়ায় স্ক্রিনে প্রিন্ট হচ্ছে `Number is 7`

```
num = 7
if num == 5:
    print("Number is 5")
else:
    if num == 11:
        print("Number is 11")
    else:
        if num == 7:
            print("Number is 7")
        else:
            print("Number isn't 5, 11 or 7")
```

আউটপুট,

```
Number is 7
```

মজার ব্যাপার হচ্ছে এরকম if else if এর চেইনকে একটু সংক্ষেপে `elif` দিয়েও লেখা যায়। উপরের প্রোগ্রামটি নিচের মত করেও লেখা যায়,

```
num = 7
if num == 5:
    print("Number is 5")
elif num == 11:
    print("Number is 11")
elif num == 7:
    print("Number is 7")
else:
    print("Number isn't 5, 11 or 7")
```

আউটপুট,

```
Number is 7
```

টারনারি অপারেটর

টারনারি শব্দের স্বাভাবিক অর্থ তিন সম্বন্ধীয়। এর নাম শুনেই বোঝা যাচ্ছে এই অপারেটরটি তিনটি আর্গুমেন্ট নিয়ে কাজ করে। ওদিকে, আমরা ইতোমধ্যে জেনেছি if এবং else সম্পর্কে। তো, এই if, else এবং সাথে একটি ভ্যালু এই তিনটি বিষয়কে নিয়ে খুব সহজে কন্ডিশনাল এক্সপ্রেশন লেখা যায় টারনারি অপারেটর এর কনসেপ্ট ইমপ্লিমেন্ট করে।

উদাহরণ,

```
a = 100
b = 200 if (a >= 100 and a < 200) else 300
print(b)
```

ধরে নেই, প্রথমেই `a` এর মান 100 অ্যাসাইন করা হয়েছে। এরপর `b` এর জন্য একটি মান অ্যাসাইন করতে চাচ্ছি। সেটা হতে পারে 200 অথবা 300. তো, আসলে কোনটা হবে সেটি নির্ধারণ করার জন্য একটি কন্ডিশন বসিয়েছি। কন্ডিশনটি হচ্ছে - `if (a >= 100 and a < 200)` অর্থাৎ `a` এর মান ১০০ থেকে বড় বা সমান এবং ২০০ থেকে ছোট হলে এই কন্ডিশনটি সত্য হবে আর তখন `b` এর মান হিসেবে 200 অ্যাসাইন হবে। কন্ডিশনটি মিথ্যা হলে `b` এর মধ্যে 300 ঢুকবে। ঠিক এগুলোই এক লাইনে লেখা হয়েছে যা বস্তুত টারনারি অপারেটর এর একটা প্রয়োগ।

আউটপুট,

```
200
```

আরেকটি উদাহরণ দেখি,

```
status  = 1
msg = "Logout" if status == 1 else "Login"
print(msg)
```

আউটপুট,

```
Logout
```

else এর আরও ব্যবহার

শুধুমাত্র `if` এর সাথে ব্যবহার বাদেও `else` কে ব্যবহার করা যায় `for` এবং `while` লুপের সাথেও। উদাহরণ সরূপ, যখন কোন ফর লুপের কাজ স্বাভাবিক ভাবে শেষ হয় তখন এর সাথে যুক্ত `else` ব্লকের কোড এক্সিকিউট হয়। নিচের উদাহরণটি দেখি,

```
for i in range(10):
    print(i)
else:
    print("Done")
```

আউটপুট,

```
0
1
2
3
4
5
6
7
8
9
Done
```

> সংকলন - নুহিল মেহেদী

বুলিয়ান লজিক

if স্টেটমেন্টের জন্য জটিল কন্ডিশন তৈরির ক্ষেত্রে বুলিয়ান লজিক ব্যবহৃত হয়ে থাকে। অর্থাৎ একটি if স্টেটমেন্ট যদি একাধিক কন্ডিশনের উপর নির্ভর করে সেখানে আমরা বুলিয়ান লজিক ব্যবহার করতে পারি। আগেও বলা হয়েছে, পাইথনে and, or এবং not এই তিন ধরণের বুলিয়ান অপারেটর আছে।

and

এই অপারেটর দুটো আর্গুমেন্ট নিয়ে যাচাই করে এবং সত্য হয় যখন দুটো আর্গুমেন্টই সত্য হয়।

```
>>> 1 == 1 and 2 == 2
True
```

এখানে `1 == 1` সত্য এবং `2 == 2` সত্য। তাই `and` অপারেটর এর আউটপুট `True`. একটি কথা মনে করিয়ে দেয়া দরকার, অন্যান্য প্রোগ্রামিং ল্যাঙ্গুয়েজে এ ধরণের AND, OR বা NOT অপারেটরকে সাধারণত `&&`, `||`, `!` দিয়ে প্রকাশ করা হয় যেখানে পাইথনে শব্দ আকারে লেখা হয়।

or

উপরে উল্লেখিত `and` অপারেটর এর মতই `or` এরও দুটো আর্গুমেন্ট থাকে কিন্তু এটি সত্য হয় যদি উক্ত দুটো আর্গুমেন্টের যেকোনো একটি সত্য হয়। অর্থাৎ নিচের স্টেটমেন্টে,

```
>>> 1 == 1 or 2 == 3
True
```

এখানে `or` এর বাম পাশের লজিকটি সত্য কিন্তু ডান পাশেরটি সত্য নয়। তারপরেও `or` এর আউটপুট `True`.

not

অন্য দুটি অপারেটর এর মত `not` দুটো আর্গুমেন্ট নিয়ে কাজ করে না। বরং এর জন্য একটি আর্গুমেন্টই যথেষ্ট। এটি দিয়ে চেক করা হয় কোন লজিক না হয় কিনা। নিচের উদাহরণ দেখলে পরিষ্কার বোঝা যাবে,

```
>>> not 1 == 1
False
>>> not 1 > 7
True
```

প্রথম স্টেটমেন্ট এর `1 == 1` এটা আমরা সবাই জানি। এর সামনে `not` জুড়ে দিয়ে দেখার চেষ্টা করছি যে `1 == 1` নয়। কিন্তু এটা আসলে ঠিক না, ১ আর ১ তো সমানই। আর তাই `not 1 == 1` এর আউটপুট আসছে `False`. সেরকম দ্বিতীয় লাইনে দেখছি/পড়ছি `নয় ১ > ৭` এবং এটা সঠিক। আসলেই ১ কিন্তু ৭ এর চেয়ে বড় নয়। তাই `not 1 > 7` স্টেটমেন্টটি সত্য রিটার্ন করছে।

উদাহরণ

```python
a = 1
b = 1

c = 5
d = 10

if a == b and d > c:
    print("a and b are equal and d is greater than c")
```

আউটপুট,

```
a and b are equal and d is greater than c
```

> সংকলন - নুহিল মেহেদী

অপারেটর প্রেসিডেন্স

সাধারণ গণিতে যেমন যোগ বা বিয়োগের আগে গুন ও ভাগ করে নিতে হয় তেমনি প্রোগ্রামিং -এও এই অপারেটর গুলোর একটা অগ্রাধিকার মূলক নিয়ম আছে। অর্থাৎ সেই নিয়ম মেনেই একটি স্টেটমেন্ট এর মধ্যে থাকা একাধিক অপারেটরের অপারেশন ঘটবে। এটা গণিতের সরল করার নিয়মের সাথেই মিলে যায় অর্থাৎ - প্রথমেই ব্র্যাকেটের কাজ, তারপর পাওয়ার/এক্সপোনেন্ট, অতঃপর গুন ও ভাগ এবং শেষে যোগ ও বিয়োগ।

যোগ, বিয়োগ, গুন, ভাগ বাদেও যেহেতু প্রোগ্রামিং -এ আরও কিছু অপারেটর আছে, তাই সেগুলোর অগ্রাধিকারও জেনে রাখা দরকার। যেমন নিচের স্টেটমেন্ট দুটি দেখি,

```
>>> False == False or True
True
>>> False == (False or True)
False
```

উপরের প্রথম স্টেটমেন্টে `==` এর অগ্রাধিকার `or` চেয়ে বেশি। আর নিচের স্টেটমেন্টে `or` অপারেশন অগ্রাধিকার পেয়েছে কারন এটি একটি বন্ধনীর মধ্যে অবস্থান করছে।

অপারেটর
`**`
`~+-`
`* / % //`
`+-`
`>> <<`
`&`
`^`
`<= < > >=`
`<> == !=`
`= %= /= //= -= += *= **=`
`is is not`
`in not in`
`not or and`

টেবিলঃ বিভিন্ন অপারেটরের অগ্রাধিকার (উপর থেকে নিচে - বেশি থেকে কম)

while লুপ

আগের কিছু চ্যাপ্টারে আমরা দেখেছি কিভাবে একটি if স্টেটমেন্টের কন্ডিশন সত্য হলে তার আওতাভুক্ত একটি কোড ব্লক রান করানো যায়। if এর ক্ষেত্রে কন্ডিশন সত্য হলে কোড ব্লকটি মাত্র একবার রান হয়। while লুপ মোটামুটি একইরকম ভাবে কাজ করে। যেমন - এরও একটি কন্ডিশন দরকার হয় এবং সেটি সত্য হলে, এর আওতাভুক্ত কোড রান করে। কিন্তু গুরুত্বপূর্ণ বিষয়টি হচ্ছে শুধু একবার না বরং অনেক বার রান করানো যায় (একে iteration বলে)।

অর্থাৎ, যতক্ষণ সেই while লুপের কন্ডিশন সত্য থাকে ততক্ষণ পর্যন্ত এর আওতাভুক্ত কোড রান করতেই থাকে। আর যখন কন্ডিশনটি মিথ্যা হয়ে যায় তখন while লুপের বাইরে গিয়ে প্রোগ্রামের পরবর্তী স্টেটমেন্ট গুলো রান করা শুরু করে।

উদাহরণ,

```
i = 1
while i <= 5:
    print(i)
    i = i + 1

print("I am priting eventually cause the WHILE loop is done with his job.")
```

উপরের প্রোগ্রামে প্রথমেই একটি ভ্যারিয়েবল `i` নেওয়া হয়েছে এবং এর মান সেট করা হয়েছে `5`. এরপর একটি while লুপ এর শুরু হয়েছে। আগেই বলেছি এটিও `if` এর মত কন্ডিশন সত্য কিনা তা যাচাই করে। তাহলে কি দাঁড়াচ্ছে? `while i <= 5:` এখানে এসে আমরা দেখছি কন্ডিশনটি সত্য। তার মানে এর আওতাভুক্ত কোড কাজ করবে। তাহলে দেখে নেই এর আওতাভুক্ত কোড কি আছে। প্রথমেই আছে একটা `print` এর কাজ যেটা প্রিন্ট করবে `i` এর বর্তমান মান তথা `1`. এর পর আরও একটা স্টেটমেন্ট আছে যেটাও কিনা সেই `while` এরই আওতাভুক্ত। তার মানে সেটিও এক্সিকিউট হবে। সেই স্টেটমেন্টটির কাজ হচ্ছে `i` এর মান এক বাড়িয়ে দেয়া। এভাবে `while` লুপের একবার কাজ করা শেষ। কিন্তু এটি `if` এর মত একবার কাজ করেই শেষ হয়ে যায় না। বরং আবার কন্ডিশন চেক করতে ফিরে যায় এর কার্যক্রমের প্রথমে অর্থাৎ `while i <= 5:` এই লাইনে।

এখানে এসে চেক করার সময় `i` এর মান পায় `2` যেটা এখন পর্যন্ত সত্য অর্থাৎ `2` কিন্তু `5` এর ছোট। তাই আবারো লুপের মধ্যে থাকা কাজ করতে ঢুকে যায়। আবারো `i` এর মান প্রিন্ট করে এবং এর মান এক বাড়িয়ে লুপের শুরুতে ফিরে যায়। এভাবে একবার `i` এর মান `6` হয় এবং লুপের শুরুতে ফিরে গিয়ে প্রোগ্রাম যখন চেক করে `i` তথা `6` কিন্তু `5` এর ছোট বা সমান নয়। তখন আর লুপের মধ্যেকার কোড গুলো রান না করে লুপ থেকে একবারে বেরিয়ে পরবর্তী অন্যান্য স্টেটমেন্ট গুলো রান করা শুরু করে।

লুপের বাইরে আমাদের একটি স্টেটমেন্ট আছে `print("I am printing eventually cause the WHILE loop is done with his job.")` যেটা একবার রান হয় কিন্তু তার আগে `while` লুপ তার কন্ডিশন মোতাবেক একাধিক বার রান হয়ে তার দায়িত্ব শেষ করেছিল।

আউটপুট,

```
1
2
3
4
5
I am priting eventually cause the WHILE loop is done with his job.
```

infinite লুপ

একটা কথা মাথায় আসতে পারে আমাদের সেটা হচ্ছে - যদি `while` লুপ একটা কন্ডিশন যতক্ষণ সত্য হয় ততক্ষণ রান করে তাহলে একটা কাজ করলে কেমন হয়; এমন একটা কন্ডিশন সেট করে দেবো ওর জন্য যেটা কোনদিন মিথ্যাই হবে না :P তাহলে `while` লুপ এর কাজ তো শেষই হবার কথা না, তাই না?

হ্যাঁ, ঠিক এরকম আজীবন চলা লুপকে inifinite লুপ বলা যেতে পারে।

উদাহরণ,

```
while 1 == 1:
   print("In the loop")
```

এখানে `while` লুপের জন্য কন্ডিশন সেট করেছি এরকম যে - যতক্ষণ ১ এর সমান ১ হবে ততক্ষণ সে তার মধ্যেকার কোড রান করবে। আর আমরা সবাই জানি যে, সারাজীবনই ১ আর ১ সমান। আর তাই এই লুপ লজিক্যালি একটি infinite লুপ।

> উপরের প্রোগ্রাম লিখে কেউ রান করার চেষ্টা করলে স্ট্যান্ডার্ড আউটপুট স্ক্রিনে অনবরত `In the loop` লেখাটি আসতেই থাকবে। এমতাবস্থায়, কিবোর্ডের `ctrl+C` চেপে প্রোগ্রামটির কার্যক্রম বন্ধ করা যাবে।

break

কিন্তু এরকম জোড় করে বন্ধ করা পছন্দ না হলে জানিয়ে রাখা ভালো যে - প্রোগ্রাম্যাটিক্যালিও `while` লুপের কাজ যেকোনো সময় কন্ডিশনের পরোয়া না করেও বন্ধ করা সম্ভব। এর জন্য শুধু লিখতে হবে `break`.

উদাহরণ,

```
i = 0
while 1 == 1:
   print(i)
   i = i + 1
   if i >= 5:
      print("Breaking")
      break

print("Finished")
```

উপরের প্রোগ্রামটি দেখে আপাতদৃষ্টিতে মনে হতে পারে এটি একটি infinite লুপ। ধারনা ঠিকি আছে কিন্তু আমরা একে অনন্তকাল চলতে না দিয়ে একটা ছোট্ট লজিকের উপর ভিত্তি করে এর চলমান প্রক্রিয়া বন্ধ করে দিয়েছি। অর্থাৎ অনন্তকাল চলার ইচ্ছায় কাজ শুরু করে ৫ বার চলার পর আমাদের `while` এর কন্ট্রোলার এর মধ্যে থাকা `if` এর

ফাঁদে (সত্যতায়) পরে যায় । আর আমরা বুদ্ধি করে সেই `if` এর কোড ব্লকের মধ্যে লিখেছি একটি প্রিন্ট স্টেটমেন্ট এবং সেই মহা ঘাতক `break` . আর তাই, এ অবস্থায় এসে ১ এর সমান ১ এবং বাকী সব ভুলে `while` লুপ তার কার্যক্রমে ক্ষান্ত দেয় এবং প্রোগ্রামের কন্ট্রোল চলে যায় নিচের সাধারণ প্রিন্ট স্টেটমেন্টে `print("Finished")`

আউটপুট,

```
0
1
2
3
4
Breaking
Finished
```

continue

এটি আরেকটি মজার জিনিষ । ধরা যাক, একটি লুপের মধ্যে আমরা বেশ কিছু কাজ করার স্টেটমেন্ট লিখেছি এবং চাচ্ছি যে লুপ যতক্ষণ সত্য থাকে (ধরে নেই ১০০ বার) ততক্ষণ এর মধ্যেকার কাজ গুলো বার বার হোক । কিন্তু, এমনও তো হতে পারে যে, সেই ১০০ বারের মধ্যে বিশেষ কয়েকবার আমরা সেই পুরো কাজটা করতে চাই না কিন্তু নির্ধারিত ১০০ বারই লুপকে কাজ করাতে দিতে চাই; তাহলে কি করতে পারি?

এর জন্যই আছে `continue` . একটি `while` লুপের মধ্যে যখনই `continue` এক্সিকিউট হবে তখনিই লুপের মধ্যে থাকা এর পরের কোড গুলো এক্সিকিউট হবে না এবং লুপের কন্ট্রোল একদম শুরুতে চলে যাবে ।

নিচের উদাহরণ দেখলে বিষয়টি পরিষ্কার হয়ে যাবে,

```
i = 0
while True:
   i = i +1
   if i == 2:
      print("Skipping 2")
      continue
   if i == 5:
      print("Breaking")
      break
   print(i)

print("Finished")
```

আউটপুট,

```
1
Skipping 2
3
4
Breaking
Finished
```

উপরের লুপটি একটি ইনফিনিট লুপ হলেও একটু বিশেষ ভাবে নিয়ন্ত্রণ করেছি। যেমন যেবার `i` এর মান `2` হয়েছে সেই বার `continue` এক্সিকিউশনের মাধ্যমে লুপকে জোড় করে শুরুতে নিয়ে যাওয়া হয়েছে তাই ওই বারের `2` প্রিন্ট হয় নি। এরপর আবার সাধারণভাবে `3,4` প্রিন্ট হয়েছে। আবার যখন `5` পেয়েছি তখনি `if` এর সত্যতার কারনে `break` এর এক্সিকিউশন হয়েছে এবং লুপ অকালে (এর অনন্তকাল চলার প্ল্যান ছিল) শেষ হয়েছে।

> সংকলন - নুহিল মেহেদী

list

পাইথনে ৬ ধরণের বিল্ট ইন টাইপ আছে। সেগুলো হচ্ছে - numeric, sequence, mapping, class, instance এবং exception. সব থেকে ব্যাসিক ডাটা স্ট্রাকচারটি হচ্ছে sequence. এর প্রত্যেকটি এলিমেন্টের জন্য একটি নাম্বার অ্যাসাইন করা হয় যাকে ইনডেক্স বা পজিশন বলা যায়। প্রথম ইনডেক্স শূন্য, তারপর ১ এবং এরপর ক্রমিক আকারে বাড়তে থাকে।
পাইথনে আবার ৩ ধরণের ব্যাসিক sequence টাইপ আছে যেগুলো হচ্ছে list, tuple, এবং xrange object. এই চ্যাপ্টারে আমরা আলোচনা করবো list নিয়ে।

দুটো স্কয়ার ব্র্যাকেট এবং এর মধ্যে কমা দিয়ে আলাদা আলাদা এলিমেন্ট যুক্ত করে একটি লিস্ট তৈরি করা যায়। আর আগেই বলা হয়েছে, এর এলিমেন্ট গুলো ইনডেক্স অনুযায়ী সাজানো থাকে অর্থাৎ ০, ১, ২ এরকম ক্রমে। যেমন -

```
words = ["Hello", "world", "!"]
print(words[0])
print(words[1])
print(words[2])
```

এখানে `words` একটি লিস্ট টাইপ ভ্যারিয়েবল যার মধ্যে ৩টি এলিমেন্ট আছে যেগুলো হল - `Hello` , `world` এবং `!` এই তিনটি string। লিস্টের যেকোনো এলিমেন্টকে অ্যাক্সেস করার জন্য ওই লিস্টের নাম, সাথে স্কয়ার ব্র্যাকেটের মধ্যে তার ইনডেক্স নাম্বার দিতে হয়। উপরে যেভাবে তিনটি এলিমেন্ট প্রিন্ট করা হয়েছে, সেভাবে।

আউটপুট,

```
Hello
world
!
```

ফাকা **list**
নিচের মত করে একটি ফাকা লিস্ট তৈরি করা যায় -

```
my_list = []
print(my_list)
```

আউটপুট,

```
[]
```

এলিমেন্ট টাইপ
পাইথনে একটি লিস্টের মধ্যে বিভিন্ন টাইপের ডাটা বা এলিমেন্ট রাখা যেতে পারে। যেমন একটি লিস্টের এলিমেন্ট হিসেবে কিছু নাম্বার, কিছু স্ট্রিং এমনকি অন্য এক বা একাধিক লিস্টকেও রাখা যেতে পারে। যদিও সাধারণত একটি

লিস্টে একই রকম এলিমেন্ট রাখা ভালো প্র্যাকটিস।

উদাহরণ,

```
number = 1
my_numbers = [number, 2, 3]

things = ["Numbers", 0, my_numbers, 4.56]

print(things[0])
print(things[1])
print(things[2])
print(things[2][2])
```

আউটপুট,

```
Numbers
0
[1, 2, 3]
3
```

list হিসেবে string

একটি স্ট্রিং টাইপ ভ্যালু পাইথনে লিস্ট হিসেবে আচরণ করে অর্থাৎ স্ট্রিং -এর প্রত্যেকটি ক্যারেক্টার একটি কাল্পনিক লিস্টের এক একটি এলিমেন্ট হিসেবে মনে হয়। নিচের উদাহরণটি দেখি,

```
str = "Hello world!"
print(str[6])
```

আউটপুট,

```
w
```

অর্থাৎ, `str = "Hello world!"` এবং `str = ["H", "e", "l", "l", "o", " ", "w", "o", "r", "l", "d"]` কার্যত একই। আর তাই `str[6]` এর মান আসছে `w`.

list অপারেশন

এই চ্যাপ্টারে আমরা আলোচনা করবো list এর কিছু ব্যাসিক অপারেশন নিয়ে। আগের চ্যাপ্টারে আমরা দেখেছি কিভাবে একটি লিস্টের নির্দিষ্ট ইনডেক্সে থাকা একটি এলিমেন্টকে অ্যাক্সেস করা যায়। তাহলে এবার দেখি, কিভাবে একটা নির্দিষ্ট ইনডেক্সে বা পজিশনে নতুন কোন এলিমেন্ট যুক্ত করা যায়,

```
my_numbers = [1, 2, 3, 5]
my_numbers[3] = 4
print(my_numbers)
```

আউটপুট,

```
[1, 2, 3, 4]
```

অর্থাৎ `my_numbers` লিস্টের `3` পজিশনে আগে ছিল `5` এবং সেই অবস্থানে আমরা নতুন ভ্যালু সেট করলাম `4`. `my_numbers[3] = 4` এভাবে। আর তাই `my_numbers` লিস্ট প্রিন্ট করার ফলে আউটপুট আসলো এই লিস্টের আপডেটেড ভ্যালু গুলো।

লিস্টের যোগ ও গুন
মজার ব্যাপার হচ্ছে string এর মত করে লিস্ট নিয়েও যোগ বা গুনের কাজ করা যায়। যেমন - নিচের উদাহরণটা দেখে নেই,

```
first_list = [1, 2, 3]
print(first_list + [4, 5, 6])
print(first_list * 3)
```

আউটপুট,

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

লিস্টের মধ্যের এলিমেন্ট চেক
কোন লিস্টের মধ্যে নির্দিষ্ট কোন এলিমেন্ট আছে কিনা সেটা চেক করার জন্য `in` অপারেটর ব্যবহার করা হয়। যদি এলিমেন্টটি লিস্টের মধ্যে এক বা একাধিকবার থাকে তাহলে এটি `True` রিটার্ন করে অন্যথায় `False` রিটার্ন করে।

উদাহরণ,

```
fruits = ["apple", "orange", "pineappe", "grape"]

print("orange" in fruits)
print("rice" in fruits)
print("apple" in fruits)
```

আউটপুট,

```
True
False
True
```

একই ভাবে এর সাথে `not` অপারেটর ব্যবহার করে কোন এলিমেন্টের অনুপস্থিতিও চেক করা যাতে পারে। যেমন -

```
fruits = ["apple", "orange", "pineappe", "grape"]

print("orange" not in fruits)
print(not "rice" in fruits)
```

আউটপুট,

```
False
True
```

লিস্ট ফাংশন

এই চ্যাপ্টারে আমরা লিস্ট নিয়ে কাজ করার জন্য এর কিছু বিল্ট-ইন মেথড এবং কিছু ফাংশনের ব্যবহার দেখবো। লিস্ট ম্যানিপুলেশনের জন্য কি কি মেথড এভেইলেবল আছে সেগুলো আমরা কিভাবে জানতে পারি? সে জন্য একটা ছোট্ট টি প্স দিচ্ছিঃ টার্মিনালে, IDLE -তে অথবা যেখানে পাইথন কোড রান করছেন সেখানে `dir(list)` লিখে এন্টার/রান/প্রিন্ট করে দেখতে পারেন। নিচের মত আউটপুট পেয়ে যাবেন -

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__', '_
```

একটা লিস্ট রিটার্ন হয়েছে। তাই তো? শেষের দিকে খেয়াল করুন - `...append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']` অর্থাৎ এগুলো মেথড বিল্ট ইন আছে লিস্ট ম্যানিপুলেশনের জন্য। অর্থাৎ `list` অবজেক্টের সব গুলো অ্যাট্রিবিউট এবং মেথড এর তালিকা দেখার জন্য এটি ব্যবহার করতে পারেন। এটি শুধু যে list এর ক্ষেত্রেই কাজ করবে তা নয়। `dir()` এর মধ্যে অন্যান্য অবজেক্ট পাস করেই দেখুন না।

append

যা হোক, তো আমরা জানতে পারলাম list এর মধ্যে `append` , `insert` ইত্যাদি করা যেতে পারে। তাহলে চলুন `append` ট্রাই করে দেখি। কিন্তু এটা কিভাবে কাজ করে সেটাও কারো কাছে না জিজ্ঞেস করেও জেনে নিতে পারেন। আরও একটা টি প্সঃ রান করুন, `help(list.append)` আর নিচের মত আউটপুট আসবে -

```
Help on method_descriptor:

append(...)
    L.append(object) -- append object to end
```

অর্থাৎ এই মেথডের কাজ কি সেটা দেখা যাচ্ছে এবং বলা আছে append object to end. অর্থাৎ কোন একটি লিস্টের শেষে নতুন এলিমেন্ট যুক্ত করতে এই মেথড ব্যবহার করা যাবে। তাহলে উদাহরণ দেখে নেই -

```
nums = [1, 2, 3]
nums.append(4)
print(nums)
```

আউটপুট,

```
[1, 2, 3, 4]
```

insert

প্রায় একই রকম কিন্তু একটু আলাদা কারনে ব্যবহার করা যেতে পারে `insert` মেথড। নিচের মত করে -

```
words = ["A", "C"]
index = 1
words.insert(index, "B")

print(words)
```

আউটপুট,

```
['A', 'B', 'C']
```

অর্থাৎ, লিস্টের কোন একটি নির্দিষ্ট পজিশনে বা ইনডেক্সে কোন এলিমেন্ট যুক্ত করতে চাইলে `append` এ কাজ হবে না (কারন এটা শেষে যুক্ত করে) বরং `insert` ব্যবহার করতে হবে। `insert` মেথডের দুটো প্যারামিটার - প্রথমটি হচ্ছে লিস্টের কোন পজিশনে নতুন এলিমেন্ট যুক্ত করতে চান আর দ্বিতীয় প্যারামিটারটি হচ্ছে যে এলিমেন্ট যুক্ত করতে চান সেটি নিজেই। উপরের উদাহরণে, আমরা `words` লিস্টের দ্বিতীয় পজিশন তথা `1` ইনডেক্সে `B` কে যুক্ত করেছি।

index

আরও একটি মেথডের ব্যবহার দেখি। যেমন - `index`. নিচের উদাহরণে আমরা যেকোনো একটি এলিমেন্ট লিস্টের কোন ইনডেক্সে অবস্থা করছে সেটা চেক করার জন্য `index` মেথড ব্যবহার করেছি।

```
letters = ['p', 'q', 'r', 's', 'p', 'u']
print(letters.index('r'))
print(letters.index('p'))
print(letters.index('z'))
```

আউটপুট,

```
2
0
ValueError: 'z' is not in list
```

count

লিস্টের মধ্যে কোন একটি এলিমেন্ট মোট কতবার আছে তার সংখ্যা জানতে নিচের মত করে `count()` মেথডের ব্যবহার করা যেতে পারে,

```
letters = ['p', 'q', 'r', 's', 'p', 'u']
letters.count('p')
```

আউটপুট,

```
2
```

> এরকম সব গুলো মেথডের কাজ জেনে নিতে `help(list.METHOD_NAME)` এভাবে রান করে আউটপুট স্ক্রিন থেকে উক্ত মেথডের বিস্তারিত দেখে নিতে পারেন।

অবজেক্ট মেথড বাদেও লিস্ট এর জন্য কিছু উপকারী ফাংশন আছে। যেমন - `max(), min(), len()` ইত্যাদি. যেমন একটি লিস্টের মধ্যে থাকা এলিমেন্ট গুলোর মধ্যে থেকে বড়টি দেখে নিতে `max()` ফাংশনের ব্যবহার করা যেতে পারে। উদাহরণ,

```
nums = [1, 2, 4, 20, 50, 3, 4]
max(nums)
```

আউটপুট,

```
50
```

> সংকলন - নুহিল মেহেদী

রেঞ্জ

আমরা আগের চ্যাপ্টার গুলোতে দেখেছি কিভাবে লিস্ট তৈরি করতে হয় এবং লিস্ট নিয়ে কাজ করতে হয়। আরও দেখেছি `while` লুপের ব্যবহার। পরের চ্যাপ্টারে আমরা আরও একধরনের লুপ (for) নিয়ে আলোচনা করবো। তার আগে, এই চ্যাপ্টারে আমরা একটি বহুল ব্যবহৃত ফাংশন নিয়ে কথা বলবো যার নাম `range`. এই ফাংশন এর মাধ্যমে স্বয়ংক্রিয় ভাবে একটি লিস্ট তৈরি করা যায়, যার এলিমেন্ট গুলো হয় একটি নির্দিষ্ট ক্রম অনুযায়ী। যেমন -

```
my_numbers = list(range(10))
print(my_numbers)
```

আউটপুট,

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

উপরের উদাহরণে, ০ থেকে ৯ পর্যন্ত ১০টি ক্রমিক সংখ্যা সম্বলিত একটি লিস্ট তৈরি করা হয়েছে। `range` এর সাথে `list` ফাংশনের ব্যবহার করা হয়েছে কারন, `range` বস্তুত একটি অবজেক্ট রিটার্ন করে আর তাই একে `list` ফাংশনের আর্গুমেন্ট হিসেবে পাঠিয়ে একটি ব্যবহার উপযোগী লিস্ট হিসেবে রূপান্তর করা হয়েছে।

সবসময় যে ০ থেকেই রেঞ্জ এর এলিমেন্ট শুরু হয় সেটা নয়। বরং `range` ফাংশনের আর্গুমেন্ট পরিবর্তন করে ইচ্ছা মত শুরু এবং শেষের লিমিট ঠিক করে দেয়া যায়। যেমন -

```
my_numbers = list(range(5,10))
print(my_numbers)
```

আউটপুট,

```
[5, 6, 7, 8, 9]
```

এখানে লিস্টের প্রথম এলিমেন্ট শুরু হয়েছে ৫ থেকে এবং শেষ হয়েছে ৯ এ গিয়ে কারন আমরা ৫ থেকে ১০ পর্যন্ত ভ্যালু চেয়েছি। আবার, এই ক্রমিক প্যাটার্নও বদলানো যায় `range` ফাংশনের তৃতীয় আর্গুমেন্ট সেট করে। অর্থাৎ আমরা চাইলে শুরু এবং শেষ নাম্বারের মধ্যে একটি ইন্টারভাল সেট করতে পারি যাতে করে এলিমেন্ট গুলো ক্রমিক না হয়ে বরং নির্ধারিত বিরতির হবে। উদাহরণ,

```
my_numbers = list(range(5, 30, 3))
print(my_numbers)
```

আউটপুট,

```
[5, 8, 11, 14, 17, 20, 23, 26, 29]
```

আশা করি আউটপুট দেখেই বুঝতে পারছেন কিভাবে ক্রমটি সাজানো।

ফর লুপ

এর আগে আমরা দেখেছি কোন একটা নির্দিষ্ট কন্ডিশনের সত্যতার উপর ভিত্তি করে একটি কাজ একাধিকবার করার জন্য `while` লুপের ব্যবহার। কিন্তু, পাইথনের যেকোনো সিকোয়েন্স টাইপ অবজেক্ট যেমন, লিস্ট (`list`) এর প্রত্যেকটি এলিমেন্ট নিয়ে কাজ করার জন্য `while` লুপ ব্যবহার করলে একটু বেশি কোড লিখতে হয়। যেমন `while` লুপের কনসেপ্ট ঝালাই করতে এবং ব্যাপারটা বুঝতে নিচের উদাহরণটি দেখি,

```
fruits = ["Apple", "Orange", "Pineapple", "Grape"]
# Lets make juice with these fruits

start_index = 0
max_index = len(fruits) - 1

while start_index <= max_index: # Work until this condition is True
    fruit = fruits[start_index]
    print(fruit + " Juice!")

    start_index = start_index + 1
```

আউটপুট,

```
Apple Juice!
Orange Juice!
Pineapple Juice!
Grape Juice!
```

ঠিক একই কাজ `for` লুপ ব্যবহার করে করলে অনেক কম কোড লিখেই করা সম্ভব। `for` লুপ দিয়ে খুব সহজেই যেকোনো সিকোয়েন্স টাইপ অবজেক্ট যেমন `list` , `string` ইত্যাদির মধ্যে iterate করা যায়। তাহলে দেখি উপরের প্রোগ্রামটি কিভাএব ফর লুপ ব্যবহার করে করা সম্ভব,

```
fruits = ["Apple", "Orange", "Pineapple", "Grape"]
# Lets make juice with these fruits

for fruit in fruits:
    print(fruit + " Juice!")
```

আউটপুট,

```
Apple Juice!
Orange Juice!
Pineapple Juice!
Grape Juice!
```

আউটপুট কিন্তু একই। তাই, যখনই কোন iterable নিয়ে কাজ করার প্রয়োজন পরবে তখন `for` লুপ ব্যবহার করাই ভালো হয়।

> অন্যান্য ল্যাঙ্গুয়েজ যেমন php তে এরকম কাজের জন্য আছে `foreach` যা দিয়ে কোন অ্যারে তে অপারেশন করা অনেক সহজ হয়ে যায়

এখন ধরুন আপনার কাছে কোন লিস্ট নাই কিন্তু নির্দিষ্ট সংখ্যকবার একটি কাজ পুনরাবৃত্তি করা দরকার। তখন কি করব? এ জন্য একটি সুন্দর ফাংশন হতে পারে `range` যা আমরা আগের চ্যাপ্টারে দেখে এসেছি। মনে আছে, `range` ব্যবহার করে ইচ্ছামত লিস্ট তৈরি করা যায়? এটাকেই কাজে লাগিয়ে নিচের উদাহরণটি দেখি,

```
for i in range(10):
    print(i)
```

আউটপুট,

```
0
1
2
3
4
5
6
7
8
9
```

অর্থাৎ, `range` ফাংশন ব্যবহার করে একটি কাল্পনিক লিস্ট তৈরি করা হয়েছে যার এলিমেন্ট গুলো ছিল ০ থেকে ৯ পর্যন্ত `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]` এরকম। আর সেই লিস্টকেই iterate করা হয়েছে for লুপ দিয়ে অর্থাৎ ১০ বার কাজ করানো হয়েছে এই লুপকে। আর কাজটা ছিল তেমন কিছুই না প্রত্যেকটি এলিমেন্টকে ধরে প্রিন্ট করা।

আচ্ছা, string তো একরকম iterable তাহলে এখানেও একবার `for` লুপ খাটিয়ে দেখি কিছু করা যায় কিনা -

```
for letter in 'Python': # Here "Python" acts like a list of characters
    print(letter)
```

আউটপুট,

```
P
y
t
h
o
n
```

`while` লুপের মত ফর লুপেও `break` , `continue` ইত্যাদি কিওয়ার্ড ব্যবহার করে কাজের ধারাকে নিয়ন্ত্রণ করা যায় । যেমন -

```
for i in range(20):
    if i == 5:
        continue
    if i > 9:
        break
    print(i)

print("Printed first 10 numbers except 5!")
```

```
0
1
2
3
4
6
7
8
9
Printed first 10 numbers except 5!
```

উপরে `0` থেকে `19` এই ২০টি এলিমেন্ট ওয়ালা একটি লিস্ট/রেঞ্জ এর উপর কাজ করা হয়েছে কিন্তু যখন `5` এলিমেন্টকে পাওয়া গেছে (`i` এর মাধ্যমে) তখন `continue` ব্যবহার করে একে প্রিন্ট না করে এড়িয়ে যাওয়া হয়েছে (লুপের শুরুতে ফিয়ে গিয়ে) । আবার যখন এলিমেন্টটি `9` এর বড়, সেই সময় ফর লুপের কাজ `break` এর মাধ্যমে থামিয়ে দেয়া হয়েছে যে কারনে `9` প্রিন্ট এর পর ফর লুপের কোন কাজ দেখা যাচ্ছে না বরং প্রোগ্রামের শেষ একটি সাধারণ প্রিন্ট স্টেটমেন্ট এর এক্সিকিউশন হয়েছে ।

> সংকলন - নুহিল মেহেদী

এই সেকশনে থাকছে

আগের কিছু চ্যাপ্টারে যথাযথ উদাহরণ এবং বর্ণনার জন্য কিছু ডাটা টাইপ নিয়ে ব্যাসিক আলোচনা করা হয়েছে বা প্রাসঙ্গিক ভাবেই সেগুলো চলে এসেছে। যেমন - প্রিন্ট এবং ইনপুট/আউটপুট নিয়ে আলোচনার সময় string এর পরিচয় ও ব্যবহার করা হয়েছে। আবার for লুপ, while লুপ নিয়ে আলোচনার সময় প্রাসঙ্গিক ভাবে list নিয়ে আলোচনা হয়েছে, যেহেতু লিস্ট একটি ব্যাসিক iterable.

এই সেকশনে আরও কিছু গুরুত্বপূর্ণ ডাটাটাইপ নিয়ে আলোচনা করা হবে যা নিম্ন উল্লেখিতঃ

- None
- ডিকশনারি
- ডিকশনারি ফাংশন
- টাপল
- আবারও লিস্ট
- লিস্ট ও ডিকশনারি কম্প্রিহেনশন

None

আচ্ছা আমরা তো জানি কোন ভ্যারিয়েবলে ডাটা ষ্টোর করা যায়। অথবা সেখানে ফাকা ভ্যালু রাখা যায় যেমন ফাকা স্ট্রিং। কিন্তু যদি এমন কোন ভ্যারিয়েবল নেই যার আসলে কোন ভ্যালুই নাই সেটা কিভাবে ইনিসিয়ালাইজ করা যেতে পারে? `None` হচ্ছে `NoneType` এর একটি অবজেক্ট যা দিয়ে আসলে ভ্যালুর অনুপস্থিতি নির্ধারণ করে দেয়া যায়।

যদি নিচের লাইনের আউটপুট দেখি -

```
type(None)
```

তাহলে আসবে,

```
<class 'NoneType'>
```

অন্যান্য ডাটা টাইপের যেমন একাধিক ভ্যালু থাকে পারে যেমন - bool টাইপের দুটো ভ্যালু হতে পারে; True অথবা False. `NoneType` এর একটাই ভ্যালু আর সেটা হল এই `None` .

`None` মানে `False` না। আবার এর মানে `0` , `""` , `[]` এসবও না। নিচের তুলনা এবং আউটপুট গুলো দেখি -

```
>>> None == False
False
>>> None == ""
False
>>> None == []
False
>>> None == 0
False
>>> None == None
True
>>> a = None
>>> a == None
True
>>> print(a)
None
```

যখন কোন ফাংশন নির্দিষ্ট করে কোন কিছু রিটার্ন করে না তখন বস্তুত সে `None` রিটার্ন করে। এরকম একটা ফাংশনের রিটার্ন চেক করে দেখা যেতে পারে -

```
def my_func():
    print("Printing Hello")

what_i_got = my_func()
print(what_i_got)
```

আউটপুট,

```
Printing  Hello
None
```

`None` এর আরেকটা সুন্দর ব্যবহার আমরা দেখতে পারি ফাংশনের ডিফল্ট আর্গুমেন্টের ডিফল্ট ভ্যালু হিসেবে ডিফাইন করার সময়। নিচের উদাহরণটি দেখি -

```
def my_func(x):
    if x:
        return x * x
    else:
        return 0

print(my_func())
```

আউটপুট,

```
Traceback (most recent call last):
  File "/Users/nuhil/Desktop/Test.py", line 7, in <module>
    print(my_func())
TypeError: my_func() takes exactly 1 argument (0 given)
```

উপরে `my_func` ডিফাইন করার সময় এর একটি প্যারামিটারও ডিফাইন করা হয়েছে। আবার লজিকের মাধ্যমে আমরা চেকও করেছি যে - যদি `x` এর ভ্যালু থাকে তাহলে সেটার স্কয়ার করে রিটার্ন করবে আর না থাকলে শূন্য রিটার্ন করবে। তাই যখন এই ফাংশনকে কল করা হচ্ছে তখন আমরা এরর পাচ্ছি যেখানে বলা আছে যে - উক্ত ফাংশনটি একটি আর্গুমেন্ট নেয় কিন্তু আমরা তাকে কিছুই পাঠাই নাই।

এই ফাংশনকে একটু মডিফাই করে আমরা এর ডিফল্ট আর্গুমেন্টের ভ্যালু হিসেবে `None` সিলেক্ট করতে পারি। এতে করে এই ফাংশনকে কল করার সময় যখন ভ্যালিড আর্গুমেন্ট পাঠানো হবে তখন ফাংশনটি সেই ভ্যালিড ভ্যালু নিয়ে কাজ করবে আর না পাঠালেও সমস্যা নাই - তখন শূন্য পাঠাবে।

```
def my_func(x = None):
    if x:
        return x * x
    else:
        return 0

print(my_func())
print(my_func(5))
```

আউটপুট,

```
0
25
```

উপরের প্রোগ্রামে উক্ত ফাংশনকে একবার আর্গুমেন্ট পাস করা ছাড়াই কল করা হয়েছে আরেকবার একটি আর্গুমেন্ট পাঠিয়েও কল করা হয়েছে। দুইবারই সঠিকভাবে কাজ করছে।

ডিকশনারি

এর আগে আমরা লিস্ট সম্পর্কে জেনেছি যেটা এমন এক ধরণের ডাটা স্ট্রাকচার যার মধ্যে এলিমেন্ট গুলো ক্রমিক ইনডেক্স অনুযায়ী সাজানো থাকে। ডিকশনারি আরেক ধরণের ডাটা স্ট্রাকচার যার মধ্যেও লিস্টের মত বিভিন্ন রকম এলিমেন্ট বা অবজেক্ট স্টোর করা যায় - কিন্তু, এ ক্ষেত্রে ওই এলিমেন্ট গুলোকে ম্যানুয়ালি ইনডেক্স করতে হয়। অন্যভাবে বলতে গেলে, আমাদের নিজেদেরকেই প্রত্যেকটা এলিমেন্টের বা value এর জন্য একটি key বা ইনডেক্স নির্ধারণ করে দিতে হয়। অতঃপর একটি key-value জোড় ওয়ালা এলিমেন্টের কালেকশন তৈরি হয়।

দুটি কার্লী ব্র্যাকেট `{}` এর মধ্যে কোলন চিহ্ন দিয়ে key-value জোড় তৈরি করে এবং প্রত্যেক জোড় কে কমা `,` দিয়ে আলাদা করে একটি ডিকশনারি তৈরি করা যায়। নিচের মত করে।

```
my_marks = {"Bengali": 80, "English": 85, "Math": 90}
```

আবার ফাকা ডিকশনারি তৈরির জন্য এভাবে লিখলেই সেটি ইনিসিয়ালাইজ হয়ে যায় - `my_dictionary = {}`

ডিকশনারির প্রত্যেকটি এলিমেন্টকে অ্যাক্সেস করার নিয়ম লিস্টের মতই। লিস্টে যেমন থার্ড ব্র্যাকেট এর মধ্যে ইনডেক্স দিয়ে উক্ত ইনডেক্সের ভ্যালু পাওয়া যেত, তেমনি এর ক্ষেত্রেও ইনডেক্সের যায়গায় key ব্যবহার করে, এর সাথে জোড় হিসেবে থাকা ভ্যালুটাকে অ্যাক্সেস করা যাবে।

উদাহরণ,

```
my_marks = {"Bengali": 80, "English": 85, "Math": 90}
print(my_marks["Math"])
```

আউটপুট,

```
90
```

কি - ভ্যালুর নিয়ম
ডিকশনারির মধ্যে যেকোনো টাইপের অবজেক্ট বা এলিমেন্টকেই স্টোর করা যায় শুধু মাত্র এর key গুলো হতে হবে Immutable (অপরিবর্তনীয়) টাইপের যেমন নিচের মত করে একটি ডিকশনারি তৈরি করা যেতে পারে -

```
my_marks = {"Bengali" : [30, 35, 32], "English" : [45, 52, 33], "Math": [60, 74, 58]}
```

অর্থাৎ প্রত্যেকটি key এর সাপেক্ষে ভ্যালু গুলো হচ্ছে এক একটি লিস্ট। এই ডিকশনারি থেকে নিচের মত করে ডাটা অ্যাক্সেস করা যাবে -

```
my_marks = {"Bengali" : [30, 35, 32], "English" : [45, 52, 33], "Math": [60, 74, 58]}
print(my_marks["Math"])
```

যার আউটপুট আসবে,

```
[60, 74, 58]
```

কিন্তু নিচের মত একটি ডিকশনারি হতে পারে না -

```
my_marks = {[30, 35, 32] : "Bengali", [45, 52, 33] : "English", [60, 74, 58] : "Math"}
```

এর আউটপুট হবে,

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

অর্থাৎ একটি লিস্ট যা কিনা একধরনের Mutable টাইপ তাকে কোন ডিকশনারির key হিসেবে ব্যবহার করা যাবে না। মজার জন্য চেক করতে পারি, যেহেতু bool টাইপও Immutable তাই নিচের মত একটা ডিকশনারিও কিন্তু হতে পারে -

```
my_marks = {True : "Bengali"}
```

> এখন পর্যন্ত আমাদের আলোচিত অবজেক্ট গুলোর মধ্যে লিস্ট এবং ডিকশনারি হচ্ছে Mutable টাইপের

ডিকশনারি ফাংশন

পাইথনের লিস্টে যেমন নির্দিষ্ট কোন ইনডেক্সে নতুন একটি ভ্যালু সেট করা যেত, তেমনি ডিকশনারির ক্ষেত্রেও একটি key তে থাকা কোন ভ্যালুকে আপডেট করে নতুন একটি ভ্যালু সেট করা যায়। যেমন -

লিস্টের ক্ষেত্রে উদাহরণ -

```
my_list = [2, 4, 6, 7]
my_list[3] = 8

print(my_list)
```

আউটপুট,

```
[2, 4, 6, 8]
```

ডিকশনারির ক্ষেত্রে -

```
my_nums = {1 : 1, 2 : 4, 3 : 9, 4 : "What?"}
my_nums[4] = 16

print(my_nums)
```

আউটপুট,

```
{1: 1, 2: 4, 3: 9, 4: 16}
```

অর্থাৎ এ ক্ষেত্রে লিস্ট এবং ডিকশনারি একই আচরণ করে। কিন্তু লিস্টের ক্ষেত্রে ম্যানুয়ালি নতুন একটি ইনডেক্স এবং তার ভ্যালু যুক্ত করা যায় না। যেমন -

```
my_list = [2, 4, 6, 8]
my_list[4] = 10
```

আউটপুট,

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

অর্থাৎ, যদিও `my_list = [2, 4, 6, 8]` এর ইনডেক্স `3` পর্যন্ত এবং আমরা চেষ্টা করেছি ম্যানুয়ালি একটি চতুর্থ ইনডেক্সে নতুন একটি ভ্যালু যুক্ত করতে, কিন্তু তা সম্ভব হয় নি। কারন লিস্টের ইনডেক্স স্বয়ংক্রিয়ভাবে একবার তৈরি হয়ে যায় এবং এভাবে ম্যানুয়ালি ইন্ডেক্সিং করা যায় না। বরং `append` ব্যবহার করা হয়।

কিন্তু চাইলে ডিকশনারির ক্ষেত্রে ম্যানুয়ালি নতুন key এবং সাথে এর জন্য একটি ভ্যালু সহ আরেকটি লিস্টে যুক্ত করা যায়। যেমন -

```
my_nums = {1 : 1, 2 : 4, 3 : 9, 4 : 16}
my_nums[5] = 25

print(my_nums)
```

আউটপুট,

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

অর্থাৎ নতুন key `5` এবং এর ভ্যালু `25` দুটোই `my_nums` ডিকশনারিতে ম্যানুয়ালি যুক্ত করা হয়েছে।

key খোঁজা

যদি আমরা চেক করতে চাই যে একটি ডিকশনারিতে নির্দিষ্ট কোন একটি key আছে কিনা তার জন্য `in` এবং `not in` ব্যবহার করা যায়। বলে রাখা ভালো, এভাবে কিন্তু লিস্টের ক্ষেত্রেও চেক করা যায়। উদাহরণ -

```
nums = {1: "one", 2: "two", 3: "three",}

print(1 in nums)
print("three" in nums)
print(4 not in nums)
```

আউটপুট,

```
True
False
True
```

get এর ব্যবহার

উপরে আমরা দেখেছি যে ডিকশনারি থেকে ডাটা অ্যাক্সেস এর জন্য লিস্টের মতই ইনডেক্স দিয়ে তথা key ব্যবহার করা যায়। কিন্তু এভাবে ডাটা অ্যাক্সেসের একটু অসুবিধা আছে। যেমন -

```
my_nums = {1 : 1, 2 : 4, 3 : 9, 4 : 16}
print(my_nums[5])
```

আউটপুট,

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 5
```

অর্থাৎ যে key আলোচ্য ডিকশনারিতে নাই সেরকম key দিয়ে ডাটা অ্যাক্সেসের চেষ্টা করলে অনাকাঙ্ক্ষিত এরর তৈরি হবে যা প্রোগ্রাম বন্ধ করতে পারে। তাই ভালো প্র্যাকটিস হচ্ছে `get` মেথডের ব্যবহার করা। নিচের মত করে -

```
my_nums = {1 : 1, 2 : 4, 3 : 9, 4 : 16}
print(my_nums.get(5))
```

আউটপুট,

```
None
```

অর্থাৎ এরর না তৈরি হয়ে বরং `None` রিটার্ন হবে। এমনকি চাইলে ডিফল্ট কোন ভ্যালুও পাওয়া যাবে যদি উক্ত key ওই ডিকশনারিতে না থাকে। যেমন -

```
my_nums = {1 : 1, 2 : 4, 3 : 9, 4 : 16}
print(my_nums.get(5, "5 not in my numbers!"))
```

আউটপুট,

```
5 not in my numbers!
```

> সংকলন - [নুহিল মেহেদী]()

টাপল

ইতোমধ্যে দেখে আসা লিস্টের মতই আরেকটি ডাটা স্ট্রাকচার হচ্ছে Tuple. কিন্তু গুরুত্বপূর্ণ পার্থক্যটি হচ্ছে এটি Immutable টাইপের অর্থাৎ, এর ভ্যালু পরিবর্তন করা যায় না। আবার, লিস্ট যেমন তৈরি করতে হয় দুটো `[]` ব্র্যাকেট দিয়ে কিন্তু টাপল তৈরি করতে হয় `()` দিয়ে (যদিও ব্র্যাকেট ছাড়াও শুধু কমা চিহ্ন দিয়ে ভ্যালু গুলোকে আলাদা করেও টাপল তৈরি করা যায়)। আরও কিছু পার্থক্য আছে এবং অবশ্যই লিস্ট ব্যবহার না করে কিছু যায়গায় কেন টাপল ব্যবহার করা উচিৎ তার কিছু কারনও আছে। সেগুলো এখানে আলোচনা করা হবে।

উদাহরণ,

```
roles = ("Admin", "Operator", "User")
# Or following line will create the same Tuple
# roles = "Admin", "Operator", "User"

print(roles[0])
```

আউটপুট,

```
Admin
```

আমরা দেখতে পাচ্ছি টাপল থেকে ভ্যালু অ্যাক্সেসের জন্যও লিস্টের মতই ইনডেক্স ব্যবহার করা যায়। কিন্তু লিস্টের মত নতুন ভ্যালু যুক্ত বা আপডেট করা যায় না। যেমন -

```
roles = ("Admin", "Operator", "User")
roles[2] = "Customer"
```

আউটপুট,

```
TypeError: 'tuple' object does not support item assignment
```

> খুব স্বাভাবিক ভাবেই `roles = ()` এভাবে ফাকা একটি টাপল ইনিসিয়ালাইজ করা যায়.

একটি টাপলের মধ্যে ভ্যালু হিসেবে অন্য লিস্ট, ডিকশনারি বা টাপল থাকতে পারে। যেমন -

```
permissions = (("Admin", "Operator", "Customer"), ("Developer", "Tester"), [1, 2, 3], {"S

print(permissions[3]["Stage"])
```

আউটপুট,

```
Development
```

টাপল আনপ্যাকিং

টাপল আনপ্যাকিং এর মাধ্যমে একটি টাপলের (বা যেকোনো ইটারেবল) মধ্যে থাকা প্রত্যেকটি ভ্যালুকে আলাদা আলাদা নতুন ভ্যারিয়েবলে অ্যাসাইন করা যায় এক লাইন কোড লিখেই। নিচের উদাহরণ দেখি,

```
numbers = (1, 2, 3)
a, b, c = numbers
print(a)
print(b)
print(c)
```

অর্থাৎ `numbers` নামের টাপলের তিনটি ভ্যালুকে পরের লাইনে আলাদা আলাদা তিনটি ভ্যারিয়েবল `a` , `b` , `c` তে জমা রাখা হয়েছে।

আউটপুট,

```
1
2
3
```

যদি এমন হয় যে একটি টাপল বা ইটারেবলে অসংখ্য ভ্যালু আছে কিন্তু আমরা এগুলো অল্প কিছু আলাদা আলাদা ভ্যারিয়েবলে জমা রাখতে চাই। অর্থাৎ, এর জন্য সমান চিহ্নের বাম পাশে অসংখ্য ভ্যারিয়েবল লিখতে চাই না তখন নিচের মত করে যেকোনো ভ্যারিয়েবলের সামনে `*` যুক্ত করে অবশিষ্ট যেকোনো সংখ্যক ভ্যালুকে এর মধ্যে জমা রাখা যায়।

উদাহরণ,

```
a, b, *c, d = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(a)
print(b)
print(c)
print(d)
```

এখানে `1` জমা হচ্ছে `a` এর মধ্যে, `2` জমা হচ্ছে `b` এর মধ্যে কিন্তু এরপর থেকে বাকিগুলো জমা হচ্ছে `c` এর মধ্যে। আর ডান পাশের ইটারেবলের শেষ ভ্যালু জমা হচ্ছে বাম পাশের শেষ ভ্যারিয়েবল `d` এর মধ্যে।

আউটপুট,

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

> সংকলন - নুহিল মেহেদী

আরও কিছু লিস্ট অপারেশন

স্বাভাবিক স্লাইস
আমরা আগে দেখেছি লিস্ট থেকে কিভাবে ইনডেক্স দিয়ে একটি মাত্র ভ্যালু অ্যাক্সেস করা যায়। এখন দেখবো কিভাবে একটি লিস্টকে ভাগ করে আরেকটি লিস্ট তৈরি করা যায় অথবা অন্যভাবে বলতে গেলে কিভাবে একটি লিস্ট থেকে একাধিক ভ্যালু নিয়ে আরেকটি লিস্ট হিসেবে অ্যাক্সেস করা যায়। এ কাজের জন্য `[]` এর মধ্যে শুধুমাত্র একটি ইনডেক্স না লিখে বরং কোলন দিয়ে একাধিক ইনডেক্স লিখতে হয়।

উদাহরণ,

```
some_marks = [2, 4, 6, 32, 60, 65, 69, 76, 80, 85, 90]

avg_marks = some_marks[4:8]
print(avg_marks)

good_marks = some_marks[8:]
print(good_marks)

poor_marks = some_marks[:4]
print(poor_marks)
```

উপরের প্রোগ্রামে আমরা `some_marks` লিস্ট থেকে স্লাইস করে বিভিন্ন সাব লিস্ট পেয়েছি। যেমন প্রথমে আমরা চতুর্থ ইনডেক্স থেকে শুরু করে অষ্টম ইনডেক্স পর্যন্ত নিয়েছি। দ্বিতীয় প্রিন্টের মধ্যে আমরা অষ্টম ইনডেক্স থেকে শুরু করে শেষ পর্যন্ত এলিমেন্ট গুলো নিয়েছি। আর তৃতীয় প্রিন্টের মধ্যে আমরা শুরু থেকে চতুর্থ ইনডেক্স পর্যন্ত ভ্যালু গুলো নিয়েছি।

> লিস্ট থেকে স্লাইস করার সময় কোলনের দু পাশে দুটো ইনডেক্স ব্যবহার করলে, বাম পাশের ইনডেক্সের ভ্যালু ইনক্লুড থাকে কিন্তু ডান পাসের ইনডেক্সের ভ্যালু ইনক্লুড হয় না। ব্যাপারটা `range` এর মতই।

উপরের প্রোগ্রামের আউটপুট,

```
[60, 65, 69, 76]
[80, 85, 90]
[2, 4, 6, 32]
```

ইনডেক্স জাম্প
লিস্ট স্লাইসের সময় শুরু ও শেষ ইনডেক্স বাদে ধাপও উল্লেখ করে দেয়া যায়। অর্থাৎ, উল্লেখিত ইনডেক্সের মধ্যবর্তী ভ্যালু গুলো সিলেক্ট হবে কিন্তু সেখান থেকে উল্লেখিত ধাপ পরিমাণ ইনডেক্স জাম্প করে করে ভ্যালু নেয়া হবে। যেমন -

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(numbers[2:9:3])
```

আউটপুট,

```
[3, 6, 9]
```

এখানে `numbers` লিস্ট থেকে দ্বিতীয় এবং নবম ইনডেক্সের মধ্যবর্তী ভ্যালু গুলো নেয়া হয়েছে কিন্তু প্রতিবার তিনটি করে স্টেপ জাম্প করে।

নেগেটিভ ইনডেক্স স্লাইস
আমরা দেখেছি কিভাবে শুরু ও শেষ ইনডেক্স নির্ধারণ করে দিয়ে, একটি লিস্ট থেকে মধ্যবর্তী কিছু ভ্যালু নিয়ে এর স্লাইস তৈরি করা যায়। চাইলে এভাবে স্লাইস না করে - মুল লিস্টের শুরুর ইনডেক্স নির্ধারণ করে দিয়ে এবং শেষ থেকে উল্টা ইনডেক্স নির্ধারণ করে দিয়েও একটি স্লাইসড লিস্ট পাওয়া যায়।

উদাহরণ,

```
squares = [1, 2, 5, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 4, 6, 7, 8]
print(squares[3:-4])
```

আউটপুট,

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`squares[3:-4]` এর মাধ্যমে আমরা এই লিস্ট থেকে স্লাইস করার জন্য শুরুর ইনডেক্স বলে দিয়েছি `3` এবং লিস্টের শেষ থেকে হিসেবে শুরু করে চতুর্থ ইনডেক্সকে নির্ধারণ করে দিয়েছি স্লাইস করার শেষ ইনডেক্স হিসেবে।

লিস্ট রিভার্স
যদি একটি লিস্ট থেক স্লাইস তৈরি করার সময় স্টেপ হিসেবে নেগেটিভ ভ্যালু সেট করা হয় (দ্বিতীয় কোলনের ডান পাশে) এবং মুল লিস্টের সব ভ্যালুকেই সিলেক্ট করে নিতে বলা হয় (প্রথম কোলনের দু পাশে ইনডেক্স উল্লেখ্য না করে) তাহলে বস্তুত মুল লিস্টের একটি রিভার্স বা উল্টো লিস্ট তৈরি হয়। নিচের উদাহরণটি দেখি -

```
values = [3, 4, 5, 6, 7, 8]
print(values[::-1])
```

আউটপুট,

```
[8, 7, 6, 5, 4, 3]
```

লিস্টের উপর করা কিছু নিউমেরিক অপারেশন
আশা করছি নিচের উদাহরণটির মধ্যে থাকা কমেন্ট গুলো দেখেই সবাই বুঝতে পারবেন প্রত্যেকটি মজার এবং গুরুত্বপূর্ণ ফাংশনের কাজ -

```
# Prints the minimum value among all the elements of the list below
print(min([1, 2, 3, 4, 0, 2, 1]))

# Prints the maximum value among all the elements of the following list
print(max([1, 4, 9, 2, 5, 6, 8]))

# Print sum of all the elements of the following list
print(sum([1, 2, 3, 4, 5]))
```

আউটপুট,

```
0
9
15
```

> সংকলন - নুহিল মেহেদী

লিস্ট ও ডিকশনারি কম্প্রিহেনশন

আমরা এই চ্যাপ্টারে লিস্ট ও ডিকশনারি কম্প্রিহেনশন সম্পর্কে জানার চেষ্টা করব। প্রথমে লিস্ট কম্প্রিহেনশন বিষয়টা বোঝার জন্য একটা উদাহরণ দিয়ে দেখা যাক।

ধরি, `num_list` একটি পাইথন লিস্ট যেখানে ১-১০ সংখ্যাগুলো আছে।

```
num_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

এখন আপনাকে যদি বলা হয় পাইথনে একটি কোড লিখতে যেটা কিনা সংখ্যাগুলো থেকে শুধু জোড় সংখ্যাগুলো নিয়ে আরেকটি লিস্ট তৈরি করতে এবং আপনার যদি লিস্ট কম্প্রিহেনশন সম্পর্কে ধারণা না থাকে তাহলে আপনি অবশ্যই এভাবে লিখবেন,

```
even_num_list = []
for num in num_list:
    if num % 2 == 0:
        even_num_list.append(num)
print (even_num_list)
```

আউটপুট

```
[2, 4, 6, 8, 10]
```

কিন্তু আপনার যদি এ ব্যাপারে ভাল ধারণা থাকে তাহলে আপনি কোডটি এক লাইনেই লিখতে পারবেন।

```
even_num_list = [even_num for even_num in num_list if even_num % 2 == 0]
print (even_num_list)
```

চমৎকার, তাই না?

এটাই পাইথনের মজা, প্রতিটা ল্যাঙ্গুয়েজের আলাদা কিছু বৈশিষ্ট্য থাকে যেটা তাকে আলাদা করে তোলে। লিস্ট কম্প্রিহেনশন পাইথনের ক্ষেত্রে এরকম অন্যতম একটি বৈশিষ্ট্য।

কিন্তু এটা করলাম কীভাবে? একটু ভিজুয়ালাইজ করার চেষ্টা করি!

```python
num_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

even_num_list = []

for num in num_list:
    if num % 2 == 0:
        even_num_list.append(num)

# Now using list comprehension

```

লিস্ট কম্প্রিহেনশন কী?

একটা লিস্ট / ইটার‍্যাবল থেকে আরেকটা লিস্টে রূপান্তর করার একটা পদ্ধতি।

পাইথনের লিস্ট কম্প্রিহেনশন জিনিসটা বেশ মজার কিন্তু একই সাথে পাইথন বিগিনার দের জন্য একটু ঝামেলাকর। সেটা দূর করার জন্যই এই চ্যাপ্টার।

রিড্যাবিলিটি

অনেকসময় অন্যের করা লিস্ট কম্প্রিহেনশনের কোডগুলো বোঝা কঠিন হয়ে যায়। সেক্ষেত্রে আপনি চাইলে ভেঙ্গেও লিখতে পারেন।

```python
even_num_list = [
  num
  for num in num_list
  if num % 2 == 0
]

print(even_num_list)
```

চলুন আরও কিছু উদাহরণ দেখা যাক।

নেস্টেড লুপের লিস্ট কম্প্রিহেনশন

নেস্টেড লুপের লিস্ট কম্প্রিহেনশন দেখানোর জন্য 2D ম্যাট্রিক্সের উদাহরণ দেখানোর উপরে কিছু নাই। মনে করুন, আপনার একটা Matrix / 2D Array আছে, সেটাকে আপনি ফ্ল্যাটেন বা 1D করতে চাচ্ছেন, এই সমস্যাটা আমরা এভাবে সমাধান করতে পারি,

```
matrix_1d = []
matrix_2d = [[1, 2, 3],
            [4, 5, 6]]

for row in matrix_2d:
    for num in row:
        matrix_1d.append(num)
print(matrix_1d)
```

আউটপুট

```
[1, 2, 3, 4, 5, 6]
```

লিস্ট কম্প্রিহেনশন ব্যবহার করে,

```
matrix_2d = [[1, 2, 3],
            [4, 5, 6]]

matrix_1d = [num for row in matrix_2d for num in row]
```

আউটপুট

```
[1, 2, 3, 4, 5, 6]
```

আগের মতই কপি পেস্ট করলাম, প্রথমে যেটা অ্যাড করব সেটা লিখলাম তারপর টপ লেভেল লুপ এবং অবশেষে নেস্টেড লুপ লিখে কোডটি কম্প্লিট করলাম।

ধরুন, আপনি `matrix_1d` তে প্রতিটা এলিমেন্টের বর্গ বা স্কয়ার ভ্যালু নিতে চাচ্ছেন, তাহলে কোড হবে এমন,

```
matrix_1d = [num**2 for row in matrix_2d for num in row]
```

আউটপুট

```
[1, 4, 9, 16, 25, 36]
```

বাক্য থেকে Vowel দূর করা

সাধারণ নিয়মে,

```
vowels = 'aeiou'
sentence = 'I am awesome!'
filtered_letters = []

for letter in sentence:
    if letter not in vowels:
        filtered_letters.append(letter)

print(''.join(filtered_letters))
```

আউটপুট

```
I m wsm!
```

লিস্ট কম্প্রিহেনশন ব্যবহার করে

```
vowels = 'aeiou'

sentence = 'I am awesome'

filtered_sentence = ''.join([letter for letter in sentence if letter not in vowels])

print(filtered_sentence)
```

আউটপুট

```
I m wsm
```

ডিকশনারি কম্প্রিহেনশন

এটা লিস্ট কম্প্রিহেনশনের মতই, শুধু লিস্টের বদলে ডিকশনারি ডেটা স্ট্রাকচার ব্যবহার করব। আমরা যদি দুইটা লিস্ট থেকে একটা ডিকশনারি বানাতে চাই, যেখানে প্রথমটি `Key` এবং পরেরটি `Value` তাহলে সাধারণ নিয়মে এভাবে কোড লিখব,

```
fruit_ranking = [1, 2, 3]
fruit_name = ['Mango', 'Pineapple', 'Watermelon']
fruit_rank_dict = {}

for i in range(len(fruit_ranking)):
    fruit_rank_dict[fruit_ranking[i]] = fruit_name[i]

print(fruit_rank_dict)
```

আউটপুট

```
{1: 'Mango', 2: 'Pineapple', 3: 'Watermelon'}
```

ডিকশনারি কম্প্রিহেনশন ব্যবহার করে,

```
fruit_ranking = [1, 2, 3]

fruit = ['Mango', 'Pineapple', 'Watermelon']

fruit_ranking_dict = { fruit_ranking[i] : fruit[i] for i in range(len(fruit_ranking)) }

print(fruit_ranking_dict)
```

আউটপুট

```
{1: 'Mango', 2: 'Pineapple', 3: 'Watermelon'}
```

কোন কোন ক্ষেত্রে লিস্ট বা ডিকশনারি কম্প্রিহেনশন ব্যবহার করা শুধু সুবিধাজনক তা-ই নয়, সিম্পল এক্সপ্রেশনে এটা ফর লুপের চেয়েও ফাস্ট। কম্পলেক্স এক্সপ্রেশনে ফর লুপ আর লিস্ট কম্প্রিহেনশনের পারফর্মেন্স প্রায় একই।

> সংকলন: মানস

এই সেকশনে থাকছে

- কোডের পুনব্যবহার
- ফাংশন
- ফাংশন আর্গুমেন্ট
- ফাংশন রিটার্ন
- কমেন্ট ও ডক স্ট্রিং
- অবজেক্ট হিসেবে ফাংশন
- মডিউল
- স্ট্যান্ডার্ড লাইব্রেরী
- pip

কোডের পুনব্যবহার

কম্পিউটারের জন্য প্রোগ্রাম বা ইন্সট্রাকশন লিখে আমরা কম্পিউটারকে দিয়ে একটি কাজ বার বার করিয়ে নিতে পারি। এটুকু আমরা সবাই জানি। একটি কাজের জন্য এক একজন এক একভাবে প্রোগ্রাম লিখতে পারে। যেমন আমাদের যদি স্ক্রিনে ১০টি লাইন প্রিন্ট করতে হয় তখন কেউ হয়তো একটি প্রোগ্রাম লিখে তার মধ্যে ১০টি প্রিন্ট স্টেটমেন্ট ব্যবহার করবে। আবার ভালো প্রোগ্রামাররা এই কাজের জন্য লুপ ব্যবহার করবে। আগের চ্যাপ্টারেই আমরা `for` , `while` ইত্যাদি লুপের ব্যবহার দেখেছি। লুপ ব্যবহার করে এর মধ্যেকার নির্দিষ্ট কিছু ইন্সট্রাকশনকে একাধিকবার এক্সিকিউট করানো যায়। এটা এক ধরণের কোডের পুনব্যবহার। অর্থাৎ ১০ বার প্রিন্ট স্টেটমেন্ট ব্যবহার না করে একটি প্রিন্ট স্টেটমেন্ট এবং সাথে একটু লজিক ব্যবহার করে একই কাজ করা সম্ভব।

যেকোনো রকম কম্পিউটার প্রোগ্রামিং -এ দুটি টার্ম শুনতে পাওয়া যায়। **WET** এবং **DRY**. WET মানে বলা হয় **Write Everything Twice** অথবা **We Enjoy Writing**. অর্থাৎ যেকোনো প্রোগ্রামের মধ্যে অপ্রয়োজনীয় এবং অতিরিক্ত স্টেটমেন্ট লেখা বা ব্যবহার করা যেখানে সেই একই কাজ অনেক কম কোড লিখেও আরও অপ্টিমাইজ ভাবে করা যেত। এটা খারাপ অভ্যাস। আবার DRY মানে বলা হয় - **Do not Repeat Yourself**. অর্থাৎ ওই যে ১০ বার ১০টা লাইন প্রিন্ট করার জন্য ১০ বার প্রিন্ট স্টেটমেন্ট ব্যবহার না করা। যেহেতু কাজটা একই (কিছু একটা প্রিন্ট করা) সেহেতু একবার প্রিন্ট স্টেটমেন্ট লিখেই কোন না কোন ভাবে ১০ বার প্রিন্ট করা সম্ভব। এটা ভালো অভ্যাস।

এরকম কোড পুনব্যবহার করার সবচেয়ে ভালো উদাহরণ হচ্ছে ফাংশন এর প্রয়োগ। আমরা কিন্তু ইতোমধ্যে ফাংশনের ব্যবহার করে ফেলেছি আগের চ্যাপ্টার গুলোতে। যেমন এই যে প্রিন্ট `print` নিয়ে এতো কথা বলা হল, এটাও একটা ফাংশন। এটা একটা বিল্ট ইন ফাংশন। অর্থাৎ এই ফাংশনের কর্মপ্রণালী আমরা নিজেরা কোড করে লিখি নাই। এই ফাংশনের কাজ হচ্ছে - এর মধ্যে যেকোনো string কে পাঠিয়ে দিলে সেই string -কে সে স্ট্যান্ডার্ড আউটপুটে প্রিন্ট করে দেখায়। অর্থাৎ, কোন কিছু প্রিন্ট করার জন্য আমরা প্রত্যেকবার কম্পিউটারের বোধগম্য এক গাদা কোড লিখি না। শুধু প্রিন্ট ফাংশনকে কল করে যা প্রিন্ট করতে হবে তা ওকে দিয়ে দেই। এভাবে আমরা প্রিন্ট ফাংশনের জন্য করা কোডকে বার বার ব্যবহার করছি। এটাই কোডের পুনব্যবহার।

সামনের কয়েকটি চ্যাপ্টারে আমরা নিজেদের বিভিন্ন কাজের জন্য ফাংশন লিখে কোডের পুনব্যবহার দেখবো।

ফাংশন

আমাদের প্রোগ্রামের যে অংশগুলো বার বার আসে সেগুলোকে আমরা পুনরায় ব্যবহারযোগ্য একক (reusable unit) হিসেবে ব্যবহার করতে পারি ফাংশনের সাহায্যে । গনিতে যেমন দেখেছি কোন ফাংশন একটি ইনপুট নিয়ে সেটার উপর বিভিন্ন ধরনের ম্যাথ করে আউটপুট দেয়, প্রোগ্রামিংএও সেই একই ব্যাপার ঘটে । আপনি এক বা একাধিক প্যারামিটার পাস করবেন একটি ফাংশনে, ফাংশনটি প্রসেস করে আপনাকে আউটপুট "রিটার্ন করবে" । তবে প্রোগ্রামিং এর ক্ষেত্রে সবসময় যে ইনপুট থাকতে হবে বা আউটপুট দিতে হবে এমন কোন কথা নেই ।

একটি ফাংশন আসলে কিছু স্টেটমেন্টের সংকলন । যখনই কোন ফাংশন কল করা হয় তখন এই ফাংশনের ভিতরে থাকা স্টেটমেন্টগুলো এক্সিকিউট করা হয় । পাইথনে আমরা ফাংশন ডিক্লেয়ার করার জন্য def কি-ওয়ার্ডটি ব্যবহার করি । আসুন দেখে নেই একটি ফাংশন:

```
def hello():
    print("Hello World!")
```

প্রথমে আমরা `def` কি-ওয়ার্ডটি লিখেছি । তারপর ফাংশনের একটা নাম দিয়েছি – `hello` , এবং তারপর একজোড়া প্রথম বন্ধনী `()` . এরপর একটি কোলন তথা `:` চিহ্ন দিয়ে এর নিচে ফাংশনের আওতাভুক্ত কোড ব্লক বা ফাংশনের কাজ ডিফাইন (নির্ধারণ) করেছি। ফাংশনটির কাজ হচ্ছে "Hello world!" বাক্যটি প্রিন্ট করা।

কিন্তু এভাবে একটা ফাংশনকে প্রোগ্রামের মধ্যে শুধু ডিফাইন করে রেখে দিলে তার মধ্যের কোড বা ইন্সট্রাকশন গুলো এমনি এমনি কাজ করবে না। এর জন্য ওই ফাংশনটিকে তার নাম ধরে কল করতে হবে। কল কিভাবে করতে হয়? কিছুই না ওই ফাংশনের নামটি সাধারণ ভাবে স্টেটমেন্ট আকারে লিখলেই হয়। যেমন - আমরা `print` ফাংশন ব্যবহার করি যখন দরকার হয় তখন। উপরের ফাংশনটিকে কল করতে হবে নিচের মত করে,

```
hello()
```

তাই পুরো প্রোগ্রামটি যদি নিচের মত হয়,

```
# Defining the function named hello
def hello():
    print("Hello World!")

# Calling the function to use it
hello()
```

তাহলে এই প্রোগ্রামটির আউটপুট হবে,

```
Hello World!
```

একটা ফাংশনকে প্রোগ্রামে একবার ডিফাইন করলেও সেটাকে বার বার কল বা ব্যবহার করা যাবে। অর্থাৎ নিচের মত -

```
# Defining the function named hello
def hello():
    print("Hello World!")

# Calling the function to use it
hello()

# Again calling the function
hello()
```

আউটপুট,

```
Hello World!
Hello World!
```

একটি কথা মনে রাখা জরুরি। কোন ফাংশনকে কল করার বা ব্যবহার করার আগেই সেই ফাংশনকে প্রোগ্রামে ডিফাইন বা প্রোগ্রামের কাছে চিহ্নিত করতে হবে। যেমন - উপরের প্রোগ্রামটিকে যদি আমরা নিচের মত করে লিখি,

```
# Calling the function to use it
hello()

# Defining the function named hello
def hello():
    print("Hello World!")
```

তাহলে আউটপুট আসবে,

```
NameError: name 'hello' is not defined
```

এটা সহজ ভাবে চিন্তা করলেই যৌক্তিক মনে হবে। অর্থাৎ, যদি প্রোগ্রামের শুরুতেই একটি ফাংশনকে কল বা ব্যবহার করতে চাই, তাহলে কিভাবে আমার প্রোগ্রাম জানবে যে তার মধ্যে `hello` নামের একটা ফাংশন আছে? বরং এটাই কি যৌক্তিক নয় যে - আগে ফাংশনটাকে প্রোগ্রামে ডিফাইন করবো এবং তার পর প্রোগ্রামের কোন এক বা একাধিক যায়গায় সেটাকে কল করবো।?

কাস্টম ফাংশনকে অনেকটা আপনার দ্বারা তৈরি একটা ছোট্ট মেশিনের সাথে তুলনা করা যায়। অর্থাৎ একবার মেশিনটি তৈরি করবো আর বার বার ব্যবহার করে কিছু একটা কাজ করব বা জিনিষ তৈরি করবো। কিছু তৈরি করতে হলে মেশিনে কিছু ইনপুট দিতে হতে পারে। আবার কিভাবে তৈরি করবে সেটাও মেশিনের মধ্যে যন্ত্রপাতি বসিয়ে সেটআপ করতে হবে। এভাবে বাস্তবে একটা মেশিন তৈরি করাকেই প্রোগ্রামের মধ্যে ফাংশন ডিফাইন করা বলা যেতে পারে।

ফাংশন আর্গুমেন্ট

মনে আছে আমরা আগের চ্যাপ্টারে ফাংশনকে একটি ছোট্ট মেশিন হিসেবে কল্পনা করেছিলাম। যেকোনো মেশিন বা যন্ত্র যখন বানানো হয় তখন তার কাজের জন্য যেমন কিছু যন্ত্রপাতির সেটআপ দরকার হয় তেমনি সেই মেশিনে ইনপুট হিসেবে কিছু কাঁচামাল দিতে হয় যেগুলো প্রক্রিয়াজাত করে মেশিন আমাদের চাহিদা মোতাবেক জিনিষ তৈরি করে দেয় বা এর থেকে আউটপুট পাওয়া যায়।
ধরে নিচ্ছি আমাদের বানানো মেশিনটির এক পাশ দিয়ে ময়দা, চিনি, দুধ, ক্রিম এসব দিলে আরেক পাশ দিয়ে সুন্দর কেক তৈরি হয়ে বের হয়। তাহলে সেই ময়দা, চিনি, দুধ, ক্রিম এসব হচ্ছে সেই মেশিনের আর্গুমেন্ট আর কেক বানানোর জন্য মেশিনের মধ্যে বিভিন্ন যন্ত্রের যে সেটআপ আছে সেটাকে বলা যেতে পারে ফাংশন বডি। আর শেষে যে সুস্বাদু কেক পাওয়া যায় তাকে বলা যেতে পারে ফাংশনের রিটার্ন ভ্যালু। এখন এরকম একটি মেশিন তৈরি হয়ে গেলে এই মেশিনকে যতবার ইচ্ছা ব্যবহার করা যাবে এবং এর থেকে কেক পাওয়া যাবে। কিন্তু অবশ্যই প্রতিবার সঠিকভাবে কেক পেতে হলে এই মেশিনের আর্গুমেন্ট তথা কাঁচামাল গুলো দিতে হবে।

প্রোগ্রামিং -এও একই ভাবে একটি ফাংশনের কিছু আর্গুমেন্ট থাকতে পারে যেগুলো পক্ষান্তরে ফাংশন বডির মধ্যে ব্যবহৃত হয়ে চাহিদা মোতাবেক প্রসেসড হবে। এই আর্গুমেন্ট গুলো পাঠানোর দায়িত্ব হচ্ছে তার, যে এই ফাংশনকে কল করবে বা ব্যবহার করতে চাইবে। নিচের উদাহরণটি দেখি -

```
def show_double(x):
    print(x*2)

show_double(2)
show_double(100)
```

আউটপুট,

```
4
200
```

উপরে `show_double` ফাংশনের আর্গুমেন্ট একটি। আর তাই যখনই আমরা এই ফাংশনকে কল করেছি বা ব্যবহার করতে চেয়েছি তখনি সেই ফাংশনের আর্গুমেন্ট (মেশিনের ক্ষেত্রে ইনপুট) পাঠিয়ে দিয়েছি এভাবে `show_double(2)` । একবার কল করার সময় ইনপুট দিয়েছি `2` আবার আরেকবার কল করার সময় ইনপুট দিয়েছি `100` এবং আমাদের ফাংশনের কাজ হচ্ছে এর কাছে আসা যেকোনো আর্গুমেন্টকে দ্বিগুণ করে স্ক্রিনে প্রিন্ট করে। তাই দুইবারই আমাদের ফাংশন কাজটি সঠিক ভাবে করেছে।

> আর্গুমেন্টকে ফাংশনের দুটি প্রথম বন্ধনীর মধ্যে ডিফাইন করতে হয়।

একটি ফাংশন কিন্তু একাধিক আর্গুমেন্ট নিয়ে কাজ করতে পারে অর্থাৎ এর একাধিক আর্গুমেন্ট থাকতে পারে। এটাই তো যৌক্তিক, তাই না? কারণ, একটি ফাংশন তথা মেশিনকে জটিল জটিল জিনিষ বানাতে বা আউটপুট দিতে তাকে অনেক গুলো ইনপুট নিয়ে কাজ করতে হতেই পারে। নিচের উদাহরণটি দেখি -

```
def make_sum(x, y):
    z = x + y
    print(z)

make_sum(5, 10)
make_sum(500, 500)
```

আউটপুট,

```
15
1000
```

একটি বিষয় খেয়াল করুন, ফাংশনের আর্গুমেন্ট গুলোকে তার নিজের বডির মধ্যে একই নামের ভ্যারিয়েবল হিসেবে ব্যবহার করা যায়। যেমন উপরের উদাহরণে, `make_sum` ফাংশনের কাছে দুটো আর্গুমেন্ট এসেছে `x` , এবং `y` নামে এবং এই দুটি ভ্যালুকে সে নিজের বডির মধ্যে ব্যবহার করেছে যোগ করার জন্য এবং যোগফল জমা করেছে `z` নামের আরেকটি ভ্যারিয়েবলে।
কিন্তু এই `x` , `y` বা `z` কে উক্ত ফাংশনের বাইরে থেকে অ্যাক্সেস করা যাবে না বা ব্যবহার করা যাবে না। যেমন -

```
def make_sum(x, y):
    z = x + y
    print(z)

make_sum(5, 10)
print(z)
```

আউটপুট,

```
15
...
NameError: name 'z' is not defined
```

উপরের উদাহরণে, `print(z)` স্টেটমেন্টটি এরর দেখাচ্ছে কারণ `z` ভ্যারিয়েবলের গণ্ডি বা স্কোপ ছিল শুধুমাত্র `make_sum` ফাংশনের মধ্যেই। তাই বাইরে থেকে একে অ্যাক্সেস করা যায় নি।

মাল্টিপল প্যারামিটার হ্যান্ডলিং | আর্বিটরারি আর্গুমেন্ট লিস্ট

মনে করুন, আপনি `make_sum` ফাংশনটিতে অনেকগুলো প্যারামিটার পাঠাতে চাচ্ছেন যেমন, 10, 20, 30 ... ইত্যাদি। যদি আপনি `make_sum (a, b)` হিসেবে ডিক্লেয়ার করেন তাহলে দুইটার বেশি প্যারামিটার পাঠাতে পারবেন না। সেক্ষেত্রে কোড হবে এইরকম,

```python
def make_sum(*args):
    sum = 0
    for num in args: # Here, args is like a Tuple which is (10, 20, 30, 40)
        sum += num
    return sum

print(make_sum(10, 20, 30, 40))
```

আউটপুট

```
100
```

পাইথনে `*` এর অর্থ

`*` এর আর্গুমেন্টে ভ্যালু `Tuple` হিসেবে প্যাকড থাকে। এর মানে `*` দিয়ে প্যারামিটার ডিক্লেয়ার করলে আমরা যেকোন সংখ্যক পজিশনাল আর্গুমেন্ট পাস করতে পারি। যেমন করলাম `make_sum` এর ক্ষেত্রে। শুরুতে `make_sum` মাত্র দুইটা আর্গুমেন্ট নিলেও পরবর্তীতে আমরা প্যারামিটারে `*` বসিয়ে দিলাম তখন সে অনেকগুলো আর্গুমেন্ট পাস করতে পারছে।

পাইথনে `**` এর অর্থ

আমরা চাইলে ফাংশনের প্যারামিটারে ডাবল অ্যাস্টেরিস্কস বসিয়েও ডিক্লেয়ার করতে পারি। ডাবল স্টারের মানে হল যেকোন সংখ্যক `named parameter` থাকতে পারে। এই মানগুলো ডিকশনারি হিসেবে প্যাকড থাকে। নিচের উদাহরণটি লক্ষ্য করা যাক,

```python
def print_dict(*args):
    print (args)


print_dict(a=1, b=2)
```

আউটপুট,

```
TypeError                                 Traceback (most recent call last)
<ipython-input-2-9970453fce76> in <module>()
----> 1 print_dict(a=1, b=2)

TypeError: print_dict() got an unexpected keyword argument 'a'
```

সিঙ্গেল অ্যাস্টেরিস্কস ব্যবহার করলে আমরা নেমড আর্গুমেন্ট পাস করতে পারব না। তাই আমাদের এসব ক্ষেত্রে ডাবল অ্যাস্টেরিস্কস ব্যবহার করতে হবে, যেমন

```
def print_dict(**kwargs):
    print(kwargs)


print_dict(a=1, b=2, c=3)
```

আউটপুট,

```
{'a': 1, 'c': 3, 'b': 2}
```

আমরা যদি কোডটা আরেকটু গুছিয়ে লেখি,

```
def print_dict(**kwargs):
    for args in kwargs:
        print("{0} : {1}".format(args, kwargs[args]))


print_dict(a=1, b=2, c=3)
```

আউটপুট,

```
a : 1
c : 3
b : 2
```

চাইলে আমরা মিক্সড ভ্যারিয়েডিক আর্গুমেন্ট পাঠাতে পারি। মানে একই ফাংশনে তিন ধরণের আর্গুমেন্ট, তবে খেয়াল রাখতে হবে প্যারামিটারগুলো এমন ভাবে ডিফাইন করা হয় যেন প্রথমে সাধারণ প্যারামিটার তারপরে সিঙ্গেল অ্যাস্টেরিস্কের প্যারামিটার এবং অবশেষে ডাবল অ্যাস্টেরিস্কস এর প্যারামিটার থাকে। মানে আমাদের অবশ্যই ক্রম মানতে হবে এইক্ষেত্রে।

```
def print_all(a, *args, **kwargs):
    print(a)
    print(args)
    print(kwargs)


print_all(10, 20, 30, 50, b=5, c=10)
```

আউটপুট,

```
10
(20, 30, 50)
{'c': 10, 'b': 5}
```

প্যারামিটার ও আর্গুমেন্ট

যখন একটি ফাংশনকে ডিফাইন করা হয় তখন এর ভ্যারিয়েবল গুলোকে প্যারামিটার বলা হয়। আর যখন একটি ফাংশনকে কল করা হয় তখন সেই ফাংশনের প্যারামিটার হিসেবে যে ভ্যালু পাঠানো হয় তাকে আর্গুমেন্ট বলা হয়।

> সংকলন - নুহিল মেহেদী
>
> পরিমার্জন - মানস

ফাংশনের রিটার্ন

এ পর্যন্ত আমরা বুঝতে পেরেছি ফাংশন কি (মেশিন), কেনই বা ফাংশন ব্যবহার করবো (কোড এর পুনব্যবহার), ফাংশনের আর্গুমেন্ট (ইনপুট) ইত্যাদি। মনে আছে, আমরা ফাংশনকে কেক বানানোর মেশিন এর সাথে তুলনা করেছিলাম? অর্থাৎ কোন একটি ফাংশন তার বডির মধ্যে কিছু কাজ করে চুপ চাপ থাকতে পারে অথবা কোন কিছু রিটার্নও দিতে পারে। কাকে রিটার্ন দিবে? যে এই ফাংশনকে কল করবে বা ব্যবহার করবে। কোথায় রিটার্ন দিবে? যেখান থেকে কল করা হবে সেখানেই রিটার্ন ভ্যালু পৌঁছে যাবে। নিচের উদাহরণ দেখলে আরও পরিষ্কার বুঝতে পারবো আমরা -

```
def get_larger(x, y):
    if x > y:
        return x
    else:
        return y

larger_value = get_larger(23, 32)
print(larger_value)
```

আউটপুট,

```
32
```

ফাংশনের মধ্যে `return` কিওয়ার্ড ব্যবহার করে ফাংশন থেকে কোন কিছু রিটার্ন করা হয়। উপরের উদাহরণে `get_larger` ফাংশনের কাছে আশা দুটো আর্গুমেন্টের মধ্যে সে তুলনা করে বড়টি বের করে এবং সেটি রিটার্ন করে। আর তাই নিচে যখন `= get_larger(23, 32)` স্টেটমেন্টের মাধ্যমে একে কল করা হয়েছে এবং এর চাহিদা মোতাবেক দুটো আর্গুমেন্ট পাঠিয়ে দেয়া হয়েছে তখন বস্তুত সেই ফাংশনের রিটার্ন করা ভ্যালুটি `=` চিহ্নের ডান পাশে এসে জমা হয়। আর যেহেতু আমরা জানি `=` চিহ্ন দিয়ে কোন ভ্যালুকে কোন ভ্যারিয়েবলে অ্যাসাইন করা হয়, তাই `larger_value` এর মধ্যে সেই রিটার্ন করা ভ্যালু স্টোর হচ্ছে।
যা ঘটেছে তা হলঃ

```
larger_value = get_larger(23, 32) # Function is done with working and returned something
larger_value = 32
```

পরিশেষে একটি সাধারণ প্রিন্ট স্টেটমেন্ট।

খুব গুরুত্বপূর্ণ একটি বিষয় হচ্ছে, যখন কোন ফাংশনের মধ্যে একটি `return` স্টেটমেন্ট থাকে এবং সেটি এক্সিকিউট হয়, তারপর থেকে ফাংশনের মধ্যে থাকা আর কোন কোড এক্সিকিউট হয় না। অর্থাৎ, ফাংশন তার কাজ শেষে কিছু একটা রিটার্ন করেই থেমে যায়। যেমন -

```
def add_numbers(x, y):
    total = x + y
    return total
     print("This won't be printed")

print(add_numbers(4, 5))
```

আউটপুট,

```
9
```

উপরের প্রোগ্রামের ফাংশনটির কাজ হচ্ছে দুটো আর্গুমেন্ট ভ্যারিয়েবলকে যোগ করে নতুন একটি ভ্যারিয়েবলে রেখে সেটিকে তার caller এর কাছে রিটার্ন করা । রিটার্ন করেই সে ক্ষান্ত। এরপরে আরেকটি প্রিন্ট স্টেটমেন্ট থাকলেও সেটার কোন এক্সিকিউশন নেই। আর তাই সেই This won't be printed লাইনটিকে স্ক্রিনে দেখা যাচ্ছে না।

কমেন্ট ও ডক স্ট্রিং

কমেন্ট
কমেন্ট মানে মন্তব্য। কোড লেখার সময় কোডের সাথে মন্তব্যও লেখা যায়। কিন্তু সেই মন্তব্যের কথা গুলো প্রোগ্রামের মত রান হয় না। তাই এভাবে কোড সাথে কমেন্ট লিখে রাখলে অন্যের সেই কোড বুঝতে সুবিধা হয় এবং ওই কোড দিয়ে কি করা যায় এবং কিভাবে করা যায় সেটাও কোড রান করার আগেই বুঝে নেয়া যায়।

পাইথনে যেকোনো কমেন্ট লাইন লেখার আগে তার আগে `#` (হ্যাস বা পাউন্ড) চিহ্ন ব্যবহার করতে হয়। নিচের উদাহরণটি দেখি -

```
# few variables below
x = 10
y = 5

# make sum of the above two variables
# and store the result in z
z = x + y

print(z) # print the result
# print (x // y)
# another comment
```

উপরের প্রোগ্রামটিতে বেশ কিছু কমেন্ট আমরা লিখেছি যেগুলো পরে বোঝা যাচ্ছে এই প্রোগ্রামে কি করা হয়েছে।

ডক স্ট্রিং
কমেন্টের মত আরও একটি জিনিষ হচ্ছে ডক স্ট্রিং বা ডকুমেন্ট স্ট্রিং। কিন্তু এর ব্যবহার একটু ভিন্ন ভাবে আরও স্পেসিফিক ভাবে করা হয়। সাধারণত মাল্টি লাইন স্ট্রিং কে কোন ফাংশনের মধ্যে শুরুতেই লিখে ওই ফাংশন সম্পর্কিত একটি মন্তব্যের ব্লক লেখা হয় যাকে ডক স্ট্রিং বলা হয়।

```
def greet(word):
    """
    Print a word with an
    exclamation mark following it.
    """
    print(word + "!")

greet("Hello World")
```

সাধারণ কমেন্ট থেকে এটি একটু আলাদা। যেমন - এই স্ট্রিং গুলোকে প্রোগ্রামের রানটাইমের সময়ও অ্যাক্সেস করা যায় নিচের মত করে,

```python
def greet(word):
    """
    Print a word with an
    exclamation mark following it.
    """
    print(word + "!")

# What the fucntion does?
print(greet.__doc__)

# Make sense, now lets use it
greet("Hello World")
```

আউটপুট,

```
    Print a word with an
    exclamation mark following it.

Hello World!
```

উপরের প্রোগ্রামের `print(greet.__doc__)` স্টেটমেন্টটির মাধ্যমে আমরা `greet` ফাংশনের ডকুমেন্টেশন দেখে নিয়েছি এবং তারপর একে কল করে ব্যবহার করেছি।

স্টাইল গাইড

গুগল স্টাইল গাইড মোতাবেক খুব সুন্দর ভাবে একটি ফাংশনের ডক স্ট্রিং লেখা যেতে পারে নিচের মত করে,

```python
def square_root(n):
    """Calculate the square root of a number.

    Args:
        n: the number to get the square root of.
    Returns:
        the square root of n.
    Raises:
        TypeError: if n is not a number.
        ValueError: if n is negative.

    """
    pass
```

অর্থাৎ প্রথমেই ফাংশনের কাজ। তারপর তার প্যারামিটার গুলোর বর্ণনা। এরপরে সেই ফাংশনের কোন রিটার্ন ভ্যালু থাকলে তার বর্ণনা। অতঃপর এটি কোন এরর বা এক্সেপশন রেইজ করবে কিনা এ ব্যাপারটা লেখা।

> ভালো প্রোগ্রামার অবশ্যই ভালো কমেন্ট এবং ডক স্ট্রিং লিখে থাকেন। তাড়াহুড়ায় এড়িয়ে যাওয়া একদম উচিৎ নয়।

অবজেক্ট হিসেবে ফাংশন

একটা কথা বলা হয় = "পাইথনে সব কিছুই অবজেক্ট"। প্রত্যেকটি অবজেক্টেরই কিছু অ্যাট্রিবিউট ও মেথড থাকে। যেমন, আমরা যখনই একটি স্ট্রিং ভ্যারিয়েবল তৈরি করি `a = "Abc"` এভাবে, তখন যদি `type(a)` দেখার চেষ্টা করি তাহলে আউটপুট পাবো `<class 'str'>` অর্থাৎ এই `a` অবজেক্টটি `str` ক্লাসের একটি অবজেক্ট এবং এর কিছু অ্যাট্রিবিউট ও মেথড আছে। যেমন - `lower` , `upper` ইত্যাদি, যেগুলো ব্যবহার করে আমরা `a` কে নিয়ে বিভিন্ন কাজ করতে পারি।

> ক্লাস, অবজেক্ট, অ্যাট্রিবিউট, মেথড নিয়ে বিস্তারিত আলোচনা আছে অবজেক্ট ওরিয়েন্টেড সেকশনে

ফাংশনও একটি অবজেক্ট অর্থাৎ এরও কিছু অ্যাট্রিবিউট ও মেথড আছে। যেমন একটি ফাংশনকে ডিফাইন করার সাথে সাথেই তার `__doc__` নামের অ্যাট্রিবিউট তৈরি হয় যার মাধ্যমে একটি ফাংশনের ডক স্ট্রিং অ্যাক্সেস করা যায়। অন্যান্য সাধারণ ভ্যারিয়েবলের ভ্যালুর মত কোন একটি ফাংশনকেও একটি ভ্যারিয়েবলে অ্যাসাইন বা স্টোর করা যায়।

উদাহরণ,

```
def add_explanation(line):
    return line + '!'

update_line = add_explanation

print(update_line("Hello World"))
```

আউটপুট,

```
Hello World!
```

উপরের প্রোগ্রামে প্রথমে `add_explanation` ফাংশনটিকে `update_line` ভ্যারিয়েবলে অ্যাসাইন করা হয়েছে। এরপর, যেখানে `add_explanation` ফাংশনের দরকার পরেছে সেখানে তাকে `update_line` নামে কল করা হয়েছে। এভাবে বস্তুত `add_explanation` -টাই কল হচ্ছে। আরেকটু পরীক্ষা করার জন্য আমরা যদি `print(update_line)` স্টেটমেন্টটি এক্সিকিউট করি তাহলে আউটপুট আসবে `<function add_explanation at 0x10dbf5668>`

তাহলে একটি প্রশ্ন মাথায় আসতে পারে - যেহেতু ফাংশনকে ভ্যারিয়েবলে অ্যাসাইন করা যায় তাহলে কি ভ্যারিয়েবলের মত করে একটা ফাংশনকেও অন্য ফাংশনের আর্গুমেন্ট হিসেবে পাঠানো যাবে? উত্তর হচ্ছে হ্যাঁ। একটা উদাহরণ দেখি,

```python
def beautify(text):
    return text + '!!!'


def make_line(func, words):
    return "Hello " + func(words)


print(make_line(beautify, "world"))
```

আউটপুট,

```
Hello world!!!
```

উপরের প্রোগ্রামটি একটু বিশ্লেষণ করা যাকঃ ধরে নিচ্ছি `beautify` ফাংশনের কাজ হচ্ছে এর কাছে যাই দেয়া হয় তার সাথে তিনটি বিস্ময় চিহ্ন যুক্ত করে রিটার্ন করে। আবার আমাদের একটি ফাংশন আছে `make_line` যা দিয়ে একটি বাক্য তৈরি করা হয়। কিন্তু আমরা চাই এর মধ্যে বাক্য তৈরির সময়ই শেষ শব্দের সাথে কিছু বিস্ময় চিহ্ন জুড়ে দিতে। তো, যেহেতু বিস্ময় চিহ্ন জুড়ে দেয়ার ফাংশন আমাদের বানানোই আছে তাই ওই ফাংশনকে `make_line` এর একটি আর্গুমেন্ট বা চাহিদা হিসেবে উল্লেখ করতে পারি। অর্থাৎ `make_line` কে কল করতে হলে এর আর্গুমেন্ট হিসেবে একটি ফাংশন এবং একটি ডাটা (ধরে নিচ্ছি একটি শব্দ) পাঠাতে হবে। যাতে করে প্রয়োজনে সে ওই `beautify` ফাংশনকে তার বডির মধ্যে থেকে কল করে ব্যবহার করতে পারে।

`make_line` ফাংশনের ডেফিনেশনে এর কাছে আসা ফাংশনকে `func` নামে রিসিভ করা হয়েছে এবং এর বডির মধ্যে সেই নামেই ব্যবহার করা হয়েছে সাধারণভাবে পাস করা ভ্যারিয়েবলের মত আর তার মাধ্যমে বস্তুত `beautify` ফাংশন কল হয়েছে।

মডিউল

মডিউল হচ্ছে কিছু কোডের সমষ্টি যেখানে বেশ কিছু ফাংশন, ভ্যারিয়েবল বা ডাটা থাকে এবং যেগুলোকে অ্যাক্সেস করে প্রয়োজনে আরেকটি পাইথন প্রোগ্রামে ব্যবহার করা যায়। পাইথনের অনেক অনেক বিল্ট-ইন মডিউল আছে যেগুলোতে অনেক অনেক প্রয়োজনীয় ফাংশন যুক্ত করাই আছে। নিজেদের জন্য কোন প্রোগ্রাম লেখার সময় চাইলে সেই মডিউল গুলো থেকে উক্ত ফিচার গুলো ব্যবহার করা যায়।

নতুন একটি প্রোগ্রামে এরকম কোন মডিউল ব্যবহার করতে চাইলে প্রথমেই সেটিকে import করে নিতে হবে। `import MODULE_NAME` এভাবে। এবার এই স্টেটমেন্টের নিচে `MODULE_NAME.VAR` এভাবে উক্ত মডিউলের ফাংশন বা ভ্যারিয়েবলকে অ্যাক্সেস করা যাবে। একটি উদাহরণ দেখি -

```
import random

value = random.randint(1, 100)
print(value)
```

উপরের প্রোগ্রামে `value` নামের ভ্যারিয়েবলে আমরা একটি র‍্যান্ডম নাম্বার স্টোর করতে চেয়েছি। যে র‍্যান্ডম নাম্বারটি হবে ১ থেকে ১০০ এর মধ্যে। কিন্তু আমরা নিজেরা সেই র‍্যান্ডম নাম্বার তৈরির ফাংশন লিখি নাই। বরং আমরা পাইথনের একটি বিল্ট ইন মডিউল `random` কে ইম্পোর্ট করে নিয়েছি এবং এর মধ্যে আগেই ডিফাইন করে রাখা `randint` ফাংশনকে ব্যবহার করে র‍্যান্ডম নাম্বার পাচ্ছি। এ প্রোগ্রামের আউটপুট এক এক বার এক এক রকম আসবে কিন্তু অবশ্যই এমন একটি ভ্যালু প্রিন্ট হবে যার মান ১ থেকে ১০০ এর মধ্যে।

আরও একভাবে মডিউল ইম্পোর্ট এর কাজ করা যায়। যদি আমাদের কোন একটি মডিউলের নির্দিষ্ট কিছু জিনিষ দরকার হয় তাহলে শুধুমাত্র সেগুলোকে ইম্পোর্ট করা যায়। যেমন নিচের উদাহরণটি -

```
from math import pi, sqrt

print(pi)
print(sqrt(25))
```

আউটপুট,

```
3.141592653589793
5.0
```

উপরের উদাহরণে আমরা `math` মডিউল থেকে শুধুমাত্র `pi` কন্সট্যান্টটি এবং `sqrt` ফাংশনটিকে ইম্পোর্ট করেছি। আর তাই, এই দুটোকে আমরা ব্যবহার করতে পারছি আমাদের প্রোগ্রামে। এখন ধরুন `sqrt` নামটা আপনার পছন্দ হচ্ছে না। আপনি চাচ্ছেন square root বের করার ফাংশনের নাম আরেকটু সুন্দর হলে ভালো হয়। সেটাও করতে পারেন নিচের মত করে -

```python
from math import sqrt as square_root

print(square_root(25))
```

আউটপুট,

```
5.0
```

> কোন একটি মডিউলের সব গুলো অবজেক্ট তথা ফাংশন, ভ্যারিয়েবল, কন্সট্যান্টকে ইম্পোর্ট করার জন্য অনেকেই `from MODULE_NAME import *` ব্যবহার করে থাকেন। এটি একদমই উচিৎ নয়। কারণ এতে করে আপনার কোডের মধ্যে ব্যবহৃত কোন ফাংশন বা ভ্যারিয়েবলের নাম মডিউল থেকে পাওয়া নাকি নিজের তৈরি সেটা নিজেরই বুঝতে সমস্যা হতে পারে।

স্ট্যান্ডার্ড লাইব্রেরি

তিন ধরণের মডিউল হতে পারে। কিছু মডিউল যেগুলো পাইথনের সাথে বিল্ট-ইন আছে, কিছু আছে যেগুলো অন্য কোন ডেভেলপের তৈরি করেছে, এবং কিছু হতে পারে আপনার নিজের তৈরি। প্রথম ধরণের মডিউলকে বলা হয় স্ট্যান্ডার্ড লাইব্রেরি। অনেক অনেক এরকম লাইব্রেরীর মধ্যে কিছু হচ্ছে - `string` , `re` , `datetime` , `math` , `random` , `os` , `multiprocessing` , `subprocess` , `socket` , `email` , `json` , `doctest` , `unittest` , `pdb` , `argparse` এবং `sys` যেগুলোর মাধ্যমে খুব সহজেই স্ট্রিং পারসিং, ডাটা সিরিয়ালাইজেশন, টেস্টিং, ডিবাগিং, ডেট টাইম নিয়ে কাজ, কমান্ড লাইন আর্গুমেন্ট রিসিভ, ইমেইল পাঠানো ইত্যাদি অনেক অনেক কাজ করা যায়। সত্যি কথা বলতে - পাইথনের এই বিশাল পরিমাণ স্ট্যান্ডার্ড লাইব্রেরির কালেকশনের জন্যও এটি একটি অন্যতম জনপ্রিয় প্রোগ্রামিং ভাষা।

মজার ব্যাপার হচ্ছে কিছু কিছু মডিউল পাইথনে লেখা আবার কিছু কিছু মডিউল সি প্রোগ্রামিং ভাষায় লেখা। এই লিঙ্কে গেলে পাইথনের স্ট্যান্ডার্ড লাইব্রেরি গুলো সম্পর্কে আরও বিস্তারিত জানা যাবে। আপনাদের সুবিধার্থে লিঙ্ক সহ সেগুলোর লিস্ট নিচেও দেয়া হলঃ

- 1. Introduction
- 2. Built-in Functions
- 3. Built-in Constants
  - 3.1. Constants added by the `site` module
- 4. Built-in Types
  - 4.1. Truth Value Testing
  - 4.2. Boolean Operations — `and` , `or` , `not`
  - 4.3. Comparisons
  - 4.4. Numeric Types — `int` , `float` , `complex`
  - 4.5. Iterator Types
  - 4.6. Sequence Types — `list` , `tuple` , `range`
  - 4.7. Text Sequence Type — `str`
  - 4.8. Binary Sequence Types — `bytes` , `bytearray` , `memoryview`
  - 4.9. Set Types — `set` , `frozenset`
  - 4.10. Mapping Types — `dict`
  - 4.11. Context Manager Types
  - 4.12. Other Built-in Types
  - 4.13. Special Attributes
- 5. Built-in Exceptions
  - 5.1. Base classes
  - 5.2. Concrete exceptions
  - 5.3. Warnings
  - 5.4. Exception hierarchy
- 6. Text Processing Services

- 6.1. `string` — Common string operations
- 6.2. `re` — Regular expression operations
- 6.3. `difflib` — Helpers for computing deltas
- 6.4. `textwrap` — Text wrapping and filling
- 6.5. `unicodedata` — Unicode Database
- 6.6. `stringprep` — Internet String Preparation
- 6.7. `readline` — GNU readline interface
- 6.8. `rlcompleter` — Completion function for GNU readline

- 7. Binary Data Services
  - 7.1. `struct` — Interpret bytes as packed binary data
  - 7.2. `codecs` — Codec registry and base classes
- 8. Data Types
  - 8.1. `datetime` — Basic date and time types
  - 8.2. `calendar` — General calendar-related functions
  - 8.3. `collections` — Container datatypes
  - 8.4. `collections.abc` — Abstract Base Classes for Containers
  - 8.5. `heapq` — Heap queue algorithm
  - 8.6. `bisect` — Array bisection algorithm
  - 8.7. `array` — Efficient arrays of numeric values
  - 8.8. `weakref` — Weak references
  - 8.9. `types` — Dynamic type creation and names for built-in types
  - 8.10. `copy` — Shallow and deep copy operations
  - 8.11. `pprint` — Data pretty printer
  - 8.12. `reprlib` — Alternate `repr()` implementation
  - 8.13. `enum` — Support for enumerations
- 9. Numeric and Mathematical Modules
  - 9.1. `numbers` — Numeric abstract base classes
  - 9.2. `math` — Mathematical functions
  - 9.3. `cmath` — Mathematical functions for complex numbers
  - 9.4. `decimal` — Decimal fixed point and floating point arithmetic
  - 9.5. `fractions` — Rational numbers
  - 9.6. `random` — Generate pseudo-random numbers
  - 9.7. `statistics` — Mathematical statistics functions
- 10. Functional Programming Modules
  - 10.1. `itertools` — Functions creating iterators for efficient looping
  - 10.2. `functools` — Higher-order functions and operations on callable objects
  - 10.3. `operator` — Standard operators as functions
- 11. File and Directory Access
  - 11.1. `pathlib` — Object-oriented filesystem paths
  - 11.2. `os.path` — Common pathname manipulations

- 11.3. `fileinput` — Iterate over lines from multiple input streams
- 11.4. `stat` — Interpreting `stat()` results
- 11.5. `filecmp` — File and Directory Comparisons
- 11.6. `tempfile` — Generate temporary files and directories
- 11.7. `glob` — Unix style pathname pattern expansion
- 11.8. `fnmatch` — Unix filename pattern matching
- 11.9. `linecache` — Random access to text lines
- 11.10. `shutil` — High-level file operations
- 11.11. `macpath` — Mac OS 9 path manipulation functions

- 12. Data Persistence
  - 12.1. `pickle` — Python object serialization
  - 12.2. `copyreg` — Register `pickle` support functions
  - 12.3. `shelve` — Python object persistence
  - 12.4. `marshal` — Internal Python object serialization
  - 12.5. `dbm` — Interfaces to Unix “databases”
  - 12.6. `sqlite3` — DB-API 2.0 interface for SQLite databases
- 13. Data Compression and Archiving
  - 13.1. `zlib` — Compression compatible with **gzip**
  - 13.2. `gzip` — Support for **gzip** files
  - 13.3. `bz2` — Support for **bzip2** compression
  - 13.4. `lzma` — Compression using the LZMA algorithm
  - 13.5. `zipfile` — Work with ZIP archives
  - 13.6. `tarfile` — Read and write tar archive files
- 14. File Formats
  - 14.1. `csv` — CSV File Reading and Writing
  - 14.2. `configparser` — Configuration file parser
  - 14.3. `netrc` — netrc file processing
  - 14.4. `xdrlib` — Encode and decode XDR data
  - 14.5. `plistlib` — Generate and parse Mac OS X `.plist` files
- 15. Cryptographic Services
  - 15.1. `hashlib` — Secure hashes and message digests
  - 15.2. `hmac` — Keyed-Hashing for Message Authentication
- 16. Generic Operating System Services
  - 16.1. `os` — Miscellaneous operating system interfaces
  - 16.2. `io` — Core tools for working with streams
  - 16.3. `time` — Time access and conversions
  - 16.4. `argparse` — Parser for command-line options, arguments and sub-commands
  - 16.5. `getopt` — C-style parser for command line options
  - 16.6. `logging` — Logging facility for Python

- 16.7. `logging.config` — Logging configuration
- 16.8. `logging.handlers` — Logging handlers
- 16.9. `getpass` — Portable password input
- 16.10. `curses` — Terminal handling for character-cell displays
- 16.11. `curses.textpad` — Text input widget for curses programs
- 16.12. `curses.ascii` — Utilities for ASCII characters
- 16.13. `curses.panel` — A panel stack extension for curses
- 16.14. `platform` — Access to underlying platform's identifying data
- 16.15. `errno` — Standard errno system symbols
- 16.16. `ctypes` — A foreign function library for Python

- 17. Concurrent Execution
  - 17.1. `threading` — Thread-based parallelism
  - 17.2. `multiprocessing` — Process-based parallelism
  - 17.3. The `concurrent` package
  - 17.4. `concurrent.futures` — Launching parallel tasks
  - 17.5. `subprocess` — Subprocess management
  - 17.6. `sched` — Event scheduler
  - 17.7. `queue` — A synchronized queue class
  - 17.8. `dummy_threading` — Drop-in replacement for the `threading` module
  - 17.9. `_thread` — Low-level threading API
  - 17.10. `_dummy_thread` — Drop-in replacement for the `_thread` module
- 18. Interprocess Communication and Networking
  - 18.1. `socket` — Low-level networking interface
  - 18.2. `ssl` — TLS/SSL wrapper for socket objects
  - 18.3. `select` — Waiting for I/O completion
  - 18.4. `selectors` – High-level I/O multiplexing
  - 18.5. `asyncio` – Asynchronous I/O, event loop, coroutines and tasks
  - 18.6. `asyncore` — Asynchronous socket handler
  - 18.7. `asynchat` — Asynchronous socket command/response handler
  - 18.8. `signal` — Set handlers for asynchronous events
  - 18.9. `mmap` — Memory-mapped file support
- 19. Internet Data Handling
  - 19.1. `email` — An email and MIME handling package
  - 19.2. `json` — JSON encoder and decoder
  - 19.3. `mailcap` — Mailcap file handling
  - 19.4. `mailbox` — Manipulate mailboxes in various formats
  - 19.5. `mimetypes` — Map filenames to MIME types
  - 19.6. `base64` — Base16, Base32, Base64, Base85 Data Encodings
  - 19.7. `binhex` — Encode and decode binhex4 files
  - 19.8. `binascii` — Convert between binary and ASCII

- 19.9. `quopri` — Encode and decode MIME quoted-printable data
- 19.10. `uu` — Encode and decode uuencode files

- 20. Structured Markup Processing Tools
  - 20.1. `html` — HyperText Markup Language support
  - 20.2. `html.parser` — Simple HTML and XHTML parser
  - 20.3. `html.entities` — Definitions of HTML general entities
  - 20.4. XML Processing Modules
  - 20.5. `xml.etree.ElementTree` — The ElementTree XML API
  - 20.6. `xml.dom` — The Document Object Model API
  - 20.7. `xml.dom.minidom` — Minimal DOM implementation
  - 20.8. `xml.dom.pulldom` — Support for building partial DOM trees
  - 20.9. `xml.sax` — Support for SAX2 parsers
  - 20.10. `xml.sax.handler` — Base classes for SAX handlers
  - 20.11. `xml.sax.saxutils` — SAX Utilities
  - 20.12. `xml.sax.xmlreader` — Interface for XML parsers
  - 20.13. `xml.parsers.expat` — Fast XML parsing using Expat
- 21. Internet Protocols and Support
  - 21.1. `webbrowser` — Convenient Web-browser controller
  - 21.2. `cgi` — Common Gateway Interface support
  - 21.3. `cgitb` — Traceback manager for CGI scripts
  - 21.4. `wsgiref` — WSGI Utilities and Reference Implementation
  - 21.5. `urllib` — URL handling modules
  - 21.6. `urllib.request` — Extensible library for opening URLs
  - 21.7. `urllib.response` — Response classes used by urllib
  - 21.8. `urllib.parse` — Parse URLs into components
  - 21.9. `urllib.error` — Exception classes raised by urllib.request
  - 21.10. `urllib.robotparser` — Parser for robots.txt
  - 21.11. `http` — HTTP modules
  - 21.12. `http.client` — HTTP protocol client
  - 21.13. `ftplib` — FTP protocol client
  - 21.14. `poplib` — POP3 protocol client
  - 21.15. `imaplib` — IMAP4 protocol client
  - 21.16. `nntplib` — NNTP protocol client
  - 21.17. `smtplib` — SMTP protocol client
  - 21.18. `smtpd` — SMTP Server
  - 21.19. `telnetlib` — Telnet client
  - 21.20. `uuid` — UUID objects according to RFC 4122
  - 21.21. `socketserver` — A framework for network servers
  - 21.22. `http.server` — HTTP servers
  - 21.23. `http.cookies` — HTTP state management

    - 21.24. `http.cookiejar` — Cookie handling for HTTP clients
    - 21.25. `xmlrpc` — XMLRPC server and client modules
    - 21.26. `xmlrpc.client` — XML-RPC client access
    - 21.27. `xmlrpc.server` — Basic XML-RPC servers
    - 21.28. `ipaddress` — IPv4/IPv6 manipulation library
- 22. Multimedia Services
    - 22.1. `audioop` — Manipulate raw audio data
    - 22.2. `aifc` — Read and write AIFF and AIFC files
    - 22.3. `sunau` — Read and write Sun AU files
    - 22.4. `wave` — Read and write WAV files
    - 22.5. `chunk` — Read IFF chunked data
    - 22.6. `colorsys` — Conversions between color systems
    - 22.7. `imghdr` — Determine the type of an image
    - 22.8. `sndhdr` — Determine type of sound file
    - 22.9. `ossaudiodev` — Access to OSS-compatible audio devices
- 23. Internationalization
    - 23.1. `gettext` — Multilingual internationalization services
    - 23.2. `locale` — Internationalization services
- 24. Program Frameworks
    - 24.1. `turtle` — Turtle graphics
    - 24.2. `cmd` — Support for line-oriented command interpreters
    - 24.3. `shlex` — Simple lexical analysis
- 25. Graphical User Interfaces with Tk
    - 25.1. `tkinter` — Python interface to Tcl/Tk
    - 25.2. `tkinter.ttk` — Tk themed widgets
    - 25.3. `tkinter.tix` — Extension widgets for Tk
    - 25.4. `tkinter.scrolledtext` — Scrolled Text Widget
    - 25.5. IDLE
    - 25.6. Other Graphical User Interface Packages
- 26. Development Tools
    - 26.1. `typing` — Support for type hints
    - 26.2. `pydoc` — Documentation generator and online help system
    - 26.3. `doctest` — Test interactive Python examples
    - 26.4. `unittest` — Unit testing framework
    - 26.5. `unittest.mock` — mock object library
    - 26.6. `unittest.mock` — getting started
    - 26.7. 2to3 - Automated Python 2 to 3 code translation
    - 26.8. `test` — Regression tests package for Python
    - 26.9. `test.support` — Utilities for the Python test suite
- 27. Debugging and Profiling

- 27.1. `bdb` — Debugger framework
- 27.2. `faulthandler` — Dump the Python traceback
- 27.3. `pdb` — The Python Debugger
- 27.4. The Python Profilers
- 27.5. `timeit` — Measure execution time of small code snippets
- 27.6. `trace` — Trace or track Python statement execution
- 27.7. `tracemalloc` — Trace memory allocations

- 28. Software Packaging and Distribution
  - 28.1. `distutils` — Building and installing Python modules
  - 28.2. `ensurepip` — Bootstrapping the `pip` installer
  - 28.3. `venv` — Creation of virtual environments
  - 28.4. `zipapp` — Manage executable python zip archives
- 29. Python Runtime Services
  - 29.1. `sys` — System-specific parameters and functions
  - 29.2. `sysconfig` — Provide access to Python's configuration information
  - 29.3. `builtins` — Built-in objects
  - 29.4. `__main__` — Top-level script environment
  - 29.5. `warnings` — Warning control
  - 29.6. `contextlib` — Utilities for `with` -statement contexts
  - 29.7. `abc` — Abstract Base Classes
  - 29.8. `atexit` — Exit handlers
  - 29.9. `traceback` — Print or retrieve a stack traceback
  - 29.10. `__future__` — Future statement definitions
  - 29.11. `gc` — Garbage Collector interface
  - 29.12. `inspect` — Inspect live objects
  - 29.13. `site` — Site-specific configuration hook
  - 29.14. `fpectl` — Floating point exception control
- 30. Custom Python Interpreters
  - 30.1. `code` — Interpreter base classes
  - 30.2. `codeop` — Compile Python code
- 31. Importing Modules
  - 31.1. `zipimport` — Import modules from Zip archives
  - 31.2. `pkgutil` — Package extension utility
  - 31.3. `modulefinder` — Find modules used by a script
  - 31.4. `runpy` — Locating and executing Python modules
  - 31.5. `importlib` – The implementation of `import`
- 32. Python Language Services
  - 32.1. `parser` — Access Python parse trees
  - 32.2. `ast` — Abstract Syntax Trees
  - 32.3. `symtable` — Access to the compiler's symbol tables

- 32.4. `symbol` — Constants used with Python parse trees
- 32.5. `token` — Constants used with Python parse trees
- 32.6. `keyword` — Testing for Python keywords
- 32.7. `tokenize` — Tokenizer for Python source
- 32.8. `tabnanny` — Detection of ambiguous indentation
- 32.9. `pyclbr` — Python class browser support
- 32.10. `py_compile` — Compile Python source files
- 32.11. `compileall` — Byte-compile Python libraries
- 32.12. `dis` — Disassembler for Python bytecode
- 32.13. `pickletools` — Tools for pickle developers

- 33. Miscellaneous Services
  - 33.1. `formatter` — Generic output formatting
- 34. MS Windows Specific Services
  - 34.1. `msilib` — Read and write Microsoft Installer files
  - 34.2. `msvcrt` – Useful routines from the MS VC++ runtime
  - 34.3. `winreg` – Windows registry access
  - 34.4. `winsound` — Sound-playing interface for Windows
- 35. Unix Specific Services
  - 35.1. `posix` — The most common POSIX system calls
  - 35.2. `pwd` — The password database
  - 35.3. `spwd` — The shadow password database
  - 35.4. `grp` — The group database
  - 35.5. `crypt` — Function to check Unix passwords
  - 35.6. `termios` — POSIX style tty control
  - 35.7. `tty` — Terminal control functions
  - 35.8. `pty` — Pseudo-terminal utilities
  - 35.9. `fcntl` — The `fcntl` and `ioctl` system calls
  - 35.10. `pipes` — Interface to shell pipelines
  - 35.11. `resource` — Resource usage information
  - 35.12. `nis` — Interface to Sun’s NIS (Yellow Pages)
  - 35.13. `syslog` — Unix syslog library routines
- 36. Superseded Modules
  - 36.1. `optparse` — Parser for command line options
  - 36.2. `imp` — Access the `import` internals
- 37. Undocumented Modules
  - 37.1. Platform specific modules

pip

আগের চ্যাপ্টারেই আমরা বলেছি পাইথনের বিল্ট ইন মডিউলের সাথে সাথে অন্যদের ডেভেলপ করা অনেক মডিউলও আছে যেগুলো পাইথন প্রোগ্রামিং -কে করেছে আরও প্রোডাক্টিভ এবং সহজ। সেরকম অন্যদের ডেভেলপ করা মডিউল গুলোকে পাওয়া যায় PyPI - the Python Package Index এখানে।

এখানে জমা থাকা মডিউল গুলোকে নিজের কম্পিউটারে ইন্সটল করার সবচেয়ে সহজ পদ্ধতি হচ্ছে **pip** নামের একটি টুল বা প্রোগ্রাম ব্যবহার করা। যদি আপনি পাইথনের অফিসিয়াল সাইট থেকে পাইথনের আপডেটেড ভার্সন ডাউনলোড করে ইন্সটল করে থাকেন তাহলে এই টুলটিও সাথে ইন্সটল হয়ে থাকার কথা।

তো যাই হোক, এই টুল ব্যবহার করে উপরোক্ত পাইথন প্যাকেজ ইনডেক্স সাইট থেকে কোন লাইব্রেরী বা মডিউলকে ইন্সটল করার সহজ পদ্ধতি হচ্ছে - প্রথমে টার্মিনাল ওপেন করতে হবে (উইন্ডোজ হলে কমান্ড প্রম্পট) এবং নিচের কমান্ডটি ইস্যু করতে হবে,

```
pip install LIBRARY_NAME
```

লাইব্রেরীর নাম দেখে নিতে হবে ওই সাইট থেকেই। যেমন মেশিন লার্নিং এবং ডাটা মাইনিং এর জন্য বহুল ব্যবহৃত মডিউল সেট `scikit-learn` কে ইন্সটল করা যাবে নিচের মত করে,

```
pip install -U scikit-learn
```

এভাবে ইন্সটল করার পর ওই লাইব্রেরী বা মডিউলকে নিজের প্রোগ্রামে `import` করে নিতে হবে।

এই সেকশনে থাকছে

- এক্সেপশন
- এক্সেপশন হ্যান্ডেলিং
- finally
- এক্সেপশন Raise
- Assertions
- ফাইল খোলা
- ফাইল পড়া
- ফাইলে লেখা
- ফাইল নিয়ে সঠিক কাজ

এক্সেপশন

এটি এমন একটি ইভেন্ট যা ঘটে তখনই, যখন একটি প্রোগ্রামের স্বাভাবিক এক্সিকিউশনের মধ্যে কোন বাধার উৎপত্তি হয়। অর্থাৎ যখন একটি পাইথন স্ক্রিপ্ট এমন কোন একটি সমস্যাপূর্ণ অবস্থার সম্মুখীন হয় যা সে এড়িয়ে যেতে পারে না অথবা সমাধান করতে পারে না অতঃপর প্রোগ্রামের এক্সিকিউশন বন্ধ হয়ে যায় - সেরকম ঘটনাকে এক্সেপশন বলা হয়। এক্সেপশন এর আভিধানিক অর্থ থেকেও বোঝা যায় যে ব্যতিক্রম কোন অবস্থার উৎপত্তি।

সাধারণত ভুল কোড বা ইনপুটের জন্য প্রোগ্রামের মধ্যে এক্সেপশন তৈরি হয় যা সঠিকভাবে হ্যান্ডেল না করলে প্রোগ্রাম অনাকাঙ্ক্ষিত ভাবে বন্ধ হয়ে যেতে পারে। একটি উদাহরণ দিয়ে আমারা বোঝার চেষ্টা করি -

```
a = 2500
b = 0

print(a/b)
print("I did it")
```

আউটপুট,

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

উপরের প্রোগ্রামে, গণিতের নিয়ম অনুযায়ী `a` কে `b` দিয়ে ভাগ করা সম্ভব না আর তাই যখনই `print(a/b)` স্টেটমেন্টটি এক্সিকিউট হতে চেয়েছে তখনি এক ধরণের ব্যতিক্রম অবস্থার উৎপত্তি হয়েছে যাকে এক্সেপশন বলা হচ্ছে। আর তাই পাইথন ওই প্রোগ্রামের পরবর্তী স্টেটমেন্ট গুলো এক্সিকিউট না করে বরং প্রোগ্রাম এক্সিকিউশন বন্ধ করে দিয়েছে। কিন্তু দেখা যাচ্ছে কোডের লেখায় বা নিয়মে কিন্তু কোন ভুল নাই। শুধুমাত্র রান টাইমেই এই পরিস্থিতি তৈরি হয়েছে। তাই এরকম অবস্থায় পাইথন এক্সেপশন তৈরি করে।

প্রোগ্রামের মধ্যে বিভিন্ন কারনে বিভিন্ন রকম exception তৈরি হয়। কিছু কিছু নির্দিষ্ট কারণের জন্য ঘটা অনাকাঙ্ক্ষিত অবস্থা গুলোর সাপেক্ষে পাইথনে অনেক এক্সেপশন আছে। নিচে কয়েকটি উল্লেখ করা হলঃ

এক্সেপশনের নাম	বর্ণনা
Exception	সব রকম এক্সেপশনের বেজ ক্লাস
StopIteration	যখন একটি ইটারেটরের next() মেথডটি কোন অবজেক্টকে পয়েন্ট করে না
ArithmeticError	নিউমেরিক ক্যালকুলেশনের জন্য তৈরি হয় এমন এক্সেপশনের বেজ ক্লাস
OverflowError	যখন একটি নিউমেরিক টাইপের ম্যাক্সিমাম লিমিট অতিক্রম করে
ZeroDivisonError	যখন শূন্য দিয়ে ভাগের ঘটনা ঘটে
ImportError	যখন import স্টেটমেন্ট ফেইল করে অর্থাৎ কোন কারনে import সম্পন্ন হয় না
IndexError KeyError	যখন একটি সিকোয়েন্স টাইপ অবজেক্টে চাহিদা মোতাবেক ইনডেক্স পাওয়া যায় না
NameError	যখন নির্দিষ্ট নামের কোন আইডেন্টিফায়ারকে লোকাল বা গ্লোবাল স্কোপে খুঁজে পাওয়া যায় না
IOError	যখন ইনপুট বা আউটপুট সম্পর্কিত কোন অপারেশন সফল হয় না যেমন ফাইল থেকে পড়ার জন্য ওপেন ফাংশন কাজ করতে না পারলে
SyntaxError IndentationError	পাইথন প্রোগ্রাম লেখার সময় ভুল কোন কি-ওয়ার্ড বা স্টেটমেন্ট থাকলে
RuntimeError	যখন কোন একটি এক্সেপশন ঘটে যা পাইথনের নির্দিষ্ট কোন ক্যাটাগরির এক্সেপশনের মধ্যেই পরে না

এক্সেপশন হ্যান্ডেলিং

আগের চ্যাপ্টারে আমরা দেখেছি, এক্সেপশন তৈরি হলে প্রোগ্রাম অনাকাঙ্ক্ষিত ভাবে বন্ধ হয়ে যায়। খুশির খবর হচ্ছে এরকম তৈরি হওয়া এক্সেপশন গুলোকে সঠিকভাবে হ্যান্ডেল করতে পারলে প্রোগ্রাম যেমন বন্ধ না হয়ে এগিয়ে চলবে তেমনি প্রোগ্রামের কোথায় কোন সমস্যা আছে সেগুলোকেও সহজে চিহ্নিত করা যাবে। এ জন্য পাইথনে আছে `try` , `except` স্টেটমেন্টের ব্যবহার।

`try` ব্লকের মধ্যে এমন কোড গুলো লেখা হয় যেখানে এক্সেপশন তৈরি হতে পারে (ইউজার ইনপুট বা সেরকম অন্যান্য কারনে)। আর `except` ব্লকের মধ্যে লেখা হয় এমন কোড যেগুলো এক্সিকিউট হবে যদি আসলেই ওই `try` ব্লকের মধ্যে কোন এক্সেপশন তৈরি হয়। অর্থাৎ `try` এর মধ্যে এক্সেপশন তৈরি হলে এই ব্লকের কোড এক্সিকিউশন বন্ধ হবে কিন্তু `except` ব্লকের কোড স্বাভাবিক ভাবে এক্সিকিউট হবে। একটি উদাহরণ দেখি -

```
try:
    a = 1000
    b = int(input("Enter a divisor to divide 1000: "))
    print(a/b)
except ZeroDivisionError:
    print("You entered 0 which is not permitted!")
```

যদি ইনপুট হয় নিচের মত,

```
Enter a divisor to divide 1000: 5
```

তাহলে আউটপুট,

```
200.0
```

অথবা যদি ইনপুট হয় এরকম,

```
Enter a divisor to divide 1000: 0
```

তবে আউটপুট,

```
You entered 0 which is not permitted!
```

উপরের প্রোগ্রামে দুটো নাম্বার নিয়ে ভাগের কাজ করা হয়েছে। একটি নাম্বারের মান 1000 এবং আরেকটি নিচ্ছি ইউজারের কাছ থেকে। যদি ইউজার ভালোয় ভালোয় সঠিক সংখ্যা ইনপুট দেয় (যেমন 5) তাহলে প্রোগ্রামটি সঠিক ভাবে কাজ করে ভাগফল প্রিন্ট করছে। কিন্তু ইউজারের মনোভাব তো আমরা জানি না। ইউজার চাইলে শূন্য ইনপুট দিতে পারে। আর তখন প্রোগ্রাম ভাগ করতে না পেরে অনাকাঙ্ক্ষিত ভাবে বন্ধ হয়ে যাবে।

আর তাই সেটুকু আন্দাজ করেই আমরা ভাগ করার কোড টুকু একটি ট্রাই ব্লকের মধ্যে লিখেছি এবং সেই ব্লকের মধ্যে যদি শূন্য দিয়ে ভাগ করার কারনে কোন এক্সেপশন তৈরি হয় তাহলে সেটা হ্যান্ডেল করার জন্য এক্সেপ্ট ব্লক ব্যবহার করেছি এবং নির্দিষ্ট করে `ZeroDivisionError` এক্সেপশন হ্যান্ডেল করেছি। এখন, ইউজার চাইলে শূন্য ইনপুট দিতে পারে, তাই বলে প্রোগ্রাম অনাকাঙ্ক্ষিত ভাবে শাটডাউন বা বন্ধ হবে না। বরং ইউজারকে যথাযথ ম্যাসেজ দেখিয়ে স্বাভাবিক কাজ চালিয়ে যেতে পারছে।

একটি `try` ব্লকের সাপেক্ষে একাধিক `except` ব্লক থাকতে পারে। আবার একটি `except` এর জন্য একাধিক এক্সেপশন ডিফাইন করা যেতে পারে ব্র্যাকেট এবং কমা ব্যবহার করে। এতে করে ট্রাই ব্লকের মধ্যে বিভিন্ন রকম এক্সেপশনের জন্য বিভিন্ন এক্সেপ্ট ব্লক দিয়ে সঠিক ভাবে সমস্যাকে চিহ্নিত করা যায় এবং সে অনুযায়ী কাজ করা যায়। আরেকটি উদাহরণ দেখি -

```
try:
    variable = 10
    print(variable + "hello")
    print(variable / 2)
except ZeroDivisionError:
    print("Divided by zero")
except (ValueError, TypeError):
    print("Type or value error occurred")
```

আউটপুট,

```
Type or value error occurred
```

উপরের প্রোগ্রামে ট্রাই ব্লকে দুই রকম অঘটন ঘটতে পারে। `variable` কে `2` দিয়ে ভাগ না করে শূন্য দিয়ে ভাগ করা হতে পারতো এবং সেক্ষেত্রে `ZeroDivisionError` এক্সেপশন তৈরি হত। আবার ট্রাই ব্লকের দ্বিতীয় স্টেটমেন্ট যেখানে একটি ইন্টিজারের সাথে স্ট্রিং কে যোগ করে প্রিন্ট করার চেষ্টা করা হয়েছে, সেখানে। এই উদাহরণে এখানেই এক্সেপশন তৈরি হচ্ছে। আর তাই `TypeError` এক্সেপশন তৈরি হচ্ছে। কিন্তু আমরা সেটা সঠিকভাবে হ্যান্ডেল করেছি আর তাই প্রোগ্রাম হুট করে বন্ধ না হয়ে বরং সুন্দর ভাবে আমাদের নির্ধারিত একটি প্রিন্ট স্টেটমেন্ট `print("Type or value error occurred")` এক্সিকিউট করেছে।

চাইলে সুনির্দিষ্ট ভাবে কোন এক্সেপশন ডিফাইন না করেও `except` ব্লক ব্যবহার করা যাবে। সেক্ষেত্রে `try` ব্লকের মধ্যে ঘটে যাওয়া যেকোনো রকম এক্সেপশনের জন্য এই `except` ব্লক রান করবে। যেমন -

```
try:
    word = "spam"
    print(word / 0)
except:
    print("An error occurred")
```

আউটপুট,

```
An error occurred
```

বোঝাই যাচ্ছে `try` ব্লকের মধ্যে উল্টা পাল্টা টাইপের ডাটা নিয়ে ভাগ করার কোড লেখা হয়েছে। রান টাইমে এখানে অবশ্যই এক্সেপশন তৈরি হচ্ছে। আর তাই `except` ব্লক ব্যবহার করে হ্যান্ডেলও করা হয়েছে। আপাত দৃষ্টিতে বিষয়টি ভালো মনে হলেও এভাবে হ্যান্ডেল করা ব্লকের মাধ্যমে ট্রাই ব্লকে ঘটে যাওয়া অঘটনের সঠিক কারণ চিহ্নিত করা যাবে না।
অর্থাৎ, ট্রাই ব্লকে যেকোনো রকম সমস্যার জন্যই এই এক্সেপ্ট ব্লক এক্সিকিউট হবে এবং বার বার শুধু `An error occurred` ম্যাসেজটাই ইউজারকে দেখানো হবে। কিন্তু যদি সম্ভাবনাময় কয়েকটি নির্দিষ্ট টাইপের এক্সেপ্ট ব্লক লিখে সেগুলোর মধ্যে আলাদা আলাদা ম্যাসেজ প্রিন্ট করা হত। তাহলে ইউজারকে আরও সুনির্দিষ্ট ম্যাসেজ দেয়া যেত এবং প্রোগ্রামটিকে পরবর্তীতে আপডেট করতেও সুবিধা হত।

> সংকলন - নুহিল মেহেদী

ফাইনালি

যদি এমন দরকার হয় যে, যতই এক্সেপশন তৈরি হোক না কেন কিছু কোডকে রান করানো দরকার, তখন `finally` স্টেটমেন্ট ব্যবহার করা হয়। `try` , `except` ব্লকের নিচে `finally` ব্লক ব্যবহার করতে হয়। `try` বা `except` ব্লকের কোড রান হবার পর এই `finally` ব্লকের মধ্যে থাকা কোড গুলো রান হবেই। একটি উদাহরণ দেখি -

```
try:
   print("Hello")
   print(1 / 0)
except ZeroDivisionError:
   print("Divided by zero")
finally:
   print("This code will run no matter what")
```

আউটপুট,

```
Hello
Divided by zero
This code will run no matter what
```

উপরের প্রোগ্রামে, `try` ব্লকের মধ্যে প্রথম প্রিন্ট স্টেটমেন্টের পর দ্বিতীয় প্রিন্ট স্টেটমেন্টে শূন্য দিয়ে ভাগের চেষ্টার কারনে `ZeroDivisionError` এক্সেপশন তৈরি হচ্ছে। সেটাকে সঠিকভাবে হ্যান্ডেল করায় `except` ব্লকের মধ্যে থাকা `print("Divided by zero")` এক্সিকিউট করছে। এবং পরিশেষে, যেহেতু ঘটনা যাই হোক `finally` ব্লক এর কোড এক্সিকিউট হবেই, তাই `print("This code will run no matter what")` স্টেটমেন্টটিও কাজ করছে।

যদি `finally` ব্লকের আগে এমন কোন এক্সেপশন তৈরি হয় যাকে সঠিক ভাবে হ্যান্ডেল করা হয় নাই, সে অবস্থাতেও `finally` ব্লকের কোড রান হবে। যেমন -

```
try:
   print(1)
   print(10 / 0)
except ZeroDivisionError:
   print(unknown_var)
finally:
   print("This is executed last")
```

আউটপুট,

```
1
This is executed last
Traceback (most recent call last):
  File "/Users/nuhil/Documents/Python/Test.py", line 3, in <module>
    print(10 / 0)
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/Users/nuhil/Documents/Python/Test.py", line 5, in <module>
    print(unknown_var)
NameError: name 'unknown_var' is not defined
```

উপরের প্রোগ্রামের `try` ব্লকের মধ্যে একটি এক্সেপশন তৈরি হয় এবং সেটা `except` ব্লকে হ্যান্ডেল করা হয়। কিন্তু সেই হ্যান্ডেল করার ব্লকের মধ্যে আবার এমন একটা ভ্যারিয়েবল প্রিন্ট করতে চাওয়া হয়েছে যাকে ডিফাইন করাই হয় নাই। আর তাতে করে সেখানে একটা `NameError` টাইপের এক্সেপশন তৈরি হয় (যদিও এটাকে হ্যান্ডেল করা হয় নি)। তারপরেও `finally` ব্লক কাজ করছে আর তাই `This is executed last` কে আউটপুট স্ক্রিনে দেখা যাচ্ছে।

এক্সেপশন রেইজ (তৈরি) করা

আগে আমরা দেখেছি কিভাবে পাইথন প্রয়োজনে নিজে থেকেই প্রোগ্রামে কিছু এক্সেপশন তৈরি করে। চাইলে ম্যানুয়ালি কোড লিখেও প্রোগ্রামের নির্দিষ্ট কোন যায়গায় এক্সেপশন রেইজ বা সহজ ভাবে বলতে গেলে এক্সেপশন তৈরি করা যায়। `raise` স্টেটমেন্ট ব্যবহার করে এভাবে কাস্টম এক্সেপশন তৈরি করা যায়। নিচের উদাহরণটি দেখি -

```
print("Hello")
raise NameError('HiThere')
```

আউটপুট,

```
Hello
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

উপরের প্রোগ্রামের দ্বিতীয় লাইনে আমরা ম্যানুয়ালি একটি `NameError` টাইপের এক্সেপশন তৈরি করেছি যার কারনে পাইথন সাধারণভাবেই সেই এক্সেপশনটি থ্রো করেছে।

উপরের মত এক্সেপশনের আর্গুমেন্ট (HiThere) সেট না করেও শুধু `NameError` এক্সপশন থ্রো করে যেত। যেমন নিচের মত -

```
raise TypeError
```

আউটপুট,

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError
```

`raise` এর একটি মজার ব্যবহার দেখবো নিচের উদাহরণে,

```
try:
    num = 5 / 0
except:
    print("Custom message about an error!")
    raise
```

আউটপুট,

```
Custom message about an error!
Traceback (most recent call last):
  File "/Users/nuhil/Desktop/Test.py", line 2, in <module>
    num = 5 / 0
ZeroDivisionError: integer division or modulo by zero
```

খেয়াল করুন কি ঘটছে উপরের প্রোগ্রামে। খুব সহজেই বোঝা যাচ্ছে যে `try` ব্লকে একটি এক্সেপশন ঘটছে। এটাও বুঝতে পারছি যে সেটা `ZeroDivisionError` এক্সেপশন হতে পারে কারন শূন্য দিয়ে ৫ কে ভাগ করার চেষ্টা করা হয়েছে। কিন্তু আমরা `except` ব্লকে নির্দিষ্ট করে কোন এক্সেপশন ডিফাইন করে সেটা হ্যান্ডেল করছি না। তারপরেও শেষ নাগাদ পাইথন আমাদেরকে `ZeroDivisionError: integer division or modulo by zero` এক্সেপশন দেখাতে পারছে। এর কারন - আমরা `except` এর মধ্যে `raise` ব্যবহার করেছি। এভাবেও `raise` কে কাজে লাগিয়ে এর আগে ঘটে যাওয়া এক্সেপশনের টাইপ পেয়ে যেতে পারি।

Assertions

পাইথনে assertion তথা স্যানিটি চেক এনাবেল বা ডিজ্যাবল করে প্রোগ্রাম টেস্টিং এর কাজ করা হয়। কিন্তু, স্যানিটি চেক (sanity-check) আসলে কি? খুব দ্রুত একটি স্টেটমেন্টকে পর্যবেক্ষণ করে সেটার ফলাফলের সত্যতা যাচাই করাকেই স্যানিটি চেক বলা হয়।

`assert` স্টেটমেন্ট ব্যবহার করে এই কুইক টেস্ট করা হয়। যখন পাইথন কোন প্রোগ্রামের যেকোনো যায়গায় এই `assert` স্টেটমেন্টটি পায় তখন সেটাকে দ্রুত যাচাই করে এবং স্টেটমেন্টটি সত্য হোক সেটা আশা করে। কিন্তু তা না হলে পাইথন `AssertionError` টাইপের এক্সেপশন থ্রো (তৈরি) করে। একটি উদাহরণ দেখি -

```
print(1)
assert 2 + 2 == 4
print(2)
assert 1 + 1 == 3
print(3)
```

আউটপুট,

```
1
2
Traceback (most recent call last):
  File "/Users/nuhil/Desktop/Test.py", line 4, in <module>
    assert 1 + 1 == 3
AssertionError
```

উপরের প্রোগ্রামের প্রথম প্রিন্ট স্টেটমেন্টের পর একটি assertion সেট করা হয়েছে। সেখানে একটি সাধারণ অ্যারিদম্যাটিক কন্ডিশন যাচাই করা হয়েছে `assert` ব্যবহার করে। সেই স্যানিটি চেকটি সত্য বা পাশ হয়েছে (২ আর ২ যোগ করলে ৪ হয়)। তাই, `print(2)` স্টেটমেন্ট কাজ করছে। এরপর আবার একটি স্যানিটি চেক সেট করা হয়েছে। কিন্তু, স্বাভাবিক ভাবেই সেটি সত্য নয় (১ আর ১ যোগ করে ৩ হয় না)। তাই পাইথন সেখানে একটি `AssertionError` এক্সেপশন থ্রো করেছে। আর তাই, এর পরে থাকা `print(3)` স্টেটমেন্টটি এক্সিকিউটও হয় নি।

সাধারণত প্রোগ্রামারগণ কোন একটি ফাংশনের ডেফিনেশনের শুরুতেই এরকম স্যানিটি চেক ব্যবহার করেন ইনপুট/আর্গুমেন্ট ডাটা চেক করার জন্য। আবার ফাংশন কল এর পরেও ব্যবহার করে থাকেন ফাংশনের আউটপুট ডাটা চেক করার জন্য।

আরেকটি উদাহরণ,

```
def KelvinToFahrenheit(Temperature):
   assert (Temperature >= 0),"Colder than absolute zero!"
   return ((Temperature-273)*1.8)+32

print KelvinToFahrenheit(273)
print int(KelvinToFahrenheit(505.78))
print KelvinToFahrenheit(-5)
```

আউটপুট,

```
32.0
451
Traceback (most recent call last):
  File "/Users/nuhil/Desktop/Test.py", line 7, in <module>
    print KelvinToFahrenheit(-5)
  File "/Users/nuhil/Desktop/Test.py", line 2, in KelvinToFahrenheit
    assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

> বলা বাহুল্য, অন্যান্য এক্সেপশনের মত এই এক্সেপশনকেও `try` , `except` দিয়ে হ্যান্ডেল করা যায়।

ফাইল খোলা

প্রোগ্রামিং মানেই ফাইল থেকে ডাটা নিয়ে কাজ করা খুবই স্বাভাবিক একটি ঘটনা। আর পাইথনে ফাইল নিয়ে কাজ করা অনেক সহজ। এর মাধ্যমে কোন ফাইল থেকে ডাটা পড়া, ফাইলে নতুন ডাটা লেখা বা ফাইল কন্টেন্টকে আপডেট করা ইত্যাদি করা যায় খুব সহজে অল্প কোড লিখেই।

ফাইল নিয়ে কাজ করার শুরুতেই পাইথনের বিল্ট ইন ফাংশন `open` ব্যবহার করে সেই ফাইলকে ওপেন করে নিতে হবে। ওপেন মানে কোন এডিটরে ওপেন নয়, বরং পাইথনের কাছে সেটা ওপেন হয়ে থাকে কাজ করার উপযোগী মোডে। নিচের মত একটি লাইন লিখেই সেটি করা যায় -

```
file_to_work_on = open("file_name.txt")
```

`open` ফাংশনের আর্গুমেন্ট হিসেবে আলোচ্য ফাইলের পাথ দিতে হয়। যদি পাইথন স্ক্রিপ্ট এবং ফাইলটি কম্পিউটারের একই লোকেশনে থাকে তাহলে উপরের মত শুধু ফাইলের নামটি লিখলেই কাজ শেষ। না হলে `file_to_work_on = open("/Users/nuhil/Desktop/file_name.txt")` এভাবে লিখতে হতে পারে।

`open` ফাংশনের আরও কিছু আর্গুমেন্ট আছে যেমন - দ্বিতীয় আর্গুমেন্ট পাস করে নির্ধারণ করা হয়, পাইথন উক্ত ফাইলটিকে কোন মোডে খুলবে অর্থাৎ শুধু সেটি থেকে ডাটা পড়ার জন্য নাকি, সেখানে ডাটা লেখার জন্য নাকি নতুন ডাটা যুক্ত করার জন্য।

যেমন লেখার জন্য তথা রাইট মোডে খোলার জন্য -

```
file_to_work_on = open("file_name.txt", "w")
```

রিড মোডে খোলার জন্য -

```
file_to_work_on = open("file_name.txt", "r")
```

অ্যাপেন্ড তথা ফাইলের শেষে নতুন কন্টেন্ট যুক্ত করার জন্য সেই মোডে খুলতে -

```
file_to_work_on = open("file_name.txt", "a")
```

টেক্সট ফাইল নয় এমন ফাইল নিয়ে কাজ করার জন্য বাইনারি মোডে সেই ফাইলকে খুলতে হবে। যেমন একটি বাইনারি ফাইলকে রাইট মোডে খোলার জন্য -

```
file_to_work_on = open("my_file", "wb")
```

ফাইল খোলার পর সেটি নিয়ে কাজ শেষে গুরুত্বপূর্ণ আরেকটি টাস্ক হচ্ছে সেই ফাইলকে ক্লোজ বা বন্ধ করা। নাহলে অকারণেই পাইথনের কাছে ফাইলটি ওপেন অবস্থাতেই থাকবে যা বস্তুত মেমোরি দখল করে থাকবে এবং প্রোগ্রামের পারফর্মেন্সে খারাপ ভূমিকা রাখবে। মোট কথা, আমরা যেমন কম্পিউটারে কোন এডিটর দিয়ে একটি ফাইল খুলে সেখানে কাজ শেষে বন্ধ করি অযথা র‍্যাম নষ্ট না করার জন্য। একই কারনে প্রোগ্রাম্যাটিক্যালি কোন একটি ফাইল নিয়ে কাজের শেষেও সেটা বন্ধ করা উচিৎ। সিম্পল ব্যাপার, তাই না?

নিচের মত করে ক্লোজ করার কাজটি করা যায় -

```
file_to_work = open("filename.txt", "w")
# do HERE whatever you like, with the file
# such as write new lines in it

# then close it
file_to_work.close()
```

> সংকলন - নুহিল মেহেদী

ফাইল থেকে পড়া

আগের চ্যাপ্টারে আমরা দেখেছি কিভাবে পাইথনে ফাইল খুলতে হয় এবং বন্ধ করতে হয়। এই চ্যাপ্টারে দেখবো কিভাবে ফাইলে খুলে সেই ফাইল থেকে বিভিন্নভাবে কন্টেন্ট পড়া যায়। একটি ফাইল খুলে সেই ফাইলের সব কন্টেন্ট পড়ে স্ক্রিনে প্রিন্ট করার একটি প্রোগ্রাম দেখি -

```
file_to_work = open("Test.txt", "r")
content = file_to_work.read()

print(content)

file_to_work.close()
```

আউটপুট,

```
Hello World!!!
This is second line in the file.

This is third one.
```

উপরের প্রোগ্রামের Test.txt ফাইলে তিনটি আলাদা আলাদা লাইনে নিচের কন্টেন্ট ছিলঃ

Hello World!!!
This is second line in the file.

This is third one.

প্রথমেই ওপেন ফাংশন ব্যবহার করে এবং ফাইলের পাথ ডিফাইন করে দিয়ে একটি ফাইল অবজেক্ট পেয়েছি `file_to_work` নামের। এরপর এই অবেজক্টের মেথড `read` ব্যবহার করে পুরো ফাইলে থাকা কন্টেন্ট পড়ে `content` ভ্যারিয়েবলে জমা করেছি। অতঃপর, একটি প্রিন্ট স্টেটমেন্ট ব্যবহার করে সেই কন্টেন্ট স্ক্রিনে প্রিন্ট করেছি। আর কাজ শেষে, ফাইল অবজেক্ট এর `close` মেথড ব্যবহার করে ফাইলকে ক্লোজ করেছি।

এভাবে পুরো কন্টেন্ট একসাথে না পড়ে বাইট হিসেবেও পড়া যায়। `read` মেথডের আর্গুমেন্ট হিসেবে কত বাইট পড়তে চাই সেটা পাঠিয়ে দেয়া যায়। উপরের প্রোগ্রামের একটু মডিফায়েড ভার্সন দেখি -

```
file_to_work = open("Test.txt", "r")

just_one_character = file_to_work.read(1)
print(just_one_character)

remaining_four_characters = file_to_work.read(4)
print(remaining_four_characters)

rest_of_the_file = file_to_work.read()
print(rest_of_the_file)

file_to_work.close()
```

আউটপুট,

```
H
ello
 World!!!
This is second line in the file.

This is third one.
```

উপরের প্রোগ্রামে তিন বার ফাইল থেকে কন্টেন্ট পড়া হয়েছে, কিন্তু তিনভাবে। প্রথমবার মাত্র একটি বাইট পড়া হয়েছে। এক বাইট মানে একটি ক্যারেক্টার। তাই সেটি প্রিন্ট করেছে শুধু `H`. এর পরে আবার পড়া হয়েছে ৪টি বাইট। তাই `ello` এই চার ক্যারেক্টার পড়া হয়েছে। যেহেতু আমরা একই ফাইল অবজেক্ট (`file_to_work`) নিয়ে দ্বিতীয় বারও কাজ করেছি তাই এইবার যে ৪বাইট পড়তে চেয়েছি সেটা আসলে `H` এর পর থেকে ৪বাইট। তৃতীয় বার কোন আর্গুমেন্ট ছাড়া `read` মেথড ব্যবহার করা হয়েছে এবং ফাইলের বাকী সব কন্টেন্ট পড়ে প্রিন্ট করা হয়েছে। এবারও যেহেতু একই ফাইল অবজেক্ট এর উপরেই কাজ করা হয়েছে তাই `rest_of_the_file` ভ্যারিয়েবলে কিন্তু `H` , `ello` এর পর থেকে অর্থাৎ `World ...` থেকে শেষ পর্যন্ত সব কন্টেন্ট জমা হয়েছে।

ইতোমধ্যে অনেকের মনে হতে পারে, এভাবে পুরো কন্টেন্ট একবারে পড়া এবং সেগুলো নিয়ে কাজ করা একটু ঝামেলা হবে; তাদের জন্য আছে `readlines` মেথড। এই মেথড ব্যবহার করলে ফাইলের প্রত্যেকটি লাইন আলাদা আলাদা করে নিয়ে পাইথন একটি লিস্ট বানায় এবং লিস্টের এক একটি এলিমেন্ট হয় এক একটি লাইন। নিচের উদাহরণটি দেখি -

```
file_to_work = open("Test.txt", "r")

lines = file_to_work.readlines()
print(lines)

file_to_work.close()
```

আউটপুট,

```
['Hello World!!!\n', 'This is second line in the file.\n', '\n', 'This is third one. \n']
```

অনেকেই হয়তো ভাবছেন লিস্ট যেহেতু পেয়ে গেছি তাহলে এবার লাইন বাই লাইন নিয়ে কাজ করার জন্য ফর লুপ ব্যবহার করে সহজেই কাজ করে ফেলবো। আপনার কথা মাথায় রেখেই পাইথনের ফর লুপ রেডি হয়েই আছে। নিচের উদাহরণটি দেখুন -

```
file_to_work = open("Test.txt", "r")

for my_line in file_to_work:
    print(my_line)

file_to_work.close()
```

আউটপুট,

```
Hello World!!!

This is second line in the file.



This is third one.
```

দেখুন কিভাবে আলাদা করে `read` বা `readlines` মেথড ব্যবহার না করেই সরাসরি ফর লুপ ব্যবহার করে প্রত্যেকটি লাইনকে অ্যাক্সেস করা যায়। আউটপুট স্ক্রিনে একটা করে ফাকা লাইন বেশি প্রিন্ট হয়েছে। এতে প্রমাণিত হয় যে, ফর লুপের মধ্যে থাকা প্রিন্ট স্টেটমেন্ট আলাদা আলাদা ভাবে তিনবার এক্সিকিউট হয়েছে যার কারনে প্রত্যেকবার প্রিন্টের পর একটি করে ফাকা লাইন প্রিন্ট হয়েছে।

> সংকলন - নুহিল মেহেদী

ফাইলে লেখা

পাইথনের মেথডের নাম গুলোও কেন যেন আমার কথা মাথায় রেখেই দেয়া। যেমন ফাইল পড়ার ফাংশনের নাম `read` এবং কেউ না বলে দিলেও ফাইলে লেখার ফাংশনের নাম যে `write` সেটা আপনি এতক্ষণে ধরে নিয়েছে। আর হ্যাঁ, আপনার ধারনা ভুল না। ফাইল থেকে পড়েন আর ফাইলে লিখেন, যাই করেন না কেন ফাইলকে আগে ওপেন করেই নিতে হবে। আবার কাজ শেষে বন্ধ করতে হবে (উচিৎ)।

উদাহরণ,

```
file_to_work = open("Test.txt", "w")
file_to_work.write("I am writing!!!")
file_to_work.close()

file_to_work = open("Test.txt", "r")
print(file_to_work.read())
file_to_work.close()
```

উপরোক্ত প্রোগ্রামের দুটি অংশ। প্রথম অংশে ফাইলকে ওপেন করে সেখানে একটি লাইন লেখা হয়েছে। আমাদের চলতি উদাহরণ মোতাবেক এই নামের ফাইলটি আগে থেকেই ছিল। কিন্তু `w` মোডে খোলার কারনে এবং এখানে নতুন করে লেখার কারনে ওই ফাইলের আগের সব কন্টেন্ট মুছে যাবে এবং নতুন `write` করা কন্টেন্ট লেখা হবে। যদি ওই নামের ফাইল না থাকতো, তাহলে পাইথন নতুন করে ওই নামে একটি ফাইল তৈরি করে সেখানে লিখতো। লেখা শেষে ফাইলটিকে ক্লোজ করা হয়েছে।
দ্বিতীয় অংশে আবার সেই ফাইলকে পড়ার জন্য `r` মোডে খোলা হয়েছে এবং সব কন্টেন্ট পড়ে স্ক্রিনে প্রিন্ট করা হয়েছে।

আউটপুট,

```
I am writing!!!
```

চাইলে ফাইল লেখার কাজ সফল হল কিনা এবং কি পরিমাণ কন্টেন্ট ফাইলে লেখা হল সেটা যাচাই করার জন্য `write` মেথডের রিটার্ন ভ্যালুকে ক্যাপচার করে দেখা যেতে পারে নিচের মত করে -

```
file_to_work = open("Test.txt", "w")
is_writing_done = file_to_work.write("I am writing!!!")

if is_writing_done:
    print("Yes, {0} byte(s) has been written!".format(is_writing_done))
file_to_work.close()
```

আউটপুট,

```
Yes, 15 byte(s) has been written!
```

> সংকলন - নুহিল মেহেদী

ফাইল নিয়ে সঠিক ভাবে কাজ করা

ইতোমধ্যে বেশ কয়েকবার বলা হয়েছে যে, ফাইল নিয়ে কাজ শেষে সেটিকে ক্লোজ করা খুব দরকারি। তো, এই দরকারি কাজটা যাতে বার বার ভুল হয়ে না যায় এর জন্য কিছু টেকনিক অবলম্বন করা যেতে পারে বা অভ্যাসে পরিণত করা যেতে পারে। যেমন, নিচের প্রোগ্রামটি দেখি -

```
try:
    file_to_work = open("Test.txt", "r")
    content = file_to_work.read()
    print(content)
finally:
    file_to_work.close()
```

মনে আছে, আমরা কয়েক চ্যাপ্টার আগেই `finally` ব্লক নিয়ে আলোচনা করেছি? `try` , `except` এর সাথে `finally` ব্লকের ব্যবহার আমরা দেখেছি এবং জানি যে এই ব্লকের মধ্যে যাই থাকুক না কেন, সেই কোড গুলো রান করবেই এমনকি যদি এর উপরের `try` , `except` ব্লকে অনাকাঙ্ক্ষিত কিছু ঘটেও। এটাই একটা টেকনিক, ফাইল ক্লোজ করতে ভুল না করার।
উপরের প্রোগ্রামে আমরা ট্রাই ব্লকের মধ্যে ফাইল ওপেন এবং পড়ার কাজ করেছি এবং ফাইনালি ব্লকের মধ্যে ক্লোজ করেছি। এতে করে, ঘটনা যাই হোক, ফাইল ক্লোজ হবেই।

আরও একটি বেস্ট প্র্যাকটিস আছে। `with` স্টেটমেন্টের ব্যবহার। প্রথমে একটি উদাহরণ দেখি তারপর বিশ্লেষণ করা যাবে -

```
with open("Test.txt") as f:
    print(f.read())
```

আউটপুট,

```
I am writing!!!
```

`with` স্টেটমেন্ট আসলে একটি টেম্পোরারি ভ্যারিয়েবল তৈরি করে। উপরের প্রোগ্রামে এটি ব্যবহার করে `open("Test.txt")` স্টেটমেন্টটির জন্য একটি টেম্পোরারি ভ্যারিয়েবল তৈরি করা হয়েছে `f` নামে। অর্থাৎ বস্তুত এমন হয়েছে `f = open("Test.txt")` . এই `f` কে `with` এর আওতাভুক্ত কোডে অর্থাৎ এর স্কোপে ব্যবহার করা যায়। আবার, `with` ব্যবহারের আরেকটি মজার ব্যাপার হচ্ছে এর আওতাভুক্ত কোড ব্লকের কাজ শেষ হয়ে গেলেই এর দ্বারা তৈরি টেম্পোরারি ভ্যারিয়েবলও ডেস্ট্রয় হয়ে যায়। এতে করে আমাদের উদ্দেশ্য হাসিল হয় তথা ফাইল ক্লোজের কাজটি হয়ে যায়। এখন পর্যন্ত এটাকেই ফাইল নিয়ে ছোট খাটো কাজ করার বেস্ট প্র্যাকটিস হিসেবে ধরা হয়।

> সংকলন - নুহিল মেহেদী

এই সেকশনে থাকছে

- ভূমিকা
- ল্যামডা
- ম্যাপ ও ফিল্টার
- জেনারেটর
- ডেকোরেটর
- রিকারসন
- সেট
- itertools

ফাংশনাল প্রোগ্রামিং কি

সহজ ভাষায় এটা একটা প্রোগ্রামিং স্টাইল যেটা বিশেষত নির্ভর করে ফাংশনের উপর । higher-order-function গুলো এই ধারার মুল জিনিষ । যে ফাংশন আরেকটি ফাংশনকে আর্গুমেন্ট হিসেবে নিতে পারে এবং অথবা রিটার্ন এলিমেন্ট হিসেবে একটি ফাংশন রিটার্ন করতে পারে তাকে higher-order-function বলা হয় । যেমন -

```
>>> def make_twice(func, arg):
...     return func(func(arg))
...
>>> def add_five(x):
...     return x + 5
...
>>> print(make_twice(add_five, 10))
20
>>>
```

উপরের উদাহরণে, প্রথমেই আমরা `make_twice` ফাংশনকে কল করছি আর এর ডেফিনেশন মোতাবেক এর কাছে একটি ফাংশন এবং একটি ভ্যালু পাঠিয়েছি । যে ফাংশন পাঠিয়েছি সেটা হচ্ছে `add_five` এবং ভ্যালুটি `10` . অন্যদিকে `make_twice` ফাংশনের মধ্যে `func` হিসেবে সেই `add_five` ফাংশনকে ক্যাচ করছি এবং একাধিকবার সেটাকে কল করছি । আবার, `func` তথা `add_five` এর একটি আর্গুমেন্ট লাগে । তাই, প্রথমবার এক্সিকিউটের সময় এর মধ্যে 10 কে পাঠিয়ে দিচ্ছি এবং রিটার্ন হয়ে আসছে 15 এবং দ্বিতীয়বার সেই 15 -ই গিয়ে ফাইনালি 20 কে রেজাল্ট হিসেবে পাচ্ছি ।

পিওর বা শুদ্ধ ফাংশন

ফাংশনাল প্রোগ্রামিং-এর সাথে সাথে পিওর এবং ইম্পিওর ফাংশনের বিষয় চলে আসে । পিওর ফাংশন হচ্ছে সেই ফাংশন যার কোন পার্শ্ব-প্রতিক্রিয়া নাই এবং যে ফাংশন কোন কিছু রিটার্ন করে শুধুমাত্র তার আর্গুমেন্ট ভ্যালুর উপর ভিত্তি করেই । যেমন -

```
>>> def my_pure_function(a,b):
...     c = (2 * a) + (2 * b)
...     return c
...
>>> my_pure_function(5,10)
30
```

উপরের প্রোগ্রামে `my_pure_function` দুটো আর্গুমেন্ট নেয় এবং সেগুলোর উপর কিছু ক্যালকুলেশন করে একটি ভ্যালু রিটার্ন করে । শুরু থেকে শেষ নাগাদ এই ফাংশন বাইরের কোন ভ্যালুর উপর নির্ভর করে না বা বাইরের কোন ভ্যালুকে পরিবর্তনের সাথে সম্পৃক্ত নয় । এটাকে একটা পিওর ফাংশন বলা যেতে পারে ।

ইম্পিওর বা অশুদ্ধ ফাংশন

```
>>> my_list = []
>>> def my_impure_function(arg):
...     my_list.append(arg)
...
>>> my_impure_function(10)
>>> print(my_list)
[10]
```

উপরে `my_impure_function` এর কাছে একটি ভ্যালু গেলে সে তার বাইরে অবস্থান করা একটি লিস্ট যার নাম, `my_list` , এর মধ্যে সেই ভ্যালুকে ঢুকিয়ে দিচ্ছে। এই ফাংশনের কাজের গণ্ডি একটু বাড়তি। এটাকে ইম্পিওর ফাংশন বলা হয়।

ল্যাম্বডা

এই চ্যাপ্টারটি সর্বশেষ হালনাগাদ হয়েছেঃ Sat Jul 01 2017 18:12:36 GMT+0000 (UTC) সময়ে

সাধারণভাবে যখন `def` কিওয়ার্ড ব্যবহার করে একটি ফাংশন তৈরি করা হয় তখন স্বয়ংক্রিয় ভাবে এটিকে একটি ভ্যারিয়েবলে অ্যাসাইন করে দেয়া হয় যার মাধ্যমে একে পরবর্তীতে কল করা যায়। আবার অন্যদিকে, খুব সহজেই স্ট্রিং বা ইন্টিজার টাইপ ভ্যালুকে কোন রকম ভ্যারিয়েবলে অ্যাসাইন করা ছাড়াও তৈরি করা যায়। ঠিক এই সুবিধাটি (ভ্যারিয়েবলে অ্যাসাইন না করা) ফাংশনের ক্ষেত্রেও উপযোগ করা যায় এবং `lambda` এর মাধ্যমে। এভাবে তৈরি ফাংশনকে anonymous ফাংশন বলা হয়ে থাকে।

ল্যাম্বডার ব্যবহার খুব ফলপ্রসূ হয় যখন খুব সিম্পল যেমন এক লাইনের একটি ফাংশনকে আরেকটি ফাংশনের আর্গুমেন্ট হিসেবে পাঠানোর দরকার পরে। অর্থাৎ যখন সেই এক লাইনের কাজ করা ফাংশনকে আলাদা ভাবে `def` দিয়ে ডিফাইন/তৈরি করা অনর্থক মনে হয়।

`lambda x,y: x+y` - প্রথমে lambda কিওয়ার্ড লিখে এর আর্গুমেন্ট গুলোকে লেখা হয় এবং একটি কোলন দেয়ার পর এই ল্যাম্বডা তথা ফাংশনের কর্মকাণ্ড লেখা হয়। যেমন এই ল্যাম্বডাটি দুটো আর্গুমেন্ট নেয় এবং কাজের কাজ বলতে সেই দুটোকে যোগ করে।

উদাহরণ -

```
>>> def my_function(func, arg):
...     return func(arg)
...
>>> print(my_function(lambda x: 2 * x, 5))
10
```

উপরের উদাহরণে, `my_function` আর্গুমেন্ট হিসেবে একটি ফাংশন এবং একটি ভ্যালু নেয়। এরপর, আমরা যখন `my_function` কে কল করছি এবং তার মধ্যে একটি ফাংশন এবং একটি ভ্যালু পাঠিয়ে দেয়ার দরকার মনে করছি তখন ফাংশন না পাঠিয়ে একটি ল্যাম্বডা `lambda x: 2 * x` কে পাঠাচ্ছি এবং 5 পাঠাচ্ছি। ওদিকে, `my_function` সেই ল্যাম্বডাকে ফাংশন হিসেবে ধরে নিয়ে এক্সিকিউট করছে এবং যেহেতু সেই ল্যাম্বডা ফাংশনের আবার একটি আর্গুমেন্ট আছে x, তার জন্য নিজের রিসিভ করা আর্গুমেন্ট 5 কে পাঠাচ্ছে (ফরওয়ার্ড করছে)।

আরেকটি উদাহরণ,

```
>>> print((lambda x,y: x + 2 * y)(2,3))
8
```

এখানকার ল্যাম্বডাটি দুটো আর্গুমেন্ট x এবং y নিয়ে `x+2y` সূত্র ব্যবহার করে একটি রেজাল্ট রিটার্ন করে। আমরা 2 এবং 3 কে পাঠিয়েছি এবং রিটার্ন হিসেবে 8 পেয়েছি যেটা print এর মাধ্যমে প্রকাশিত হয়েছে। যেহেতু ল্যাম্বডা anonymous ফাংশন তাই একে আলাদা করে কল করার দরকার হয় না। এ ধরনের ফাংশনের একটি অসুবিধা হচ্ছে এর মধ্যে শুধু এক লাইনের এক্সপ্রেশন/কোড প্রসেস করা সম্ভব।

`print("Nuhil")` লিখে যেভাবে একটি String ভ্যালুকে কোথাও স্টোর করা ছাড়াই তৈরি এবং প্রিন্ট করা সম্ভব হল, সেভাবেই উপরের উদাহরণে `x+2y` নিয়ে কাজ করা ফাংশনকে তৈরি এবং কল করা দুটোই সম্ভব হল কোথাও স্টোর (def) করা ছাড়াই।

> সংকলন - নুহিল মেহেদী

ম্যাপ ও ফিল্টার

এই চ্যাপ্টারটি সর্বশেষ হালনাগাদ হয়েছেঃ Sat Jul 01 2017 18:12:36 GMT+0000 (UTC) সময়ে

লিস্ট এবং সমগোত্রীয় অবজেক্ট যাদেরকে পাইথনে iterable বলা হয়, তাদের উপর বিভিন্ন অপারেশনের জন্য ম্যাপ ও ফিল্টার খুবই উপকারী। এরা বিল্টইন ফাংশন।

ম্যাপ

`map` ফাংশনটি এর আর্গুমেন্ট হিসেবে একটি ফাংশন এবং একটি iterable নেয়। পাঠানো ফাংশনটি বস্তুত সেই iterable এর প্রত্যেকটি এলিমেন্টের উপর প্রয়োগ হয়। শেষে পরিবর্তিত iterable টিকে রিটার্ন করে। যেমন -

```
>>> def make_double(x):
...     return x * 2
...
>>> my_marks = [10, 12, 20, 30]
>>> result = map(make_double, my_marks)
>>> print(list(result))
[20, 24, 40, 60]
```

উপরের উদাহরণে প্রথমে একটি সাধারণ ফাংশন ডিফাইন করেছি যেটার কাজ হচ্ছে এর কাছে আসা যেকোনো নাম্বারকে দিগুণ করে রিটার্ন করে। তারপর আমরা আরেকটি লিস্ট ডিফাইন করেছি যার মধ্যে কিছু নাম্বার আছে। এরপর আমরা ম্যাপ ফাংশন কল করেছি এবং এর প্রথম আর্গুমেন্ট হিসেবে সেই `make_double` ফাংশনকে এবং দ্বিতীয় আর্গুমেন্ট হিসেবে `my_marks` লিস্ট (iterable) কে পাঠিয়েছি। ম্যাপ ফাংশন আরেকটি iterable কে রিটার্ন করে `result` ভ্যারিয়েবলের মধ্যে যেটা আসলে আগের লিস্টের মান গুলোর দিগুণ পরিমাণ নিয়ে গঠিত। শেষে, প্রিন্ট করার আগে সেই iterable কে list হিসেবে কনভার্ট করে প্রিন্ট করেছি।

ফিল্টার

filter ফাংশনের নাম শুনেই বোঝা যাচ্ছে এটা কোন কিছু ফিল্টার করে আলাদা করে। এই ফাংশন তার কাছে দেয়া কোন iterable থেকে কিছু এলিমেন্ট রিমুভ করে একটা প্রেডিকেট এর উপর ভিত্তি করে (প্রেডিকেট হচ্ছে ফাংশন যেটা বুলিয়ান ভ্যালু রিটার্ন করে)। ম্যাপের মত, ফিল্টারও দুটো আর্গুমেন্ট নেয় - একটা ফাংশন এবং একটা iterable (লিস্ট)।

ফিল্টার সেই সমস্ত এলিমেন্ট কেই রিমুভ করে যার জন্য এর কাছে পাঠানো ফাংশনের রিটার্ন তথা প্রেডিকেট মিথ্যা হয়। নিচের উদাহরণ দেখলে পরিষ্কার হয়ে যাবে -

```
>>> def is_even(x):
...     return x % 2 == 0
...
>>> my_numbers = [1,2,3,4,5,6]
>>> result = filter(is_even, my_numbers)
>>> print(list(result))
[2, 4, 6]
```

`is_even` ফাংশনটি এর কাছে আসা ভ্যালু জোড় হলে True এবং নাহলে False রিটার্ন করে। আর আমরা ফিল্টারের প্রথম আর্গুমেন্ট হিসেবে এই ফাংশনকেই পাঠিয়েছি। অন্যদিকে iterable হিসেবে `my_numbers` কে পাঠিয়েছি। এখন filter ফাংশন আমাদের লিস্টের প্রত্যেকটা এলিমেন্টের উপর সেই ফাংশনকে প্রয়োগ করে এবং যখন যখন এর রিটার্ন False হয় তখনকার এলিমেন্টটিকে রিমুভ করে শেষ নাগাদ নতুন একটা অবজেক্ট রিটার্ন করে।

ল্যাম্বডা রিভিউ

আগের চ্যাপ্টারে আমরা দেখেছি সহজ ও সিম্পল ফাংশন এবং যেটা স্টোর করার প্রয়োজন নেই সেগুলোকে কিভাবে lambda হিসেবে ডিফাইন করা ভালো প্র্যাকটিস। উপরের দুটি ক্ষেত্রেই আমরা দেখতে পাচ্ছি যে ম্যাপ বা ফিল্টারের কাছে পাঠানো প্রথম আর্গুমেন্ট যা কিনা একটা ফাংশন, সেটাকে কোথাও ডিফাইন/স্টোর করার প্রয়োজন পরছে না। তো, এই ফাংশনকে আমরা ল্যাম্বডা দিয়েই তৈরি ও ব্যবহার করতে পারি।

ফিল্টারের প্রোগ্রামটা আবার করছি একটা ল্যাম্বডা পাঠিয়ে -

```
>>> nums = [11, 22, 33, 44, 55]
>>> res = list(filter(lambda x: x % 2 == 0, nums))
>>> print(res)
[22, 44]
```

> সংকলন - নুহিল মেহেদী

জেনারেটর

এই চ্যাপ্টারটি সর্বশেষ হালনাগাদ হয়েছেঃ Sat Jul 01 2017 18:12:36 GMT+0000 (UTC) সময়ে

আগের চ্যাপ্টার গুলোতে iterable নিয়ে বেশ কিছু কথা বলা হয়েছে। লিস্ট, টাপল এসব হচ্ছে একধরনের iterable. জেনারেটরও এক রকমের iterable. কিন্তু লিস্ট এর মত এর এলিমেন্ট গুলোকে ইন্ডেক্সিং করা যায় না। কিন্তু তার মানে এই না যে, এর এলিমেন্ট গুলোকে অ্যাক্সেস করা যায় না। বরং for লুপ দিয়ে এর এলিমেন্ট গুলোকেও অ্যাক্সেস করা যায়। সব চেয়ে বড় কথা এলিমেন্ট এর চেইন তৈরি এবং অ্যাক্সেস এক সাথেই করা যায়।

সাধারণ ফাংশন এবং yield স্টেটমেন্ট ব্যবহার করেই এই বিশেষ ধরনের iterable কে তৈরি করা যায়। নিচের উদাহরণ দেখে নেই -

```
>>> def my_iterable():
...     i = 5
...     while i > 0:
...         yield i
...         i -= 1
...
>>> for i in my_iterable():
...     print(i)
...
5
4
3
2
1
```

এখানে my_iterable() দেখতে একটা সাধারণ ফাংশন। কিন্তু একটু একটু খেয়াল করলে দেখা যাবে, এখানে রিটার্ন এর বদলে yield কিওয়ার্ড ব্যবহার করা হয়েছে। এই ফাংশন খুব সহজ ভাবে while লুপ ব্যবহার করে 5 থেকে 1 পর্যন্ত রিটার্ন (yield) করে। কার কাছে রিটার্ন করে? ওই ফাংশনের নিচেই আমাদের তৈরি একটা for লুপ ওয়ালা স্টেটমেন্টের কাছে। এবং সেই লুপের মধ্যে print ব্যবহার করে এর কাছে আসা রিটার্ন ভ্যালুকে বার বার প্রিন্টও করা যাচ্ছে।

এক্ষেত্রে বলাই যায় যে, লিস্ট এর মত আমাদের my_iterable() -ও একটা iterable যাকে for লুপ দিয়ে অ্যাক্সেস করে কিছু ভ্যালু পাওয়া যায় যে ভ্যালু গুলো কিনা একটু আগেই আমাদের মত করেই তৈরি।

গুরুত্বপূর্ণ একটা ব্যপার খেয়াল করুন, সাধারণ কোন ফাংশনকে বার বার কল করলে সেই ফাংশন বার বার নতুন ভাবে এক্সিকিউট হয় এবং কাজ শেষে নতুন ভ্যালু রিটার্ন করে। কিন্তু এই ক্ষেত্রে একটা মজার ব্যপার ঘটছে। তা হল - যদিও for লুপ দিয়ে বার বার my_iterable() ফাংশনকে কল করা হচ্ছে কিন্তু ওই ফাংশনের মধ্যে থাকা i এর ভ্যালু কিন্তু ঠিকি সেইভ থাকছে (স্মরণ রাখছে) অর্থাৎ while লুপ টি প্রথমে i এর মান 5 তারপর 4 এভাবে রিটার্ন করছে। এমন না যে, প্রত্যেক বার 5 রিটার্ন হচ্ছে যেভাবে একটা সাধারণ ফাংশনকে একাধিক বার কল করলে হত।

আরেকটা উধাহরন -

```
>>> def even_numbers(x):
...     for i in range(x):
...         if i % 2 == 0:
...             yield i
...
>>> even_nums_list = list(even_numbers(10))
>>> print(even_nums_list)
[0, 2, 4, 6, 8]
```

আবার আসি সেই বিশেষ ধরনের একটা ফাংশন যা একাধারে কিছু ভ্যালু yield করে পক্ষান্তরে একে একটি iterable হিসেবে প্রকাশ করে। এখানে even_numbers() একটি ফাংশন তথা জেনারেটর (কারণ yield ব্যবহার করছে) যা একটি নির্দিষ্ট রেঞ্জ পর্যন্ত কিছু ভ্যালুর উপর for লুপ দিয়ে অপারেশন চালিয়ে সেখান থেকে শুধু মাত্র জোড় সংখ্যা গুলোকে yield (রিটার্ন) করে। ততক্ষণ পর্যন্ত রিটার্ন করে যতক্ষণ তার কাজের সীমা অর্থাৎ তার কাছে আর্গুমেন্ট হিসেবে আসা ভ্যালুর উপর for লুপ এর অপারেশনের শেষ পর্যন্ত।

ওদিকে খুব সহজেই আমরা ওই জেনারেটর কর্তৃক রিটার্ন করা ভ্যালু গুলোকে list() ফাংশনের মধ্যে দিয়ে সেখান থেকে একটি লিস্ট পেতে পারি যা শেষ লাইনে প্রিন্ট করে দেখানো হয়েছে।

ডেকোরেটর

এই চ্যাপ্টারটি সর্বশেষ হালনাগাদ হয়েছেঃ Sat Jul 01 2017 18:12:36 GMT+0000 (UTC) সময়ে

নাম শুনেই সবাই বুঝতে পারার কথা কোন কিছুর সৌন্দর্য বর্ধন করার মতই কিছু একটা হবে এখন। ডেকোরেটর হচ্ছে সাধারণ রকমেরই একটা ফাংশন যা অন্য আরেকটি ফাংশনকে মডিফাই করে তথা তার কাজকে বর্ধিত বা পরিবর্তিত করে।

অন্য ভাবে বলতে গেলে, যদি কখনো এমন দরকার পরে যে একটা ফাংশনের ফাংশনালিটি একটু পরিবর্তন/পরিবর্ধন করা দরকার কিন্তু আমরা সেই ফাংশনের কোড পরিবর্তন করতে চাচ্ছি না। তখন ডেকোরেটর ব্যবহার করে আমরা সেই কাজটা করতে পারবো।

একটা উদাহরণ -

```
>>> def my_decorator(func):
...     def decorate():
...         print("--------------")
...         func()
...         print("--------------")
...     return decorate
...
>>> def print_raw():
...     print("Clear Text")
...
>>> decorated_function = my_decorator(print_raw)
>>> decorated_function()
--------------
Clear Text
--------------
```

ধরুন আমাদের একটা সাধারণ ফাংশন আছে যার নাম print_raw এবং এটি খুব সাধারণ ভাবেই Clear Text এই বাক্যকে প্রিন্ট করে। এখন আমরা চাই যখনই আমি কোথাও Clear Text বাক্যকে প্রিন্ট করবো সেখানে যেন এর আগে পরে একটু স্টাইল যুক্ত হয় --- চিহ্ন দিয়ে। কিন্তু আবার চাচ্ছি না যে, print_raw ফাংশনটার কোড পরিবর্তন করতে।

তখন আমি একটি ডেকোরেটর বানালাম যার নাম my_decorator. এর একটি প্যারামিটার যা হচ্ছে একটি ফাংশন। এই my_decorator এর মধ্যে আমরা আরেকটি ফাংশন তৈরি করেছি যার নাম decorate.

my_decorator এর কাছে আসা ফাংশনকে এই decorate ফাংশনটি এক্সিকিউট করে। কিন্তু তার আগে ও পরে দুটি অতিরিক্ত প্রিন্ট স্টেটমেন্ট যোগ করে স্টাইল করে দেয়। পরিশেষে my_decorator ফাংশনটি এই decorate ফাংশন কে রিটার্ন করে।

এরপর আমরা my_decorator ফাংশন কে কল করেছি এবং এর আর্গুমেন্ট হিসেবে print_raw কে পাঠিয়ে দিয়েছি। এটা মডিফাই হয়ে ফিরে এসে decorated_function ভ্যারিয়েবলে জমা হয়েছে। অতঃপর, decorated_function() কল করে আমরা Clear Text এর স্টাইলড ভার্সন পাই।

ব্যাপারটাকে আরেক্টু সুন্দর করার জন্য আমরা ভ্যারিয়েবল রি-অ্যাসাইন এর সুবিধা নিতে পারি অর্থাৎ -

```
>>> print_raw = my_decorator(print_raw)
>>> print_raw()
--------------
Clear Text
--------------
```

এবার মনে হচ্ছে print_raw এর নাম ধাম ঠিকি আছে শুধু decorate হয়ে এসেছে :)

@decorator

ধরে নিচ্ছি my_decorator নামের একটি ডেকোরেটর ডিফাইন করা আছে। এখন আমরা চাইলে আমাদের তৈরি যেকোনো নতুন ফাংশনের উপর একে আপ্লাই করতে পারি। যেমন, আমরা যদি কোথাও নিচের মত একটা ফাংশন লিখি,

```
def print_text():
    print("Hello World!")
```

এবং চাই যে এর উপর আমাদের decorator এর স্টাইল আপ্লাই হোক। তাহলে খুব সহজ ভাবে আমরা নিচের মত করে একে একটি decorator এর আওতাধীন করতে পারি,

```
@my_decorator
def print_text():
    print("Hello World!")
```

তাহলে যখনই print_text কল করা হবে তখনি নিচের মত আউটপুট আসবে,

```
--------------
Hello World!
--------------
```

> সংকলন - নুহিল মেহেদী

রিকারসন

ফাংশনাল প্রোগ্রামিং -এ রিকারসন খুব গুরুত্বপূর্ণ একটি বিষয়। খুব সহজে বলতে গেলে, রিকারসন হচ্ছে এমন একটা অবস্থা যেখানে একটি ফাংশন নিজেকেই কল করে।

একটা সমস্যা যেটাকে সমাধানের জন্য ছোট ছোট ভাগে ভাগ করা যেতে পারে এবং প্রত্যেকটি ভাগের কাজ আবার অনেকটা একই রকম হবে, সেরকম ক্ষেত্রে রিকারসিভ ফাংশন তথা রিকারসন খুব কাজে লাগে।

বাস্তব উদাহরণ

ফ্যাক্টরিয়াল সম্পর্কে অনেকেই জানেন, একটা সংখ্যার ফ্যাক্টরিয়াল মানে হচ্ছে সেই সংখ্যা থেকে শুরু করে তার নিচের ক্রমিক সংখ্যা গুলোর প্রত্যেকটির সামগ্রিক গুণফল। অর্থাৎ, 5 এর ফ্যাক্টরিয়াল = 5x4x3x2x1 = 120

এটাকে এভাবেও চিন্তা করা যায়,

5 এর ফ্যাক্টরিয়াল
= 5x(4 এর ফ্যাক্টরিয়াল)
= 5x4x(3 এর ফ্যাক্টরিয়াল)
= 5x4x3(2 এর ফ্যাক্টরিয়াল) = 5x4x3x2(1 এর ফ্যাক্টরিয়াল)
= 5x4x3x2x1

অর্থাৎ প্রত্যেকবার একই কাজ করতে হয় কিন্তু আলাদা আলাদা সংখ্যার জন্য। এবং এই কাজের ফাংশন একটাই হলেই চলে। তাই কি করা যেতে পারে? একই ফাংশনকে বার বার কল করা অর্থাৎ নিজেকে নিজেই একতা নির্দিষ্ট সময় পর্যন্ত কল করা।

প্রোগ্রাম

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

print(factorial(5))
```

উপরের প্রোগ্রামটি দিয়েই যেকোনো সংখ্যার ফ্যাক্টরিয়াল বের করা সম্ভব। এখানে ফাংশনের শুরুতেই চেক করা হয়েছে যে সংখ্যার ফ্যাক্টরিয়াল বের করতে হবে সেটি 1 কিনা। যদি তাই হয় তাহলে ফ্যাক্টরিয়াল 1 এর মান 1 রিটার্ন করা হচ্ছে। এই অবস্থায় রিকারসন থেমে যায়। এটাকে বেইজ কেস বলা হয়।

এই কন্ডিশন মিথ্যা হলে আরেকটি জিনিষ রিটার্ন করা হয়। কি রিটার্ন করা হয় সেটাই মজার। রিটার্ন করা হচ্ছে সেই সংখ্যা এবং তার সাথে গুন আকারে ঠিক এই ফাংশনকেই (কল) শুধু আর্গুমেন্ট হিসেবে এক ক্রম কমিয়ে দিয়ে। এভাবে ঘটনা ক্রমে এবং প্রয়োজন অনুসারে একটি ফাংশন নিজেই নিজেকে কল করছে যেটাকেই রিকারসন বলা হয়।

আউটপুট

```
120
```

বেইজ কেস এর গুরুত্ব

নিচের প্রোগ্রামে কোন বেইজ কেস নাই কিন্তু একটি ফাংশন নিজেই নিজেকে কল করছে। অর্থাৎ এর কল থামার কোন লজিক সেট করা হয় নাই। এটা অনন্তকাল পর্যন্ত চলার চেষ্টা করবে।

```
def factorial(x):
  return x * factorial(x-1)

print(factorial(5))
```

আউটপুট

```
RuntimeError: maximum recursion depth exceeded
```

ডিরেকশন বা দিক

রিকারসন যেকোনো দিকেই ঘটতে পারে। অর্থাৎ প্রথম একটি ফাংশন আরেকটি দ্বিতীয় ফাংশনকে কল করতে পারে আবার সেই দ্বিতীয় ফাংশন প্রথেম ফাংশনকে কল করতে পারে যেটা কিনা আবার দ্বিতীয় ফাংশনকে কল করতে পারে।

উদাহরণ

```
def is_even(x):
  if x == 0:
    return True
  else:
    return is_odd(x-1)

def is_odd(x):
  return not is_even(x)


print(is_odd(17))
print(is_even(23))
```

আউটপুট

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

সেট

এই চ্যাপ্টারটি সর্বশেষ হালনাগাদ হয়েছেঃ Sat Jul 01 2017 18:12:36 GMT+0000 (UTC) সময়ে

লিস্ট এবং ডিকশনারির মতই সেটও এক ধরনের ডাটা স্ট্রাকচার। { } ব্র্যাকেট অথবা set ফাংশন ব্যবহার করে সেট তৈরি করা যায়। লিস্টের মতই কিছু ফাংশন সেট এরও আছে যেমন in ব্যবহার করে কোন এলিমেন্ট এর অস্তিত্ব চেক করা।

সাধারণ গণিতের সেট এর সাথে এই সেট এর অনেক মিল আছে। আমরা পরবর্তীতে কিছু উদাহরণ এর মাধ্যমে সেগুলো দেখবো।

যেমন,

```
num_set = {1, 2, 3, 4, 5}
word_set = set(["spam", "eggs", "sausage"])

print(3 in num_set)
print("spam" not in word_set)
```

আউটপুট,

```
True
False
```

মজার ব্যাপার হচ্ছে, ফাকা সেট তৈরি করার সময় { } ব্যবহার করা যাবে না কারণ এটা ফাকা ডিকশনারি তৈরি করার সাথে কনফ্লিক্ট করে। বরং set() ব্যবহার করে ফাকা সেট তৈরি করতে হয়।

সেটের কিছু গুরুত্বপূর্ণ বৈশিষ্ট্য

- সেটের এলিমেন্ট গুলোর কোন ক্রম নেই অর্থাৎ এদেরকে ইন্ডেক্সিং করা যায় না
- একটি সেটে একই এলিমেন্ট একাধিক বার থাকতে পারে না
- একটি এলিমেন্ট কোন একটি সেটের অংশ কিনা সেটা খুব দ্রুত চেক করা যায়, লিস্ট এর তুলনায়

সেটের উপর কিছু অপারেশন নিচের মত করা যায়,

```
# Has some duplicate eliments such as 1
nums = {1, 2, 1, 3, 1, 4, 5, 6}
print(nums)

# To add an eliment to the set
nums.add(-7)

# To remove an element to the set
nums.remove(3)
print(nums)
```

আউটপুট,

```
{1, 2, 3, 4, 5, 6}
{1, 2, 4, 5, 6, -7}
```

সেটের বৈশিষ্ট্য থেকে সহজেই অনুমান করা যায়, মেম্বারশিপ টেস্ট, এবং ডুপ্লিকেট এলিমেন্ট রিমুভ করার জন্য set() এর ব্যাবহার উপযুক্ত।

গণিতের সাথে তুলনীয় কিছু অপারেশন

সাধারণ গণিতে সেট এ যেমন ইউনিয়ন, ইন্টারসেকশন, ডিফারেন্স ইত্যাদি অপারেশন গুলো আছে, তেমনি পাইথনের সেটেও এই অপারেশন গুলো ভ্যালিড।

ইউনিয়ন = |
ইন্টারসেকশন = &
ডিফারেন্স = -
সিমেট্রিক ডিফারেন্স = ^

উদাহরণ,

```
first = {1, 2, 3, 4, 5, 6}
second = {4, 5, 6, 7, 8, 9}

print(first | second)
print(first & second)
print(first - second)
print(second - first)
print(first ^ second)
```

আউটপুট,

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
{4, 5, 6}
{1, 2, 3}
{8, 9, 7}
{1, 2, 3, 7, 8, 9}
```

কিছু সিদ্ধান্ত

ইতোমধ্যে আমরা জেনেছি পাইথনে যে ডাটা স্ট্রাকচার গুলো আছে সেগুলো হচ্ছে - লিস্ট, ডিকশনারি, টাপল এবং সেট। কিন্তু একটা দ্বিধা দ্বন্দ্ব সব সময় কাজ করতে পারে - কোন সময় কোন ধরনের ডাটা স্ট্রাকচার ব্যবহার করা উচিৎ।

নিচের অনুসিদ্ধান্ত গুলো কাজে আসতে পারে,

- ডিকশনারি -
    - যখন key-value জোড় এর মাধ্যমে বেশ কিছু ভ্যালু নিয়ে কাজ করতে হবে
    - যখন key এর উপর ভিত্তি করে ডাটা খুঁজে নেয়ার প্রয়োজন পর্বে বেশি
    - যখন তখন ডাটা গুলোর পরিবর্তন দরকার পরলে
- লিস্ট -
    - যখন ডাটা গুলোর র্যান্ডোম অ্যাক্সেস দরকার পরবে না বরং ইনডেক্স ধরে অ্যাক্সেস করলেই হবে।
    - সাধারণ একটি iterable দরকার হলে লিস্ট নিয়ে কাজ করা যেতে পারে
- সেট -
    - যখন এলিমেন্ট গুলোর মধ্যে ইউনিকনেস দরকার পরবে।
- টাপল -
    - যখন ডাটা পরিবর্তনের দরকার একদমই পরবে না। টাপল immutable.

> সংকলন - নুহিল মেহেদী

itertools

এই চ্যাপ্টারটি সর্বশেষ হালনাগাদ হয়েছেঃ Sat Jul 01 2017 18:12:36 GMT+0000 (UTC) সময়ে

এটি পাইথনের একটি স্ট্যান্ডার্ড মডিউল যার বেশ কিছু ফাংশন ব্যবহৃত হয় ফাংশনাল প্রোগ্রামিং এর সময়। যেমন,
`count` ফাংশন একটি নির্দিষ্ট ভ্যালু থেকে ইনফিনিট পর্যন্ত হিসাব করে।
`cycle` ফাংশন একটি iterable কে ইনফিনিট পর্যন্ত ইটারেট করে।
`repeat` ফাংশন ইনিফিনিট অথবা একটি নির্দিষ্ট পরিমাণ পর্যন্ত একটি অবজেক্টকে রিপিট করে।

উদাহরণ,

```
from itertools import count

for i in count(3):
    print(i)
    if i >= 11:
        break
```

আউটপুট,

```
3
4
5
6
7
8
9
10
11
```

ম্যাপ ও ফিল্টার যেমন কোন ইটারেবল এর উপর কাজ করে তেমনি itertools এর বেশ কিছু ফাংশন যেকোনো রকম iterable যেমন লিস্ট, ডিকশনারি এর উপর কাজ করতে সাহায্য করে। যেমন `accumulate` ফাংশনের মাধ্যমে একটি লিস্টের সব গুলো ভ্যালুর রানিং টোটাল পাওয়া সম্ভব।

উদাহরণ,

```
from itertools import accumulate

my_numbers = [1, 2, 3, 4, 5, 6]
accumulated_numbers = accumulate(my_numbers)
list_of_accu_nums = list(accumulated_numbers)
print(list_of_accu_nums)
```

আউটপুট,

```
[1, 3, 6, 10, 15, 21]
```

আরেকটি মজার ফাংশন `takewhile` যার নাম শুনেই বোঝা যাচ্ছে এটা কিছু সময় পর্যন্ত কিছু একটা নিয়ে নেয়। আর আগেই বলা হয়েছে এর অপারেশন হতে পারে যেকোনো ইটারেবলের উপর। এটা সেই সব ভ্যালুকে বের করে নেয় যেগুলোর জন্য একটি নির্দিষ্ট প্রেডিকেট সত্য হয়। নিচের উদাহরণে `lambda x: x <= 6` ল্যাম্বডাটি একটি প্রেডিকেট। ল্যাম্বডা নিয়ে পড়তে হবে এখানে

উদাহরণ,

```
from itertools import takewhile

my_numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
nums_less_equal_six = takewhile(lambda x: x <= 6, my_numbers)
filtered_numbers = list(nums_less_equal_six)
print(filtered_numbers)
```

আউটপুট,

```
[1, 2, 3, 4, 5, 6]
```

আরও ফাংশন এবং উদাহরণ,

```
from itertools import product, permutations

letters = ("A", "B")
print(list(product(letters, range(2))))
print(list(permutations(letters)))
```

আউটপুট,

```
[('A', 0), ('A', 1), ('B', 0), ('B', 1)]
[('A', 'B'), ('B', 'A')]
```

> সংকলন - নুহিল মেহেদী

অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং

এর আগে চ্যাপ্টার গুলোতে দু ধরনের প্রোগ্রামিং কনসেপ্ট দেখানো হয়েছে। ইম্পারেটিভ এবং ফাংশনাল। বিভিন্ন স্টেটমেন্ট, লুপ, ফাংশন এবং সাবরুটিন ব্যবহার করে সাধারণ প্রোগ্রামিং -কে ইম্পারেটিভ প্রোগ্রামিং বলা হয়ে থাকে। আবার পিউর ফাংশন, হাইয়ার অর্ডার ফাংশন, রিকারসন ব্যবহার করে যে ধরনের প্রোগ্রামিং করা হয় তাকে ফাংশনাল প্রোগ্রামিং বলা হয়ে থাকে।
ঠিক এরকম আরেকটি প্রোগ্রামিং কন্সেপ্ট/স্টাইল/ধরন এর নাম হচ্ছে অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং। এই কনসেপ্টের সাথে ক্লাস এবং অবজেক্ট এর সম্পর্ক ওতপ্রোত ভাবে জড়িত।

এই সেকশনে থাকছে

- ক্লাস
- ইনহেরিটেন্স
- ম্যাজিক মেথড
- অপারেটর অভারলোডিং
- অবজেক্ট লাইফ সাইকেল
- ডাটা হাইডিং
- স্ক্লাস মেথড ও ট্যাটিক মেথড
- প্রোপার্টিস

ক্লাস

এই চ্যাপ্টারটি সর্বশেষ হালনাগাদ হয়েছেঃ Sat Jul 01 2017 18:12:36 GMT+0000 (UTC) সময়ে

খুব সহজ ভাবে বলতে গেলে ক্লাস হচ্ছে এক ধরনের টেম্পলেট বা ব্লুপ্রিন্ট যার উপর ভিত্তি করে অনেক গুলো আলাদা আলাদা অবজেক্ট তৈরি করা সম্ভব। উদাহরণ সরূপ, যদি কখনো এমন প্রয়োজন হয় যে, একটি গেম ডেভেলপ করা দরকার এবং সেটায় বেশ কিছু দৈত্য, দানব থাকবে এবং সেগুলোকে একটা নায়ক চরিত্র ধ্বংস করে গেম ওভার করবে।
এরকম একটা গেম ডিজাইনে অবজেক্ট ওরিয়েন্টেড কনসেপ্ট খুবি উপকারী হতে পারে। যেমন, প্রথমেই সাধারণ একটা দৈত্য কেমন হয় তার উপর ভিত্তি করে একটা টেম্পলেট বা ব্লুপ্রিন্ট বানানো যেতে পারে। এরপর, গেমের প্রয়োজনে যতগুলো ভিন্ন ভিন্ন দৈত্য চরিত্র দরকার পরবে আমরা সেই টেম্পলেট ব্যবহার করে এবং দরকার হলে টুকটাক পরিবর্তন করে তত গুলো ভিন্ন ভিন্ন দৈত্য চরিত্র তৈরি করে নিতে পারবো।
মজার ব্যাপার হচ্ছে আমাদের তৈরি করা দৈত্য গুলোর কম্পিউটার মেমোরিতে আলাদা আলাদা অস্তিত্ব থাকবে। অর্থাৎ সেগুলো স্বাধীন এক একটা দৈত্য যাদের সবার মধ্যে মূল জিনিষ গুলোতে মিল আছে (যেহেতু একই টেম্পলেট থেকে তৈরি করা) এবং আলাদা আলাদা কিছু বৈশিষ্ট্য এবং আচরণও আছে।

> এই যে একটা দৈত্য বানানোর টেম্পলেট যেখান থেকে অনেক গুলো দৈত্য চরিত্র তৈরি করা যায়, সেটাকে বলা হয় ক্লাস। আর সেই ক্লাস ব্যবহার করে তৈরি করা আলাদা আলাদা দৈত্য চরিত্র গুলোকে বলা হয় অবজেক্ট।

`class` কিওয়ার্ড ব্যবহার করে পাইথনে ক্লাস তৈরি করা হয়, একটি ক্লাসের মধ্যে বিভিন্ন মেথড(ফাংশন) এবং অ্যাট্রিবিউট (প্রপার্টি) থাকতে পারে যেগুলো পাইথনের নিয়ম অনুযায়ী ইন্ডেন্টেড ব্লকের মধ্যে থাকে।

উদাহরণ,

```
# The blueprint to create monsters
class Monster:
    def __init__(self, color, heads):
        self.color = color
        self.heads = heads

# Create some real monsters
fogthing = Monster("Black", 5)
mournsnake = Monster("Yellow", 4)
tangleface = Monster("Red", 3)

# Check whether those monsters got different existence in memory or not
print('I have ' + str(fogthing.heads) + ' heads and I\'m ' + fogthing.color)
print('I also have ' + str(mournsnake.heads) + ' heads and I\'m ' + mournsnake.color)
print('I got ' + str(tangleface.heads) + ' heads and I\'m ' + tangleface.color)
```

আউটপুট,

```
I have 5 heads and I'm Black
I also have 4 heads and I'm Yellow
I got 3 heads and I'm Red
```

উপরে প্রথমেই একটি ক্লাস (আমাদের ভাষায় ব্লুএপ্রিন্ট বা টেম্পলেট) তৈরি করা হয়েছে। এর মধ্যে একটি ম্যাজিক মেথড আছে `__init__` (যেটা নিয়ে নিচে আলোচনা আছে) এবং এর দুটো অ্যাট্রিবিউট আছে `color` ও `heads` এবং এই ক্লাসকে ব্যবহার করে বা ইন্সট্যান্টশিয়েট করে ৩টি অবজেক্ট তৈরি করা হয়েছে যেগুলো কিনা নিজেরা আলাদা আলাদা।

`__init__` মেথড

এই মেথডটি যেকোনো ক্লাসের খুবি গুরুত্বপূর্ণ একটি মেথড। যখনি কোন ক্লাস থেকে কোন অবজেক্ট বা ইন্সট্যান্স তৈরি করা হয় তখনি এই মেথডটি স্বয়ংক্রিয় ভাবে কল হয়। ক্লাসের মেথডের ক্ষেত্রে আরেকটি গুরুত্বপূর্ণ বিষয় হচ্ছে - সব মেথডের প্রথম প্যারামিটারটি হতে হয় `self` . যদিও এই মেথড গুলোকে কল করার সময় নির্দিষ্ট করে এই আর্গুমেন্টটি পাঠাতে হয় না (পাইথন নিজে থেকেই এটা ম্যানেজ করে)।

বস্তুত এই `self` , ওই মেথডকে কল করা ইন্সট্যান্সটিকেই নির্দেশ করে। অর্থাৎ, উপরের উদাহরণ অনুযায়ী যখন `fogthing = Monster("Black", 5)` লিখে `fogthing` নামের একটি অবজেক্ট তৈরি করা হচ্ছে। তখন কিন্তু `__init__` কল হচ্ছে স্বয়ংক্রিয় ভাবে। আর এই `__init__` মেথডের কাছে প্রথম আর্গুমেন্ট `self` হিসেবে চলে যাচ্ছে এই `fogthing` ইন্সট্যান্সটি। আর ওই মেথডের ডেফিনেশনের মধ্যে সেই `self` এর দুটো অ্যাট্রিবিউট `color` এবং `heads` কে সেট করা হচ্ছে (এর কাছে আসা দ্বিতীয় ও তৃতীয় আর্গুমেন্ট এর ভ্যালু নিয়ে)। তাহলে পক্ষান্তরে কিন্তু `fogthing` অবজেক্ট এর দুটো অ্যাট্রিবিউট সেট হয়ে গেলো অবজেক্ট তৈরির সাথে সাথেই।

আর সাধারণ ভাবেই,

```
print('I have ' + str(fogthing.heads) + ' heads and I\'m ' + fogthing.color)
```

স্টেটমেন্টের মাধ্যমে `fogthing.heads` সিনট্যাক্স ব্যবহার করে এর অ্যাট্রিবিউটকে অ্যাক্সেস করা হচ্ছে।

> dot চিহ্ন ব্যবহার করে একটি অবজেক্টের অ্যাট্রিবিউট এবং মেথড গুলোকে অ্যাক্সেস করা হয়। `__init__` মেথড কে কন্সট্রাক্টর বলা হয়

ক্লাসের অন্যান্য মেথড

আগেই বলা হয়েছে ক্লাস হচ্ছে এক ধরনের ব্লুপ্রিন্ট যাকে ব্যবহার করে বিভিন্ন অবজেক্ট বা ইন্সট্যান্স তৈরি করা যায়। এখন, বাস্তবে একটি অবজেক্ট (উদাহরণ অনুযায়ী দৈত্য চরিত্র গুলো) এর যেমন কিছু বৈশিষ্ট্য (অ্যাট্রিবিউট - color, heads) থাকে, তেমনি কিছু কার্যকলাপ বা সক্রিয়তা থাকে। তো, ক্লাস তথা ব্লুপ্রিন্টের মধ্যে সেগুলোকে মেথড হিসেবে ডিফাইন করা হয়।

উদাহরণ,

```
class Monster:
    def __init__(self, color, heads):
        self.color = color
        self.heads = heads

    def attack(self):
        print("Just attacked a Hero, Mu...hahahaha!!!")

# Create an instance/object/monster-character
mournsnake = Monster("Yellow", 4)
# Check if its created or not
print('I am a ' + str(mournsnake.heads) + ' headed monster.')
# Make an attack by the new monster
mournsnake.attack()
```

আউটপুট,

```
I am a 4 headed monster.
Just attacked a Hero, Mu...hahahaha!!!
```

উপরের উদাহরণে, `Monster` ক্লাসের মধ্যে একটি কাস্টম মেথড ডিফাইন করা হয়েছে `attack` নামে। অর্থাৎ সাধারণ ভাবে যেকোনো দৈত্যের আক্রমণ করার ক্ষমতা থাকতেই পারে। আর এই ক্লাস থেকে যখন কোন ইন্সট্যান্স তৈরি করা হবে তখন সেই ইন্সট্যান্স তথা অবজেক্টেরও আক্রমণ করার ক্ষমতা থাকবে। অর্থাৎ, `attack` মেথডটিকে সেই অবজেক্টও ব্যবহার করতে পারবে।

ক্লাস অ্যাট্রিবিউট

একটি ক্লাসের মধ্যে সাধারণ কিছু বৈশিষ্ট্য ডিফাইন করা যায় যেগুলো একটু অন্য রকম অর্থাৎ নির্দিষ্ট কোন অবজেক্টের বাধ্যগত না। যেমন, `__init__` মেথডের সাহায্য নিয়ে, এর মধ্যেকার `self` কে কাজে লাগিয়ে আমরা একটি ক্লাস থেকে তৈরি করা অবজেক্টের কিছু অ্যাট্রিবিউট সেট করতে পারি যেগুলো ওই অবজেক্টের অ্যাট্রিবিউট।
কিন্তু, একটি ক্লাসের এমন কিছু অ্যাট্রিবিউট থাকতে পারে যেগুলো এর সব অবজেক্টই অ্যাক্সেস করতে পারবে তথা সব অব্জেক্টেরই অ্যাট্রিবিউট বলা যেতে পারে। এদেরকে ক্লাস অ্যাট্রিবিউট বলা হয়।

উদাহরণ,

```python
class Monster:
    identity = "negative character"

    def __init__(self, color, heads):
        self.color = color
        self.heads = heads

    def attack(self):
        print("Just attacked a Hero, Mu...hahahaha!!!")


mournsnake = Monster("Yellow", 4)
tangleface = Monster("Red", 3)

print('I am a ' + str(mournsnake.heads) + ' headed ' + mournsnake.identity)
print('I am a ' + str(tangleface.heads) + ' headed ' + tangleface.identity)
```

আউটপুট,

```
I am a 4 headed negative character
I am a 3 headed negative character
```

উপরের উদাহরণে, `Monster` ক্লাসের একটি `ক্লাস অ্যাট্রিবিউট` আছে যার নাম `identity` এবং এটা এই ক্লাসের সব অবজেক্টের জন্যই এক এবং সব অবজেক্টই অ্যাক্সেস করতে পারে।
আবার, যেহেতু এটা একটা ক্লাস অ্যাট্রিবিউট তাই একে ক্লাসের নাম দিয়েও অ্যাক্সেস করা যায় নিচের মত করে, `print(Monster.identity)` যার আউটপুট আসবে, `negative character`

রিভিউ

> ক্লাস হচ্ছে ব্লুপ্রিন্ট যার উপর ভিত্তি করে মেমোরিতে ওই টাইপের কিছু অবজেক্ট তৈরি করা যায়। `class` কিওয়ার্ড ব্যবহার করে ক্লাস তৈরি করা হয়। ক্লাসের কিছু নিজস্ব অ্যাট্রিবিউট থাকে যাদেরকে ক্লাস অ্যাট্রিবিউট বলে। এদেরকে ক্লাস অথবা ইন্সট্যান্স দিয়ে অ্যাক্সেস করা যায়। এছাড়া ক্লাসের কিছু ইন্সট্যান্স অ্যাট্রিবিউট থাকতে পারে যেগুলোকে ইন্সট্যান্স এর অ্যাট্রিবিউট বলা হয় এবং ইন্সট্যান্স এর মাধ্যমেই অ্যাক্সেস করা হয়। `__init__` মেথড এর সাহায্যে যেকোনো ক্লাস থেকে অবজেক্ট তৈরির সময় ব্যাসিক কিছু কাজ করে নেয়া যায়। এছাড়া ক্লাসের অনেক কাস্টম মেথড থাকতে পারে যেগুলো পক্ষান্তরে ওই ক্লাস থেকে তৈরি অবজেক্ট গুলোরই মেথড হিসেবে ব্যবহার করা যায়।

সংকলন - নুহিল মেহেদী

ইনহেরিট্যান্স

এই চ্যাপ্টারটি সর্বশেষ হালনাগাদ হয়েছেঃ Sat Jul 01 2017 18:12:36 GMT+0000 (UTC) সময়ে

ইংলিশ শব্দ inheritance এর একটা বাংলা অর্থ হচ্ছে উত্তরাধিকার সূত্রে কিছু পাওয়া। পাইথনে ক্লাস গুলোর মধ্যে কিছু ফাংশনালিটি ও বৈশিষ্ট্য শেয়ার করার একটা পদ্ধতি হচ্ছে ইনহেরিট্যান্স। অর্থাৎ, একটা ক্লাসকে ইনহেরিট (অনুসরণ) করে তার কিছু বৈশিষ্ট্য আরেকটি উত্তরসূরি ক্লাসের মধ্যে নিয়ে আসার ব্যাপারটিকেই ইনহেরিট্যান্স বলা হয়।

আগের চ্যাপ্টারে আমরা একটা ক্লাস দেখেছি `Monster` . এই ক্লাস থেকে আমরা বিভিন্ন রকম আলাদা আলাদা অবজেক্ট তথা দৈত্য তৈরি করা দেখেছি। একটি ক্লাসে আমরা সেই সব প্রোপার্টি এবং মেথড রাখি যেগুলো এই ক্লাস থেকে বানানো অবজেক্টের কাজে লাগে। কিন্তু যদি এমন হয় যে, `Monster` ক্লাস থেকে বানানো অবজেক্ট গুলোর আরও নিজস্ব কিছু অ্যাক্টিভিটি (মেথড) দরকার যেগুলো আমাদের `Monster` ক্লাস বা টেম্পলেটে নাই। তাহলে কি হবে? আমরা কি তবে ওই ক্লাস থেকে অবজেক্ট বানানো বাদ দিয়ে দেবো? আরেকটি ক্লাস বানিয়ে সেই নতুন ক্লাস থেকে অবজেক্ট তথা একটু ভিন্ন বৈশিষ্ট্যের দৈত্য বানানো শুরু করবো?

না। বরং আমরা শুধুমাত্র বাড়তি প্রয়োজনীয় ফিচার গুলো নিয়ে একটি ছোট ক্লাস বানাবো ঠিকি কিন্তু কমন বৈশিষ্ট্য গুলোর সুবিধা নেয়ার জন্য সেই `Monster` ক্লাসকেই ইনহেরিট করবো।

উদাহরণ,

```
class Monster:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def attack(self):
        print('I am attacking...')


class Fogthing(Monster):
    def make_sound(self):
        print('Grrrrrrrrrr\n')


class Mournsnake(Monster):
    def make_sound(self):
        print('Hiiisssssshhhh\n')


fogthing = Fogthing("Fogthing", "Yellow")
fogthing.attack()
fogthing.make_sound()

mournsnake = Mournsnake("Mournsnake", "Red")
mournsnake.attack()
mournsnake.make_sound()
```

ধরে নিচ্ছি আমাদের সব দৈত্যই আক্রমণ করে এবং একটা নাম এবং একরকম রং আছে। তাই এসব মিলে আমরা একটি ব্লুপ্রিন্ট (ক্লাস) বানিয়েছি `Monster` নামে। এবার সত্যিকারের দৈত্য অবজেক্ট তৈরির সময় মনে হল যে, `fogthing` আর `mournsnake` দৈত্যের শব্দ তো আলাদা রকম। তখন আমরা বাড়তি এই ফাংশনটা সহ `Fogthing` আর `Mournsnake` নামের দুটো ক্লাস বানিয়ে নিয়েছি কিন্তু সেগুলো মূল `Monster` ক্লাসকে ইনহেরিট করছে। অর্থাৎ নতুন এই দুটি ক্লাসের নিজস্ব কিছু ফিচার আছে সাথে `Monster` ক্লাসের সব ফিচারও পাচ্ছে।

আউটপুট,

```
I am attacking...
Grrrrrrrrr

I am attacking...
Hiiissssshhhh
```

> কোন ক্লাসকে ইনহেরিট করার জন্য ওই ক্লাসের নামটি নতুন ক্লাসের নামের পর ব্র্যাকেটের মধ্যে লিখতে হয়। এখানে `Monster` হচ্ছে সুপারক্লাস আর `Fogthing`, `Mournsnake` হচ্ছে সাবক্লাস

অভাররাইড
যদি সুপারক্লাসের কোন মেথড বা অ্যাট্রিবিউটকে এর একটা সাবক্লাসের মধ্যে আবার ডিফাইন করা হয় তাহলে সেগুলো অভাররাইড হয়ে যায়। যেমন,

```
class Monster:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def attack(self):
        print('I am attacking...')


class Fogthing(Monster):
    def attack(self):
        print('I am killing...')

    def make_sound(self):
        print('Grrrrrrrrr\n')


fogthing = Fogthing("Fogthing", "Yellow")
fogthing.attack()
fogthing.make_sound()
```

আউটপুট,

```
I am killing...
Grrrrrrrrr
```

উপরের উদাহরণে, `Fogthing` ক্লাস মুল `Monster` ক্লাসের `attack` মেথডকে অভাররাইড করেছে।

মাল্টিপল ইনহেরিট্যান্স

নিচের প্রোগ্রামটি দেখি এবং অনুমান করি এর আউটপুট কি আসবে,

```
class A:
    def where(self):
        print("I am from class A")


class B:
    def where(self):
        print("I am from class B")


class C(A, B):
    pass

an_instance_of_c = C()
an_instance_of_c.where()
```

এখানে, `C` ক্লাসটি `A` এবং `B` দুটো ক্লাসকেই ইনহেরিট করেছে। আবার ওই দুটো ক্লাসের প্রত্যেকটিতেই `where` নামের মেথড আছে। পরিশেষে, যখন `C` ক্লাসের ইন্সট্যান্স তৈরি করে `where` মেথডকে কল করা হচ্ছে তখন আসলে কোন ক্লাসের `where` মেথডটি কল হবে?

প্রথমেই সেই মেথডকে `C` ক্লাসের মধ্যে খোঁজা হবে, না পেলে `A` ক্লাসের মধ্যে আর সেখানেও না পেলে `B` ক্লাসের মধ্যে খোঁজা হবে। এখন বলা যায় আউটপুট কি আসবে,

```
I am from class A
```

এই অর্ডার ভিত্তিক মেথড খোঁজার জন্য পাইথন নিজস্ব নিয়ম ফলো করে এবং সেটা দেখতে চাইলে আমরা প্রোগ্রামে `print(C.mro())` স্টেটমেন্টটি ব্যবহার করতে পারি যার আউটপুট আসবে নিচের মত,

```
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
```

অর্থাৎ, `C` , `A` , `B`

`super` মেথড

ইনহেরিট্যান্স এর ক্ষেত্রে `super` একটি গুরুত্বপূর্ণ মেথড। এর মাধ্যমে একটি অবজেক্টের সুপার ক্লাসের মধ্যেকার মেথডকে কল করা যায়। যেমন,

```
class A:
    def spam(self):
        print(1)

class B(A):
    def spam(self):
        print(2)
        super().spam()

B().spam()
```

আউটপুট,

```
2
1
```

> সংকলন - নুহিল মেহেদী

ম্যাজিক মেথড

পাইথনে কিছু বিশেষ ধরনের বিল্ট ইন মেথড আছে যেগুলোকে ম্যাজিক মেথড বলা হয়। এগুলার চেনার খুব সহজ উপায় হচ্ছে এদের নামের দুই পাসেই দুটো করে আন্ডারস্কোর সিম্বল থাকে। অর্থাৎ, `__init__` মেথডের মত। এই মেথডের সঙ্গে ইতোমধ্যে আমাদের পরিচয় হয়েছে। কোন ক্লাসে এই ম্যাজিক মেথড ব্যবহার করলে এবং পরবর্তীতে সেই ক্লাসের ইন্সট্যান্স তৈরির সময় এই ম্যাজিক মেথডটি স্বয়ংক্রিয় ভাবেই কল হয় যাতে করে এর মাধ্যমে কিছু সেটআপ রিলেটেড কাজ করে নেয়া যায়।

এই মেথড গুলোকে ভাষায় প্রকাশ করার সময় একটু জটিলতা হয়। যেমন, "আন্ডারস্কোর আন্ডারস্কোর ইনিট আন্ডারস্কোর আন্ডারস্কোর" এভাবে বললে অদ্ভুত শোনায়। তাই এদেরকে সুন্দর ভাবে "dunders" তথা "ডাণ্ডার ইনিট" এভাবে বলা হয়ে থাকে।

তো, এই `__init__` মেথড বাদেও অনেক গুলো ম্যাজিক মেথড আছে পাইথনে। ম্যাজিক মেথডের খুব বহুল ব্যবহার দেখা যায় অপারেটর অভারলোডিং এর সময় যা পরের চ্যাপ্টারেই আলোচনা করা হয়েছে। প্রত্যেকটি অপারেটর এর জন্যই একটি ম্যাজিক মেথড আছে। যেমন, `+` অপারেটর এর জন্য ম্যাজিক মেথডটি হচ্ছে `__add__`

ঘটনাটি এভাবে ঘটে, যদি আমাদের এমন একটি এক্সপ্রেশন থাকে `x+y` এবং `x` বস্তুত `K` ক্লাসের ইন্সট্যান্স হয়। তখন পাইথন `K` ক্লাসের ডেফিনেশন চেক করবে। যদি `K` ক্লাসের একটি মেথড থাকে `__add__` তাহলে সেটাকে কল করা হবে এভাবেঃ `x.__add__(y)`

কিছু কমন অপারেটরের ম্যাজিক মেথডঃ

`__sub__` হচ্ছে `-` জন্য

`__mul__` হচ্ছে `*` জন্য

`__truediv__` হচ্ছে `/` জন্য

`__floordiv__` হচ্ছে `//` জন্য

`__mod__` হচ্ছে `%` জন্য

`__pow__` হচ্ছে `**` জন্য

`__and__` হচ্ছে `&` জন্য

`__xor__` হচ্ছে `^` জন্য

`__or__` হচ্ছে `|` জন্য

তুলনা করার অপারেটর গুলোর জন্যও পাইথনে ম্যাজিক মেথড আছেঃ

`__lt__` হচ্ছে `<` জন্য

`__le__` হচ্ছে `<=` জন্য

`__eq__` হচ্ছে `==` জন্য

`__ne__` হচ্ছে `!=` জন্য

`__gt__` হচ্ছে `>` জন্য

`__ge__` হচ্ছে `>=` জন্য

এছাড়াও আরও অনেক ম্যাজিক মেথড আছে পাইথনে যেমন - `__len__` `__getitem__` `__setitem__` `__delitem__` `__iter__` `__contains__` ইত্যাদি

অপারেটর ওভারলোডিং

আমরা আগের চ্যাপ্টারগুলোতে নানাবিধ অপারেশন দেখেছি - যোগ, বিয়োগ, গুন ভাগ ইত্যাদি । পাইথনের একটা চমৎকার ফিচার হচ্ছে এই অপারেটরগুলোর ফাংশনালিটি পরিবর্তন করা যায় ।

প্রথমেই আমরা দেখে নেই, এই অপারেটর গুলো আসলে কিভাবে কাজ করে । আমরা যখন কোন অপারেটর ব্যবহার করি, যেমন:

```
a = b + c
```

পাইথন এই `+` অপারেশনের ফলাফল নির্নয়ের জন্য ইন্টারনালি `b` অবজেক্টের `__add__` মেথডে আর্গুমেন্ট হিসেবে `c` কে পাস করে দেয় । ঐ মেথডের রিটার্ন ভ্যালুই হয় উক্ত অপারেশনের ফলাফল । অর্থাৎ, উপরে দেখানো অপারেশনটি আসলে এভাবে কাজ করে -

```
a = b.__add__(c)
```

এখানে লক্ষ্য করুন, আপনি যদি এখন এই `b` অবজেক্টের `__add__` মেথডটি পরিবর্তন করে দেন, তাহলে ঐ `b` অবজেক্টের উপর `+` অপারেশনের ফাংশনালিটিও পরিবর্তন হয়ে যাচ্ছে ।

এই জিনিসটাই হচ্ছে অপারটের ওভারলোডিং ।

আসুন একটি উদাহরন দেখে নেইঃ

```
class MyNum():
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return (self.value * 2) + (other.value * 2)


a = MyNum(2)
b = MyNum(3)

c = a + b

print(c)
```

উপরের উদাহরণে, `MyNum` ক্লাস এর ইন্সট্যান্স হিসেবে আমরা দুটো অবজেক্ট তৈরি করলাম এবং এদের প্রত্যেকের `value` সেট করলাম `2` এবং `3` . এরপর এই দুটি অবজেক্টকে সাধারণ যোগ চিহ্ন দিয়ে যোগ করলাম। আমরা চাচ্ছি এই স্পেশাল নাম্বার দুটো একটু আলাদা ভাবে আমাদের নিজেদের মত করে যোগ হয়ে ফিরে আসুক। এক্ষেত্রে, `c`

`= a + b` লাইনে বস্তুত `c = a.__add__(b)` স্টেটমেন্টটি এক্সিকিউট হবে। অর্থাৎ `a` অবজেক্টের `__add__` মেথড কল হবে এবং এর মধ্যে প্রথমে `2` এর ভ্যালু দিগুণ হবে, অতঃপর `b` অবজেক্ট তথা `other` প্যারামিটার এর `value` `3` ও দিগুণ হবে। পরিশেষে এদের যোগ ফল রিটার্ন হবে। অর্থাৎ আউটপুট,

```
10
```

আরও একটি উদাহরণ দেখে নেইঃ

```
class MyInt():
    def __init__(self, value):
        self.__value = value

    def __int__(self):
        return self.__value

    def __add__(self, other):
        return self.__value + int(other) * int(other)


a = MyInt(2)
b = MyInt(3)

c = a + b

print(c)
```

কমন অপারেটর ও তাদের জন্য ব্যবহৃত স্পেশাল মেথড:

- `+` => `__add__`
- `-` => `__sub__`
- `*` => `__mul__`
- `/` => `__div__`

ইনপ্লেইস অপারেটর ওভারলোডিং
ইনপ্লেইস অপারেটরগুলোর জন্যও এরকম মেথড রয়েছে । এগুলো হলো:

- `__iadd__`
- `__isub__`
- `__imul__`
- `__idiv__`

তাই আর ব্যখ্যায় না গিয়ে কোড দেখে ফেলি:

```
class MyInt():
    def __init__(self, value):
        self.__value = value

    def __int__(self):
        return self.__value


    def __iadd__(self, other):
        return self.__value + int(other) * int(other)


a = MyInt(2)

a += MyInt(3)

print(a)
```

কোড রান না করেই আন্দাজ করে বলতে পারবেন আউটপুট কি আসতে পারে?

সংকলন - আবু আশরাফ মাসনুন

অবজেক্ট লাইফ সাইকেল

একটা অবজেক্টের পুরো জীবন কাল তথা মেমোরিতে এর আলাদা অস্তিত্ব হিসাব হয় প্রধান তিনটি বিষয়ের উপর - Creation, Manipulation, Destruction.

প্রথমেই যখন একটি ক্লাস তৈরি করা হয় যার উপর ভিত্তি করে বিভিন্ন অবজেক্ট তৈরি হবে, তখনি লাইফ সাইকেলের প্রথম ধাপটি শুরু হয়। এর পরের ধাপটি হচ্ছে, যখন একটি অবজেক্ট ইন্সট্যান্সিয়েট হয় অর্থাৎ `__init__` মেথড কল হয়। এ সময় নতুন অবজেক্টটির জন্য মেমোরি অ্যালোকেট হয়। এর একটু আগে অবশ্য ক্লাসের `__new__` মেথডটিও কল হয়। এর পর নতুন অবজেক্টটি ব্যবহারের উপযোগী হয়। অর্থাৎ অন্যান্য কোড এই অবজেক্টের অ্যাট্রিবিউট ও মেথড গুলোকে কল করতে পারে। ব্যবহার শেষে এই অবজেক্টকে ডিস্ট্রয় করা যায় (একেই গারবেজ কালেকশন বলা হয়ে থাকে)।

উদাহরণ,

```
## Definition
class Add:
## Initialization
    def __init__(self,a,b):
        self.a = a
        self.b = b
    def add(self):
        return self.a+self.b
obj = Add(3,4)
## Access
print obj.add()
## Garbage collection
```

যখন একটি অবজেক্ট ধ্বংস (Destroy) করা হয় তখন তার জন্য পূর্ব নির্ধারিত মেমোরি টুকু ফ্রি হয়ে যায় এবং সেটা অন্য কাজে লাগানো যায়।

রেফারেন্স কাউন্ট

পাইথনে স্বয়ংক্রিয় ভাবে অপ্রয়োজনীয় অবজেক্ট ধ্বংস হয়ে যায় যখন একটি অবজেক্টের রেফারেন্স কাউন্ট জিরো হয়ে যায়। একটি অবজেক্ট যতগুলো ভ্যারিয়েবল বা অন্যান্য এলিমেন্ট এর সাথে প্রত্যক্ষ ভাবে সম্পর্কিত থাকে অর্থাৎ যতগুলো ভ্যারিয়েবল বা এলিমেন্ট ওই অবজেক্টকে নির্দেশ করে সেই সংখ্যাকে ওই অবজেক্টের রেফারেন্স কাউন্ট বলা হয়। যদি কোন ভ্যারিয়েবল বা এলিমেন্ট ওই অবজেক্টকে নির্দেশ না করে তাহলে বলা যায় সেটির রেফারেন্স কাউন্ট শূন্য তথা সেটাকে মেমোরি থেকে ডিলিট করা যেতে পারে। যখন কোন অবজেক্টের রেফারেন্স কাউন শূন্য হয়ে যায় তখন পাইথন সেটাকে স্বয়ংক্রিয় ভাবে ডিলিট করে দেয়। এটাকে একটি প্রোগ্রামিং ল্যাঙ্গুয়েজের Automatic Memory Management ফিচার বলা হয়ে থাকে। `del` কিওয়ার্ড ব্যবহার করেও ম্যানুয়ালি কোন অবজেক্টের রেফারেন্স কাউন্ট কমানো যায়।

```
a = 42  # <42> অবজেক্টটি তৈরি হল
b = a  # a এর মান b তে ঢুকানো হল এবং তাই <42> রেফারেন্স কাউন্ট বেড়ে গেল
c = [a]  # a কে একটি লিস্টের এলিমেন্ট হিসেবে অন্তর্ভুক্ত করা হল এবং তাই <42> রেফারেন্স কাউন্ট আরও বেড়ে গেল

del a  # <42> এর রেফারেন্স কাউন্ট কমানো হল
b = 100  # b এখন a কে নয় বরং অন্য একটি ভ্যালুকে নির্দেশ করে। তাই <42> এর রেফারেন্স কাউন্ট আরও কমে গেলো
c[0] = -1  # c লিস্টের যে অবস্থানে a ছিল সেখানে এখন অন্য ভ্যালু তথা রেফারেন্স কাউন্ট আরও কমে গেলো
```

ডাটা হাইডিং

অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং -এ এনক্যাপসুলেশন (Encapsulation) একটি গুরুত্বপূর্ণ কনসেপ্ট যার অর্থ কিছু ভ্যারিয়েবল এবং ফাংশনকে একত্রিত করে একটি সিঙ্গেল ইউনিট হিসেবে প্রকাশ করা। এই কনসেপ্টে একটি ক্লাসের ভ্যারিয়েবল গুলোকে অন্য ক্লাস এর কাছ থেকে আড়ালে রাখা হয় এবং শুধুমাত্র ওই ক্লাসের নির্দিষ্ট মেথডের মাধ্যমে অ্যাক্সেস করার পারমিশন থাকে। এজন্য এই কনসেপ্টকে ডাটা হাইডিং -ও বলা হয়ে থাকে। অন্যভাবে বলা হয়, একটি ক্লাসের ইমপ্লিমেন্টেশন ডিটেইল গুলো আড়ালে রাখা।

> সাধারণ অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং এর চারটি গুরুত্বপূর্ণ কনসেপ্ট হচ্ছে - **encapsulation, inheritance, polymorphism, and abstraction.**

অন্যান্য প্রোগ্রামিং ল্যাঙ্গুয়েজে একটি ক্লাসের অ্যাট্রিবিউট ও মেথড গুলোকে নির্দিষ্ট কিওয়ার্ড (অ্যাক্সেস মডিফায়ার) ব্যবহার করে প্রাইভেট বা প্রটেক্টেড হিসেবে ডিফাইন করে এই উদ্দেশ্য পূরণ করা হয়ে থাকে। এর মাধ্যমে ওই ক্লাসের ওই নির্দিষ্ট মেথড বা অ্যাট্রিবিউট গুলোকে বাইরের ক্লাস থেকে অ্যাক্সেস করা থেকে বিরত রাখা হয়।

কিন্তু, পাইথনে এই বিষয়টাকে একটু আলাদাভাবে দেখা হয়। বলা হয়ে থাকে - *"we are all consenting adults here"* অর্থাৎ - একটি ক্লাসের কোন এলিমেন্টকে শক্তভাবে বাইরের অ্যাক্সেস থেকে বিরত রাখার ব্যবস্থা করা উচিৎ নয়। আর তাই, পাইথনে সত্যিকারের কোন পদ্ধতি নেই যার মাধ্যমে একটি ক্লাসের অ্যাট্রিবিউট বা মেথডকে প্রাইভেট হিসেবে ডিফাইন করা যেতে পারে। বরং, এধরনের এলিমেন্ট গুলোকে বাইরে থেকে অ্যাক্সেস করতে নিরুৎসাহিত করা হয় এবং এগুলো যে আসলে ক্লাসের ইমপ্লিমেন্টেশন ডিটেইল তা প্রকাশ করার মাধ্যমে এগুলোর সরাসরি অ্যাক্সেস/ব্যবহার বন্ধ রাখতে বলা হয়।

উইকলি প্রাইভেট
অ্যাট্রিবিউট এবং মেথডের নামের শুরুতে একটি আন্ডারস্কোর ব্যবহার করে এরকম প্রাইভেট এলিমেন্ট গুলোকে ডিফাইন করা হয়ে থাকে। আবার বলতে হচ্ছে - এভাবে প্রাইভেট এলিমেন্ট হিসেবে ডিফাইন করে এটাই প্রকাশ করা হয় যে, বাইরের কোড থেকে এগুলো অ্যাক্সেস করার দরকার নেই বা উচিৎ নয়। কিন্তু তার মানে এই না যে এগুলোকে অ্যাক্সেস করা যাবে না।

```
class Queue:
    def __init__(self, contents):
        self._hiddenlist = list(contents)

    def push(self, value):
        self._hiddenlist.insert(0, value)

    def pop(self):
        return self._hiddenlist.pop(-1)

    def _show_list(self):
        return self._hiddenlist


queue = Queue([1, 2, 3])
print(queue._hiddenlist)

queue.push(0)
print(queue._hiddenlist)

queue.pop()
print(queue._hiddenlist)

print(queue._show_list())
```

আউটপুট,

```
[1, 2, 3]
[0, 1, 2, 3]
[0, 1, 2]
[0, 1, 2]
```

উপরের উদাহরণে, কিছু উইকলি প্রাইভেট এলিমেন্ট ডিফাইন করা থাকলেও সেগুলো ক্লাসের বাইরে থেকে অ্যাক্সেস করা গেছে।

কিন্তু হ্যাঁ, এভাবে ডিফাইন করা ভ্যারিয়েবল নিয়ে তৈরি একটি পাইথন ফাইলকে মডিউল হিসেবে ইম্পোরট করলে ওই প্রাইভেট ভ্যারিয়েবল গুলো ইম্পোরট হয় না। এতে করে এগুলোর সরাসরি অ্যাক্সেস বাধাগ্রস্ত রাখা হয়। অর্থাৎ, `from module_name import *` কোড ব্যবহার করলেও `module_name` এর মধ্যে থাকা আন্ডারস্কোর ওয়ালা ভ্যারিয়েবল গুলো ইম্পোরট হবে না।

উদাহরণ,

```
# myfile.py
_my_private_variable = 10
```

```
# data-hiding-test.py
from myfile import *


print(_my_private_variable)
```

`data-hiding-test.py` ফাইলকে রান করলে আউটপুট আসবে,

```
Traceback (most recent call last):
  File "/Users/nuhil/Documents/Python/data-hiding-test.py", line 4, in <module>
    print(_my_private_variable)
NameError: name '_my_private_variable' is not defined
```

স্ট্রংলি প্রাইভেট
এ ধরনের অ্যাট্রিবিউট ও মেথডের নামের শুরুতে ডাবল আন্ডারস্কোর ব্যবহার করা হয়। পাইথন এরকম নামের অ্যাট্রিবিউট বা মেথড পেলে এগুলোর নামকে আরেক্টু পরিবর্তন করে ফেলে। এতে করে ক্লাসের বাইরে থেকে সেই ডিফাইন করা নামে এদেরকে আর অ্যাক্সেস করা যায় না।

> মূলত অ্যাক্সেস রোধ করার জন্য এরকম করা হয় না বরং একটি ক্লাসের সাবক্লাসে যদি একই নামের এলিমেন্ট থাকে তাহলে যেন সেগুলোর সাথে কনফ্লিক্ট না করে।

উদাহরণ,

```
class Spam:
    __egg = 7

    def print_egg(self):
        print(self.__egg)

s = Spam()
s.print_egg()
print(s.__egg)
```

আউটপুট,

```
7
Traceback (most recent call last):
  File "/Users/nuhil/Documents/Python/myfile.py", line 10, in <module>
    print(s.__egg)
AttributeError: 'Spam' object has no attribute '__egg'
```

যদিও নিচের মত করে ঠিকি ওই প্রাইভেট এলিমেন্টকে অ্যাক্সেস করা যায়,

```
class Spam:
    __egg = 7

    def print_egg(self):
        print(self.__egg)

s = Spam()
s.print_egg()
print(s._Spam__egg)
```

আউটপুট,

```
7
7
```

অর্থাৎ, পাইথন আসলে আড়ালে, ডাবল আন্ডারস্কোর ওয়ালা এলিমেন্টের নামের সাথে তার ক্লাসের নামটি জুড়ে দেয় আর তাই `s._Spam__egg` ব্যবহার করে `Spam` ক্লাসের `__egg` কে অ্যাক্সেস করা হয়েছে।

> সংকলন - নুহিল মেহেদী

ক্লাস মেথড ও স্ট্যাটিক মেথড

ক্লাস মেথড

আমরা আগেই জেনেছি, ইন্সট্যান্স মেথডকে একটি ক্লাসের ইন্সট্যান্স এর মাধ্যমে কল করা হয় এবং সেই ইন্সট্যান্সকে ওই মেথডের `self` প্যারামিটার হিসেবে পাঠানো হয় (ক্লাসের মেথড গুলোর প্রথম প্যারামিটার হিসেবে `self` ডিফাইন করতে হয়)।

কিন্তু ক্লাস মেথড একটু আলাদা। এ ধরনের মেথডকে সরাসরি ক্লাসের মাধ্যমেই কল করা হয় এবং সেই ক্লাস কে ওই মেথডের `cls` প্যারামিটার হিসেবে পাঠানো হয় (ক্লাস মেথডের প্রথম প্যারামিটার সাধারণত `cls` হয়ে থাকে)।

`classmethod` ডেকোরেটর ব্যবহার করে ক্লাস মেথড নির্দেশিত করা হয়। যেমন,

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

    @classmethod
    def new_square(cls, side_length):
        return cls(side_length, side_length)

square = Rectangle.new_square(5)
print(square.calculate_area())
```

উপরের উদাহরণে, `new_square` একটি ক্লাস মেথড। আর তাই একে আমরা ক্লাসের মাধ্যমেই কল করতে পারি। একটি বিষয় লক্ষণীয় যে, এ ধরনের মেথডের প্রথম প্যারামিটার হিসেবে `cls` তথা সেই ক্লাসকেই পাঠানো হয়।

> ইন্সট্যান্স মেথডের `self` এবং ক্লাস মেথডের `cls` এর নামকরণ শুধুই একটু কনভেনশন। আলাদা নামও চাইলে ব্যবহার করা যেতে পারে।

আর সেই `new_square` মেথডের প্যারামিটার হচ্ছে একটি। তার মানে আমরা এই মেথডকে কল করতে পারছি একটি মাত্র প্যারামিটার দিয়েই এবং যেহেতু তার প্রথম প্যারামিটার হিসেবে সেই ক্লাসটি নিজেই নির্দেশিত হচ্ছে তার মানে ওই `new_square` মেথডের মধ্যে থেকে আমরা সেই ক্লাস তথা `Rectangle` কেই ধরে সেটাকে ইন্সট্যান্সিয়েট করতে পারি। `return cls(side_length, side_length)` লাইনে আমরা ঠিক সেই কাজটিই করছি অর্থাৎ, `Rectangle` ক্লাসের কন্সট্রাক্টর এর দুটি প্যারামিটারের চাহিদা মোতাবেক দুটি প্যারামিটারই পাঠিয়ে ফ্রেশ একটি `Rectangle` ক্লাসের অবজেক্ট ইনিশিয়েট করেছি এবং রিটার্ন করছি।

তার মানে, `square = Rectangle.new_square(5)` লাইনের মাধ্যমে আমরা `square` ভ্যারিয়েবলের মধ্যে বস্তুত স্ট্যান্ডার্ড `Rectangle` ক্লাসের অবজেক্ট পাচ্ছি। আর তাই শেষ লাইনে সেই অবজেক্টের মেথড তথা একটি স্বাভাবিক ইন্সট্যান্স মেথড `calculate_area` কে কল করে আশানরূপ ফল পাই।

উপরের প্রোগ্রামের আউটপুট,

```
25
```

> ক্লাস মেথডের বহুল ব্যবহার হতে পারে ফ্যাক্টরি মেথড তৈরি জন্য যেখানে একটি ক্লাসের অবজেক্ট দরকার হলে আমরা চাইলে ওই ক্লাসের কন্সট্রাক্টরের চাহিদা মোতাবেক আর্গুমেন্ট না পাঠিয়েও আরেকটি মেথডের মাধ্যমে (এ ক্ষেত্রে ক্লাস মেথড) ওই ক্লাসের স্বাভাবিক একটি অবজেক্ট পেতে পারি।

স্ট্যাটিক মেথড
স্ট্যাটিক মেথড অনেকটাই ক্লাস মেথডের মত যেমন, সরাসরি ক্লাস এর মাধ্যমেই কল করা যায়। কিন্তু আবার একটু আলাদা যেমন, ক্লাস মেথডের মত এই মেথড এর প্রথম প্যারামিটার হিসেবে কলার ক্লাসকে পাঠাতে হয় না।
আর তাই, সহজ ভাবে স্ট্যাটিক মেথডকে নরমাল ফাংশনের সাথে তুলনা করা হয় কিন্তু যা বিশেষত ক্লাসের এলিমেন্ট অর্থাৎ ক্লাস বা ক্লাসের ইন্সট্যান্স এর মাধ্যমে কল করা যায়। `staticmethod` ডেকোরেটর ব্যবহার করে স্ট্যাটিক মেথড ডিফাইন করা হয়।

উদাহরণ,

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    @staticmethod
    def validate_topping(topping):
        if topping == "pineapple":
            raise ValueError("No pineapples!")
        else:
            return True

ingredients = ["cheese", "onions", "spam"]
if all(Pizza.validate_topping(i) for i in ingredients):
    pizza = Pizza(ingredients)
```

উপরের প্রোগ্রামটি কোন এক্সেপশন ছাড়াই রান করবে। এখানে `validate_topping` একটি স্ট্যাটিক মেথড। ফর লুপ ব্যবহার করে `Pizza.validate_topping(i)` স্টেটমেন্টের মাধ্যমে `ingredients` লিস্টের প্রত্যেকটি এলিমেন্টের জন্য আমরা স্ট্যাটিক মেথডটিকে কল করে একটা সাধারণ চেকিং এর কাজ সম্পন্ন করেছি এবং তা সফল হলে `Pizza` ক্লাসের অবজেক্ট তৈরি করেছি।

> সংকলন - নুহিল মেহেদী

প্রোপার্টিস

কোন একটি মেথডের উপর `property` ডেকোরেটোর ব্যবহার করে প্রোপার্টি ডিফাইন করা হয়। এর একটা বহুল ব্যবহার দেখা যায় কোন ইন্সট্যান্স অ্যাট্রিবিউটকে রিড-অনলি বানানোর ক্ষেত্রে। যখন একটি ইন্সট্যান্স অ্যাট্রিবিউটকে কল করা হয় এবং যদি ওই নামে একটি মেথড থাকে যা কিনা `property` ডেকোরেটোর দিয়ে ডেকোরেট করা, তখন পক্ষান্তরে সেই মেথডটিই কল হয়। আর এভাবেই একটি অ্যাট্রিবিউটকে রিড-অনলি করার একটা রাস্তা পাওয়া যায়। নিচের উদাহরণটি দেখলে আরও পরিষ্কার হয়ে যাবে।

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    @property
    def pineapple_allowed(self):
        return False

pizza = Pizza(["cheese", "tomato"])
print(pizza.pineapple_allowed)
pizza.pineapple_allowed = True
```

আউটপুট,

```
False
Traceback (most recent call last):
  File "/Users/nuhil/Documents/Python/property.py", line 11, in <module>
    pizza.pineapple_allowed = True
AttributeError: can't set attribute
```

`setter/getter` ফাংশন ব্যবহার করেও প্রোপার্টি ডিফাইন করা যায়। `setter` ফাংশন ব্যবহার করে প্রোপার্টির ভ্যালু সেট করা যায়। আর `getter` ফাংশন ব্যবহার করে ওই প্রোপার্টির ভ্যালু অ্যাক্সেস করা যায়। `setter` ডিফাইন করার জন্য প্রোপার্টির নাম এবং একটি ডট চিহ্ন দিয়ে `setter` কিওয়ার্ড লিখে একটি ডেকোরেটর হিসেবে নির্দেশ করতে হয়। `getter` ডিফাইন করার ক্ষেত্রেও একই নিয়ম।

উদাহরণ,

```
1  class Pizza:
2      def __init__(self, toppings):
3          self.toppings = toppings
4          self._pineapple_allowed = False
5
6      @property
7      def pineapple_allowed(self):
8          return self._pineapple_allowed
9
10     @pineapple_allowed.setter
11     def pineapple_allowed(self, value):
12         if value:
13             password = input("Enter the password: ")
14             if password == "Sw0rdf1sh!":
15                 self._pineapple_allowed = value
16         else:
17             raise ValueError("Alert! Intruder!")
18
19 pizza = Pizza(["cheese", "tomato"])
20 print(pizza.pineapple_allowed)
21 pizza.pineapple_allowed = True
22 print(pizza.pineapple_allowed)
23
```

উপরের প্রোগ্রামে, প্রথমত ৭ নাম্বার লাইনে `pineapple_allowed` মেথডকে একটি প্রোপার্টি ডিফাইন করা হয়েছে। এতে করে, ২০ নাম্বার লাইনে এই প্রোপার্টির ভ্যালু অ্যাক্সেস করতে গেলে বস্তুত `pineapple_allowed` মেথডটি কল হয় এবং যা পক্ষান্তরে `self._pineapple_allowed = False` এর উপর ভিত্তি করে `False` রিটার্ন করে। এরপর, ১০ নাম্বার লাইনে, `@pineapple_allowed.setter` ডেকোরেটর ব্যবহার করে ওই প্রোপার্টির জন্য একটি `setter` মেথড ডিফাইন করা হয়েছে। আর তাই, ২১ নাম্বার লাইনে `pizza.pineapple_allowed = True` স্টেটমেন্ট এক্সিকিউট হবার সময় আসলে `pineapple_allowed` সেটার মেথডটি কল হচ্ছে। এই মেথডটি ১৫ নাম্বার লাইনে, `_pineapple_allowed` নামের প্রাইভেট ভ্যারিয়েবলের মান বদলে দেয় যার প্রমাণ পাওয়া যায় ২২ নাম্বার লাইনে নতুন করে `print(pizza.pineapple_allowed)` প্রিন্টের মাধ্যমে।

আউটপুট,

```
False
Enter the password: Sw0rdf1sh!
True
```

এই সেকশনে থাকছে

রেগুলার এক্সপ্রেশন হচ্ছে স্ট্রিং ম্যানিপুলেশন বা স্ট্রিং নিয়ে কাজ করার জন্য অন্যতম বহুল ব্যবহৃত একটি টুল। এটা একধরনের ডোমেইন স্পেসিফিক ল্যাঙ্গুয়েজ। অর্থাৎ পুরো পুরি আলাদা কোন প্রোগ্রামিং ল্যাঙ্গুয়েজ নয় কিন্তু এর মাধ্যমে কাজ করার জন্য এর নির্দিষ্ট গ্রামার মেনে ইন্সট্রাকশন লিখতে হয়। সব পপুলার প্রোগ্রামিং ল্যাঙ্গুয়েজের জন্যই রেগুলার এক্সপ্রেশনের লাইব্রেরী আছে। SQL আরেকটি ডোমেইন স্পেসিফিক ল্যাঙ্গুয়েজের উদাহরণ।

রেগুলার এক্সপ্রেশনের মাধ্যমে প্রধানত দুই ধরণের কাজ করা হয়। প্রথমত কোন স্ট্রিং এর মধ্যে নির্দিষ্ট প্যাটার্ন খোঁজার জন্য এবং দ্বিতীয়ত, ম্যাচ পাওয়া গেলে কোন নির্দিষ্ট কাজ করার জন্য যেমন সেখানে স্ট্রিং রিপ্লেস করা।

- পরিচিতি
- মেটা ক্যারেক্টার
- ক্যারেক্টার ক্লাস
- গ্রুপ
- স্পেশাল সিকুয়েন্স

পরিচিত

পাইথনে রেগুলার এক্সপ্রেশন নিয়ে কাজ করার জন্য বিল্ট ইন মডিউল হিসেবে আছে `re` নামের মডিউল। রেগুলার এক্সপ্রেশন ব্যবহারের সময় প্যাটার্ন খোঁজার জন্য যে স্পেশাল এক্সপ্রেশন বা সহজ করে বলতে সার্চ টার্ম ডিফাইন করতে হয় সেটা সাধারণত `r"expression"` এভাবে ডিফাইন করতে হয়। `r` দিয়ে Raw স্ট্রিং বোঝানো হয়। অর্থাৎ এর মধ্যে কোন রকম ক্যারেক্টার এস্কেইপ করা হয় না যা রেগুলার এক্সপ্রেশনের ব্যবহারকে আরও ইফিসিয়েন্ট করে তোলে।

প্রথমেই একটি উদাহরণ দেখি -

```
import re

pattern = r"Bangla"

result = re.match(pattern, "Bangladesh")

if result:
    print("Match Found!")
else:
    print("No match")
```

আউটপুট,

```
Match Found!
```

প্রথমেই `re` মডিউলকে import করে নেয়া হয়েছে যাতে করে এর মধ্যেকার সব ফাংশনকে সহজে আমাদের প্রোগ্রামে ব্যবহার করতে পারি। এরপর একটি ভ্যারিয়েবলে Raw ফরম্যাটে Bangla স্ট্রিং টিকে স্টোর করা হয়েছে। বস্তুত এই প্যাটার্নটিকেই আমরা একটু পরে আরেকটি স্ট্রিং এর মধ্যে খুঁজব। এরপর `re` এর `match` ফাংশন কে কল করা হয়েছে এবং এর দুটো আর্গুমেন্ট পাঠিয়ে দেয়া হয়েছে - একটি হচ্ছে কি ম্যাচ করে দেখতে চাই, আরেকটি হচ্ছে কোথায় ম্যাচ করে দেখতে চাই। `match` ফাংশন একটি স্ট্রিং এর শুরুতে ডিফাইন করা প্যাটার্নকে খুঁজে দেখে।

এই অপারেশনটির রেজাল্ট স্টোর করা হয়েছে `result` ভ্যারিয়েবলে। সাধারণত, নির্দিষ্ট প্যাটার্ন ম্যাচ পাওয়া গেলে এখানে একটি ম্যাচ সম্বলিত অবজেক্ট পাওয়া যাবে আর ম্যাচ পাওয়া না গেলে None রিটার্ন আসবে। এরপরের if-else এর কাজ টুকু সবাই বুঝতে পারছেন আশা করি।

এরকম আরও মজার সব ফাংশন আছে `re` মডিউলে। যেমন - `search` , `findall` , `finditer` ইত্যাদি। আমরা নিচে একটি প্রোগ্রামের মধ্যেই সব গুলোর ব্যবহার দেখবো এবং তারপর বিশ্লেষণ করবো।

```
import re

pattern = r"Bangladesh"

if re.search(pattern, "There is country named Bangladesh in south asia!"):
    print("Match Found!")
else:
    print("No match")

pattern = r"bangla"
print(re.findall(pattern, "Bangladeshi bangla and indian bangla are differnet."))
```

আউটপুট,

```
Match Found!
['bangla', 'bangla']
```

`search` ফাংশন এর মাধ্যমে একটি প্যাটার্নকে একটি স্ট্রিং এর যেকোনো যায়গায় খুঁজে দেখা হয়। `match` এর মত শুধু শুরুতে চেক করার মত নয়। `findall` ফাংশনও `search` এর মত সব যায়গায় ম্যাচ খুঁজে দেখে এবং খুঁজে পাওয়া সব গুলো ম্যাচকে একটি লিস্ট হিসেবে রিটার্ন করে। উপরের প্রোগ্রামে এই দুটি ফাংশনের ব্যবহারকেই দেখানো হয়েছে।

রিটার্ন অবজেক্টের কিছু মেথড
আগেও একবার বলা হয়েছে যে - রেগুলার এক্সপ্রেশন সম্বলিত একটি স্টেটমেন্ট বা অপারেশন একটি অবজেক্ট রিটার্ন করে। এই রিটার্ন করা অবজেক্টের আবার অনেক গুলো সুবিধাজনক মেথড থাকে যেগুলো ব্যবহার করে আরও কিছু গুরুত্বপূর্ণ কাজ সম্পন্ন করা যায়। যেমন - `group` , `start` , `end` , `span` ইত্যাদি।

উদাহরণ,

```
import re

pattern = r"bin"

match = re.search(pattern, "combination")
if match:
    print(match.group())
    print(match.start())
    print(match.end())
    print(match.span())
```

আউটপুট,

```
bin
3
6
(3, 6)
```

`group` মেথড রিটার্ন করছে ম্যাচ হয়ে যাওয়া সাব স্ট্রিং টিকে। `start` , `end` দিয়ে স্ট্রিং এর মধ্যে হওয়া ম্যাচ এর শুরু আর শেষের ইনডেক্স বা অবস্থান জানা যায়। `span` এর মাধ্যমে এই ইনডেক্স দুটিকে একটি টাপল হিসেবে রিটার্ন পাওয়া যায়।

> সংকলন - নুহিল মেহেদী

মেটা ক্যারেক্টার

আমরা এর আগে জেনেছি যে রেগুলার এক্সপ্রেশন হচ্ছে এক ধরণের ডোমেইন স্পেসিফিক ল্যাঙ্গুয়েজ। উদাহরণ হিসেবে জেনেছি SQL এর কোথা। তাই স্বাভাবিক ভাবেই এর মাধ্যমে জটিল কিছু লজিক বা প্যাটার্ন লিখতে হতেই পারে। আর তাই, রেগুলার এক্সপ্রেশন লেখার সময় কিছু মেটা ক্যারেক্টার এর ব্যবহার করতে হয়। এগুলোর ব্যবহারের মাধ্যমে বস্তুত রেগুলার এক্সপ্রেশনকে আরও ডাইনামিক ভাবে ব্যবহার করা যায়। নিচের কিছু উদাহরণ দেখলেই বিষয়টি পরিষ্কার হয়ে যাবে।

. (dot)
এই মেটা ক্যারেক্টারের মাধ্যমে যেকোনো ক্যারেক্টার ম্যাচ করার নির্দেশ দেয়া হয় (শুধু নিউ লাইন ক্যারেক্টার বাদে)।

উদাহরণ,

```
import re

pattern = r"gr.y"

if re.match(pattern, "grey"):
    print("Match 1")

if re.match(pattern, "gray"):
    print("Match 2")

if re.match(pattern, "blue"):
    print("Match 3")
```

আউটপুট,

```
Match 1
Match 2
```

উপরে আমরা একটি রেগুলার এক্সপ্রেশন ডিফাইন করেছি `r"gr.y"` এর মাধ্যমে। এখানে `.` দিয়ে ওই অবস্থানে যেকোনো ক্যারেক্টার এর সাথে ম্যাচ দেখতে বলা হয়েছে। আর তাই যখন `grey` বা `gray` এর সাথে ম্যাচ করা হয়েছে তখন রেজাল্ট সত্য এসেছে এবং একটি প্রিন্ট স্টেটমেন্ট এক্সিকিউট হয়েছে। `blue` এর ক্ষেত্রে তা হয় নি।

> ইতোমধ্যে হয়তো খেয়াল করেছেন এক্সপ্রেশন এর শুরুতে `r` এর ব্যবহার। এর মাধ্যমে একটি স্ট্রিং কে Raw বা শুধুই সাধারণ স্ট্রিং হিসেবে ডিফাইন করা হয়। এতে করে রেগুলার এক্সপ্রেশন এর মধ্যে থাকা "মেটা ক্যারেক্টার" এবং ওই "মেটা ক্যারেক্টারের মতই অন্য সাধারণ ম্যাচ করার ক্যারেক্টার" এর মধ্যে পার্থক্য তৈরি করা হয়।

^ এবং $

আরও দুটি বহুল ব্যবহৃত মেটা ক্যারেক্টার হচ্ছে `^` এবং `$`. এ দুটোর মাধ্যমে যথাক্রমে কোন একটি স্ট্রিং এর শুরু এবং শেষ চেক করে দেখা হয়। যেমন,

```
import re

pattern = r"^wr.te$"

if re.match(pattern, "write"):
   print("Match 1")

if re.match(pattern, "wrote"):
   print("Match 2")

if re.match(pattern, "writer"):
   print("Match 3")
```

আউটপুট,

```
Match 1
Match 2
```

উপরের প্রোগ্রামে `r"^wr.te$"` এর মাধ্যমে একটি স্ট্রিং যার শুরু এবং শেষ নির্দিষ্ট অর্থাৎ যথাক্রমে `w` এবং `e` কিন্তু `wr` এর পর যেকোনো ক্যারেক্টার থাকতে পারে এবং সেটির পর আবার `te` থাকতে হবে। তাই `write` এবং `wrote` ম্যাচ করেছে।

ক্যারেক্টার ক্লাস

ক্যারেক্টার ক্লাস হচ্ছে কিছু ক্যারেক্টারের সমষ্টি বা সেট। এর মাধ্যমে এই সেটের মধ্যেকার যেকোনো একটি ক্যারেক্টারের সাথে নির্দিষ্ট কোন স্ট্রিং -কে ম্যাচ করে দেখা যায়। আবার একটি নির্দিষ্ট রেঞ্জ পর্যন্ত ক্যারেক্টার এর সাথেও ম্যাচ করা যায়। `[]` বন্ধনী ব্যবহার করে এবং এর মধ্যে নির্দিষ্ট কিছু ক্যারেক্টার যুক্ত করে একটি ক্যারেক্টার ক্লাস তৈরি করা হয় যা পরবর্তীতে সার্চ প্যাটার্ন হিসেবে ব্যবহার করা হয়।

উদাহরণ,

একটি ক্যারেক্টার ম্যাচ

```
import re

# A character set containing all vowels
pattern = r"[aeiou]"

# Lets check whether a word got a vowel in it or not
if re.search(pattern, "grey"):
   print("The word 'grey' got at least one vowel!")
else:
   print("No vowel found!")

if re.search(pattern, "qwertyuiop"):
   print("The word 'qwertyuiop' got at least one vowel!")
else:
   print("No vowel found!")

if re.search(pattern, "rhythm myths"):
   print("The word 'rhythm myths' got at least one vowel!")
else:
   print("No vowel found!")
```

আউটপুট,

```
The word 'grey' got at least one vowel!
The word 'qwertyuiop' got at least one vowel!
No vowel found!
```

ক্যারেক্টার রেঞ্জ ম্যাচ

ক্যারেক্টার সেট তৈরি করার সময় `-` চিহ্ন দিয়ে একটি রেঞ্জ ডিফাইন করা হয়। যেমন, `[a-z]` ক্লাস দিয়ে ছোট হাতের যেকোনো ইংলিশ বর্ণ ম্যাচ করা হয়। `[A-Z]` দিয়ে যেকোনো বড় হাতের ইংলিশ বর্ণ। `[0-9]` দিয়ে যেকোনো নিউমেরিক ডিজিট ম্যাচ করে দেখা হয়, ইত্যাদি।

উদাহরণ,

```
import re

pattern = r"[A-Z][A-Z][0-9]"

if re.search(pattern, "NS1 is prefix of first name server address."):
   # Found NS1 as match
   print("OK")

if re.search(pattern, "You should put a second one with NS2 as prefix."):
   # Found NS2 as match
   print("OK")

if re.search(pattern, "I don\'t have any nameserver."):
   print("NS3")
else:
   print("Not OK!")

if re.search(pattern, "PY3K"):
   # Found PY3 as match
   print("OK")
```

আউটপুট,

```
OK
OK
Not OK!
OK
```

অর্থাৎ উপরের `[A-Z][A-Z][0-9]` প্যাটার্নের মাধ্যমে একটি স্ট্রিং এর মধ্যে "দুটি বড় হাতের ইংলিশ বর্ণ এবং তার সাথেই যুক্ত একটি নিউমেরিক ডিজিট" সম্পন্ন একটি প্যাটার্ন ম্যাচ করা হচ্ছে।

ক্যারেক্টার ক্লাসে ^ এর ব্যবহার

আমরা আগে জেনেছি যে, `^` হচ্ছে একটি মেটা ক্যারেক্টার। ক্যারেক্টার ক্লাসে `^` এর গুরুত্বপূর্ণ একটি ভূমিকা আছে। এর মাধ্যমে সাধারণ ভাবে তৈরি করা একটি ক্যারেক্টার ক্লাসের ঠিক উল্টো অর্থ ডিফাইন করা হয়। অর্থাৎ এটি একটি ক্যারেক্টার ক্লাসের অর্থকে invert করে ফেলে। অর্থাৎ `[A-Z]` দিয়ে যদি যেকোনো বড় হাতের ইংলিশ বর্ণের উপস্থিতি যাচাই করা হয়, তাহলে `[^A-Z]` দিয়ে যেকোনো বড় হাতের ইংলিশ বর্ণের অনুপস্থিতি যাচাই বা ম্যাচ করা হয়।

উদাহরণ,

```python
import re

# Match string that contains NOT ALL Capital letters
pattern = r"[^A-Z]"

if re.search(pattern, "a sentence with all lower case letters."):
    print("Match 1")

if re.search(pattern, "A sentence with mixed English letters."):
    print("Match 2")

if re.search(pattern, "HEADING"):
    # All Capital letters
    # No Match
    print("Match 3")

if re.search(pattern, "HEADING WITH ALL CAPITAL LETTERS"):
    # All Capital letters
    # but "spaces" makes it True to NOT ALL Capital
    print("Match 4")
```

আউটপুট,

```
Match 1
Match 2
Match 4
```

> ক্যারেক্টার ক্লাস ডিফাইন করার সময় `^` মেটা ক্যারেক্টারটির এর ভূমিকা থাকলেও অন্য মেটা ক্যারেক্টার যেমন - `.` বা `$` এর কোন ভূমিকা নেই।

গ্রুপ্স

রেগুলার এক্সপ্রেশনের একটি নির্দিষ্ট অংশকে বন্ধনীর মধ্যে আবদ্ধ করে একটি গ্রুপ তৈরি করা হয়।

উদাহরণ,

```
import re

pattern = r"egg(spam)*"

if re.match(pattern, "egg"):
    print("Match 1")

if re.match(pattern, "eggspamspamspamegg"):
    print("Match 2")

if re.match(pattern, "spam"):
    print("Match 3")
```

উপড়ে `(spam)` একটি গ্রুপ। অর্থাৎ উপরোক্ত প্যাটার্নটি এই প্রকাশ করে যে - স্ট্রিং এর শুরুতে egg থাকবে এবং এর পর এক বা একাধিক spam ওয়ার্ড থাকবে অথবা নাও থাকতে পারে (* দিয়ে প্রকাশ করা হয়েছে)।

আউটপুট,

```
Match 1
Match 2
```

গ্রুপের ম্যাচ করা কন্টেন্ট গুলকে group() ফাংশনের সাহায্যে অ্যাক্সেস করা যায়। যেমন, group() বা group(0) ব্যবহার করে পুরো ম্যাচটি অ্যাক্সেস করা যেতে পারে। নিচের মত করে,

উদাহরণ,

```
import re

pattern = r"a(bc)(de)(f(g)h)i"

match = re.match(pattern, "abcdefghijklmnop")
if match:
    print(match.group())
    print(match.group(0))
    print(match.group(1))
    print(match.group(2))
    print(match.groups())
```

আউটপুট,

```
abcdefghi
abcdefghi
bc
de
('bc', 'de', 'fgh', 'g')
```

স্পেশাল গ্রুপ

অনেক রকম স্পেশাল গ্রুপের মধ্যে named group এবং non-capturing group অন্যতম।

named group এর ফরম্যাট দেখতে `(?P<name>...)` -এ রকম। যেখানে name হচ্ছে গ্রুপটির নাম এবং ... হচ্ছে কন্টেন্ট। এর আচরণ অন্যান্য নরমাল গ্রুপের মতই, শুধুমাত্র যেহেতু এর একটি নাম আছে তাই একে অ্যাক্সেস করার জন্য group(name) অর্থাৎ নাম ব্যবহার করা যায়। যদিও সাথে সাথে নাম্বারও ব্যবহার করা যেতে পারে। অন্যদিকে, non-capturing group এর ফরম্যাট দেখতে (?: ...) -এ রকম এবং গ্রুপ ফাংশন ব্যবহার করে একে অ্যাক্সেস করা যায় না।

উদাহরণ,

```
import re

pattern = r"(?P<first>abc)(?:def)(ghi)"

match = re.match(pattern, "abcdefghi")
if match:
    print(match.group("first"))
    print(match.groups())
```

আউটপুট,

```
abc
('abc', 'ghi')
```

আরও একটি মেটাক্যারেক্টার

`|` অর্থাৎ অথবা প্রকাশক একটি মেটা ক্যারেক্টার মাঝে মাঝে খুবি উপকারী। এটা অনেক লজিক্যাল OR অপারেটর এর মত। নিচের উদাহরণ দেখলে আরও পরিষ্কার হয়ে যাবে,

```
import re

pattern = r"gr(a|e)y"

match = re.match(pattern, "gray")
if match:
    print ("Gray is fine!")

match = re.match(pattern, "grey")
if match:
    print ("Grey is OK also!")

match = re.match(pattern, "griy")
if match:
    print ("No way, what Griy is?!!")
```

অর্থাৎ, প্যাটার্নটি এরকম - প্রথমে gr থাকবে, এরপর হয় a অথবা e থাকবে এবং শেষে y থাকবে। এরকম একটি ম্যাচ খুঁজবে এই প্যাটার্নটি। আর তাই উপড়ের প্রোগ্রামের আউটপুট আসবে নিচের মত,

```
Gray is fine!
Grey is OK also!
```

স্পেশাল সিকোয়েন্স

একটি ব্যাকস্ল্যাশ `\` এবং সাথে একটি ক্যারেক্টার ব্যবহার করে রেগুলার এক্সপ্রেশনের জন্য একটি স্পেশাল সিকোয়েন্স তৈরি করা হয়। যেমন, একটি বহুল ব্যবহৃত সিকোয়েন্স হচ্ছে `\1` বা `\2` বা এরকম। এর মাধ্যমে ওই নাম্বার গ্রুপের (আগের চ্যাপ্টারে গ্রুপ নিয়ে আলোচনা আছে) এক্সপ্রেশনকে ম্যাচ করে দেখা হয়।

উদাহরণ,

```
import re

pattern = r"(.+) \1"

match = re.match(pattern, "word word")
if match:
    print ("Match 1")

match = re.match(pattern, "?! ?!")
if match:
    print ("Match 2")

match = re.match(pattern, "abc cde")
if match:
    print ("Match 3")
```

প্রথম ম্যাচ স্টেটমেন্টটি খেয়াল করি - এখানে `(.+)` দিয়ে প্রথম গ্রুপে একটি খুশি মত যেকোনো স্ট্রিং চেক করা হচ্ছে আর এর পরেই `\1` দিয়ে সেই গ্রুপের (group(1)) জন্য ম্যাচ হওয়া এক্সপ্রেশনকে (word) ম্যাচ করে দেখা হচ্ছে। অর্থাৎ প্রথম অংশে যা থাকবে পরের অংশেও কেবল তাই থাকলেই ম্যাচ সত্য হবে।

আউটপুট,

```
Match 1
Match 2
```

তৃতীয় ম্যাচ স্টেটমেন্টের ক্ষেত্রে `abc` এবং `cde` এক নয়। তাই এটি মিথ্যা হয়েছে।

`\d` `\s` এবং `\w`

`\d` দিয়ে ডিজিট, `\s` দিয়ে হোয়াইট স্পেস এবং `\w` দিয়ে ওয়ার্ড ক্যারেক্টার ম্যাচ করা হয়ে থাকে। ASCII মুডে এগুলো যথাক্রমে এভাবেও লেখা যায় বা একই এক্সপ্রেশন প্রকাশ করে - [0-9], [\t\n\r\f\v], and [a-zA-Z0-9_]

মজার বিষয় হচ্ছে এই স্পেশাল সিকোয়েন্স গুলোর বড় হাতের ভার্সন ঠিক উল্টো জিনিষ প্রকাশ করে। অর্থাৎ - `\D` দিয়ে সব কিছু কিন্তু ডিজিট নয় এমন ম্যাচ করে।

উদাহরণ,

```
import re

pattern = r"(\D+\d)"

match = re.match(pattern, "Hi 999!")

if match:
    print("Match 1")

match = re.match(pattern, "1, 23, 456!")
if match:
    print("Match 2")

match = re.match(pattern, " ! $?")
if match:
    print("Match 3")
```

আউটপুট,

```
Match 1
```

অর্থাৎ, (\D+\d) দিয়ে এমন একটি স্ট্রিং কে ম্যাচ করার কথা বলা হচ্ছে যার শুরুতে কিছু ক্যারেক্টার থাকবে যেগুলো আর যাই হোক ডিজিট নয়, এবং এরপরে কিছু ডিজিট থাকবে।

আরও কিছু স্পেশাল সিকোয়েন্স

`\A` এবং `\Z` দিয়ে কোন স্ট্রিং এর শুরু এবং শেষ ম্যাচ করা হয়। `\b` দিয়ে ওয়ার্ডের মধ্যেকার বাউন্ডারি বা সীমা গুলোকে চিহ্নিত করা হয়। নিচের উদাহরণটি দেখলে এর ব্যবহার আরও পরিষ্কার বোঝা যাবে।

```python
import re

pattern = r"\b(cat)\b"

match = re.search(pattern, "The cat sat!")
if match:
    print ("Match 1")

match = re.search(pattern, "We s>cat<tered?")
if match:
    print ("Match 2")

match = re.search(pattern, "We scattered.")
if match:
    print ("Match 3")

match = re.search(pattern, "We/cat.tered.")
if match:
    print ("Match 4")

match = re.search(pattern, "We{cat!tered.")
if match:
    print ("Match 5")
```

আউটপুট,

```
Match 1
Match 2
Match 4
Match 5
```

> এখানে `\b(cat)\b` এর মাধ্যমে `cat` শব্দটিকে ম্যাচ করা হচ্ছে যাতে এর দুপাশে যেকোনো রকম ওয়ার্ড বাউন্ডারি থাকে। তাহলেই ম্যাচ সত্য হবে।

কিছু সচারচর জিজ্ঞাসা

এই সেকশনে থাকছে বেশ কিছু গুরুত্বপূর্ণ কিন্তু কম আলোচিত বিষয় এবং নতুনদের জন্য উপকারী কিছু টপিক নিয়ে আলোচনা।

- পাইথনিকনেস
- PEP
- **main**
- *# -- coding: utf-8 --*
- #! /usr/bin/env python
- CPython
- ডকুমেন্টেশন পড়া

> এখানে আরও কিছু বিষয় নিয়ে পরবর্তীতে আলোচনা করা হবে।

পাইথনিকনেস

একজন ভালো পাইথন প্রোগ্রামার হতে হলে একটা জিনিষ মাথায় রাখা উচিৎ। আর তা হচ্ছে - যেকোনো প্রোগ্রাম তৈরির সময় খেয়াল রাখতে হবে যে, ওই প্রোগ্রাম তৈরির সময় সেটাকে দিয়ে যে কাজ করিয়ে নেয়ার প্ল্যান ছিল, সেই প্রোগ্রাম যেন ঠিক ঠাক ভাবে ঠিক ওই কাজটি-ই করে। আরও, খেয়াল রাখতে হয় কোড লেখার সময় যেন কোড গুলো ক্লিন হয় এবং যেকোনো মানুষের দারা বুঝতে সুবিধা হয়। এমনকি নিজের লেখা কোডও অনেক সময় কিছুদিন পর নিজেই পড়ে বোঝা যায় না। সেরকম যাতে না হয় তাই, কোড লেখার সময় শত ব্যস্ততার মধ্যেও একটু গুছিয়ে এবং পরিষ্কারভাবেই লেখা উচিৎ। এতে করে পরে এর সুবিধা পাওয়া যাবেই।

তো, এরকম অনেক গুলো নিয়ম বা দর্শন আছে যেগুলো ফলো করলে ভালো একজন পাইথন প্রোগ্রামার হবার পথে আসলেই উপকার পাওয়া যাবে। একে বলে Zen of Python. এর আওতাভুক্ত সেই সুন্দর নিয়ম বা টিপ্স গুলো খুব সহজেই দেখা যাবে যখন তখন। এর জন্য পাইথন কনসোল রান করে নিচের একটিমাত্র স্টেটমেন্ট এক্সিকিউট করলেই হবে -

```python
import this
```

উপড়ের স্টেটমেন্টটি লিখে এন্টার চাপলেই স্ক্রিনে নিচের মত আউটপুট আসবে -

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

আশা করা যায় প্রত্যেকটি লাইন পড়ে বোঝা যাবে তা দিয়ে প্রোগ্রামিং এর কন্টেক্সট থেকে কি বোঝানো হয়েছে।

> মজার বিষয় হচ্ছে, Zen of Python এ ২০ টি প্রিন্সিপাল এর উল্লেখ আছে কিন্তু এখানে আউটপুট আসে ১৯ লাইনের। ২০তম লাইনকে উহ্য রাখা হয়েছে। কেউ বলে থাকেন এটা যার যার নিজের মতকে গুরুত্ব দেয়ার জন্য, আবার কেউ বলে থাকেন হোয়াইট স্পেস ব্যবহারের কথাটাই এভাবে উপমা দিয়ে বোঝানো হয়েছে :)

PEP

এর পূর্ণ অর্থ হচ্ছে, Python Enhancement Proposals অর্থাৎ অভিজ্ঞ পাইথন প্রোগ্রামারদের পক্ষ থেকে কিছু প্রস্তাবনা যা পাইথনকে আরও বেশি শক্তিশালী, গোছানো, ফিচারফুল এবং ইফিসিয়েন্ট করতে সাহায্য করে।

যেমন, PEP 8 হচ্ছে রিডেবল পাইথন কোড লেখার ব্যাপারে একটি স্টাইল গাইড। এখানে বেশ কিছু গাইডলাইন আছে যেগুলো সব পাইথন প্রোগ্রামারের মেনে চলা উচিৎ। উদাহরণ সরূপঃ

- মডিউলের নাম হতে হবে সব ছোট হাতের অক্ষর দিয়ে এবং এর নাম ছোট হওয়া বাঞ্ছনীয়।
- ক্লাস এর নাম হওয়া উচিৎ ক্যাপ-ওয়ার্ড তথা CapitalWords স্টাইলে
- ভ্যারিয়েবল এবং ফাংশন এর নামও হওয়া উচিৎ ছোট হাতের অক্ষর দিয়ে এবং প্রয়োজনে আন্ডার স্কোর ব্যবহার করে, যেমন - my_function
- কন্সট্যান্ট এর নাম হওয়া উচিৎ বড় হাতের অক্ষর দিয়ে
- অপারেটর এর দু পাশে এবং প্রত্যেকটি কমা চিহ্নের পরে স্পেস ব্যবহার করা উচিৎ
- কোন লাইক ৮০ ক্যারেক্টারের বেশি লম্বা হওয়া উচিৎ নয়
- `from module import *` এরকম ইম্পোরট করা ঠিক নয়। নির্দিষ্ট করে শুধুমাত্র দরকারি ফাংশনকেই ইম্পোরট করা উচিৎ
- ঈন্ডেন্টেশনের জন্য ট্যাবের বদলে স্পেস (যেমন ৪টি) ব্যবহার করা উচিৎ

ইত্যাদি ...

> এরকম আরও কিছু PEP আছে যেমন - PEP 20, PEP 257 যেগুলোতে নির্দিষ্ট কিছু কাজের জন্য গাইডলাইন উল্লেখ আছে।

`__main__` কি

`if __name__ == "__main__":` এই লাইনটা অনেক পাইথন ফাইলে দেখে ঘাবড়ে যাবার কিছু নাই। আমরা বোঝার চেষ্টা করি কেন লোকজন এই অদ্ভুত if কন্ডিশনটাকে তাদের পাইথন স্ক্রিপ্টে লিখে। পাইথন ইন্টারপ্রেটার একটি প্রোগ্রামের এক্সিকিউশনের আগে যখন একটি সোর্স ফাইলকে পার্স (পড়ে) করে তখন সে এর জন্য কিছু স্পেশাল ভ্যারিয়েবল সেট করে। যখন স্বাধীনভাবে কোন পাইথন স্ক্রিপ্টকে রান করানো হয় তখন স্বয়ংক্রিয়ভাবে এর জন্য একটি `__name__` নামের ভ্যারিয়েবল তৈরি হয় যার ভ্যালু সেট করা হয় স্ট্রিং `"__main__"` . তাই আমরা যদি চাই আমাদের স্ক্রিপ্ট এর মধ্যেকার কিছু স্টেটমেন্ট শুধুমাত্র তখনি কাজ করুক যখন এটা একটা স্ট্যান্ডঅ্যালোণ স্ক্রিপ্ট হিসেবে রান করবে তখন বুদ্ধি করে এরকম একটা `if` কন্ডিশন লিখে তার মধ্যে ওগুলো লিখবো। কারন আমরা তো জানিই যে এই `if` কন্ডিশন তখনি সত্য হবে যখন এই স্ক্রিপ্টটা শুধু স্ক্রিপ্ট হিসেবেই রান হবে। যেমন আমাদের যদি নিচের মত একটা প্রোগ্রাম থাকে `Nuhil.py` ফাইলে,

```
def my_module_func():
    print("Nuhil Mehdy")

if __name__ == "__main__":
    print("Nuhil")
```

তাহলে আমরা যখন `python Nuhil.py` এভাবে একে রান করাবো তখন আউটপুট আসবে,

`Nuhil`

কিন্তু যদি এই `Nuhil.py` কে আরেকটি পাইথন ফাইল যেমন `Mehdy.py` এর মধ্যে মডিউল হিসেবে `import` করি তখন কিন্তু পাইথন এই `Nuhil.py` ফাইল পার্স (পড়ার সময়) করার সময় `__name__` নামের ভ্যারিয়েবলের জন্য `"__main__"` ভ্যালু সেট করবে না। আর তাই `if` কন্ডিশনটা মিথ্যা হবে। তো, `Mehdy.py` এর কোড যদি হয় নিচের মত,

```
import Nuhil
Nuhil.my_module_func()
```

তাহলে আউটপুট আসবে,

```
Nuhil Mehdy
```

এবং সুন্দর মত `print("Nuhil")` স্টেটমেন্টটি গোপনেই থেকে যাবে।

`# -*- coding: utf-8 -*-` কি

`# -*- coding: utf-8 -*-` এই কমেন্ট লাইনটাকে অনেক পাইথন স্ক্রিপ্টের শুরুতেই দেখা যায়। একটু একটু বোঝা যায় যে এখানে এনকোডিং ডিফাইন করে দেয়া হয়েছে। কিন্তু কিসের এবং কেন? কার জন্যই বা দরকার এটা? প্রথমেই জানতে হবে কম্পিউটারের সাথে টেক্সট নিয়ে কাজ করতে হলে তাকে টেক্সট গুলো চেনাতে হয়, আর আমরা সবাই জানি কম্পিউটার 1, 0 ছাড়া আর কাউকেই চেনে না। তো এনকোডিং এর সহজ মানে হচ্ছে একটা ম্যাপিং টেবিল যেখানে বলা আছে A এর মানে 01000001 (এটা ASCII এনকোডিং/টেবিল)। কিন্তু এই টেবিলে দুনিয়ার সব ভাষার সব ক্যারেক্টার এর সাপেক্ষে এরকম কম্পিউটার উপযোগী কোড নাই। এদিকে আমাদের প্রোগ্রাম লিখতে বা টেক্সট ফাইল লিখতে সেই ক্যারেক্টার গুলো লাগতেই পারে। তাই এরকম আরও একটা বিশাল লম্বা টেবিল আছে, utf-8 এনকোডিং/টেবিল। এই টেবিলে বাংলা "ক" সাপেক্ষেও একটা বাইনারি পাওয়ার ব্যবস্থা আছে। তো আমাদের পাইথন প্রোগ্রামে যদি এরকম ASCII এর বাইরের ক্যারেক্টার থাকে তাকে চেনাতে হলে ফাইলের শুরুতে বলে দিতে হবে যে - ভাই পাইথন তুমি আমার এই সোর্স ফাইলকে দয়া করে utf-8 টেবিল/এনকোডিং মোতাবেক পার্স করিয়ো নাহলে বুঝবা না আমি ফাইলে কি লিখছি :)

নিচের প্রোগ্রামটা যদি একটা `Test.py` ফাইলে লিখে পাইথন ২ দিয়ে রান করাই,

```
name = "নুহিন"
print(name)
```

তাহলে আউটপুট আসবে,

```
File "Test.py", line 3
SyntaxError: Non-ASCII character '\xe0' in file Test.py on line 3
```

তাই ওই `Test.py` কে আপডেট করে নিচের মত করতে হবে,

```
# -*- coding: utf-8 -*-
name = "নুহিন"
print(name)
```

এবার ঠিকঠাক রান করবে। আর হ্যা, ঠিক `# -*- coding: utf-8 -*-` এভাবেই যে সোর্স ফাইলের এনকোডিং ডিফাইন করতে হবে তাও কিন্তু না। `# Please encoding: utf-8` (সত্যি) এরকম লিখলেও পাইথন বুঝে যাবে :) শেষ কথা, এই ঝক্কি ঝামেলা কিন্তু Python 3 তে করতে হবে না কারন পাইথন ৩ ডিফল্ট এনকোডিং হিসেবে utf-8 কেই ধরে নেয় :D

`#! /usr/bin/env python` কি

অনেক পাইথন স্ক্রিপ্ট এর শুরুতেই এরকম একটা লাইন দেখতে পাওয়া যায়। আসলে এর মাধ্যমে নির্দিষ্ট করে এই স্ক্রিপ্টটির ডিফল্ট ইন্টারপ্রেটার কোনটা হবে সেটা ডিফাইন করে দেয়া হয়।
অর্থাৎ টার্মিনালে `chmod +x file.py` কমান্ড দিয়ে `file.py` কে এক্সিউকিউটেবল বানিয়ে অতঃপর `./file.py` কমান্ড দিয়ে সেটাকে এক্সিকিউট করতে চাইলে তার জন্য কোন ইন্টারপ্রেটার ব্যবহার হবে সেটা নির্ধারণ করে দেয়া হয়। এখানে Unix টাইপ সিস্টেমের ডিফল্ট পাইথনকে উক্ত স্ক্রিপ্ট বা ফাইলের ইন্টারপ্রেটার হিসেবে উল্লেখ্য করে দেয়া হচ্ছে। কিন্তু কিভাবে?

`env` হচ্ছে Unix সিস্টেমের মধ্যে থাকা একটা এক্সিকিউটেবল বাইনারি যে কিনা উক্ত সিস্টেমের সব এনভ্যায়রনমেন্ট ভ্যারিয়েবল গুলোকে খুঁজে নিতে পারে। তাই এর মাধ্যমে বস্তুত Python এর এক্সিকিউটেবল python -কে এই স্ক্রিপ্ট এর জন্য ব্যবহার করতে বলা হচ্ছে। যেহেতু এই লাইনটির উদ্দেশ্য হচ্ছে Python কে চিনিয়ে দেয়া। তাই নিচের মত করেও এটা ডিফাইন করা যায়।

```
#! /usr/bin/python
```

ধরে নিচ্ছি এটাই ওই সিস্টেমের python এর পাথ। কিন্তু এটা অ্যাবস্যুলেট পাথ। এক এক সিস্টেমে এক এক পাথে python থাকতে পারে। আমি আমার সিস্টেমে `which python` কমান্ড ইস্যু করে এটা পেয়েছি তাই এটা লিখলাম। এই অ্যাবস্যুলেট পাথের ঝামেলা মেটাতেই `/usr/bin/env` ব্যবহার করে python কে খুঁজে (যেখানেই থাকুক) সেটাকে ব্যবহার করতে বলা হয়।

এতক্ষণে বুঝে ফেলার কথা কিভাবে আমার এক্সিকিউটেবল পাইথন স্ক্রিপ্ট গুলোকে আমি Python 3 দিয়ে সবসময় রান করাতে পারবো। আমি স্ক্রিপ্টের শুরুতে নিচের মত করে Shebang লিখবো,

```
#! /usr/bin/env python3
```

CPython কি জিনিষ

Python প্রোগ্রাম বলতে আমরা যা ব্যবহার করি তা হচ্ছে CPython. এই, CPython হচ্ছে আসলে Python এর একটা ডিফল্ট এবং বহুল ব্যবহৃত ইমপ্লিমেন্টেশন।

এই ইমপ্লিমেন্টেশন জিনিষটা কি?

Python বা অন্য প্রোগ্রামিং ল্যাঙ্গুয়েজ গুলো হচ্ছে ফর্মাল ল্যাঙ্গুয়েজ অর্থাৎ একটু এদিক সেদিক করে এসব ভাষা ব্যবহার করলে কাজ হবে না। এটা বাংলা বা ইংলিশ এর মত ইনফর্মাল না যে - বানান বা ব্যাকরণগত ভুলে ভরা ভাষাতেও কিছু লিখলে বুঝে নেয়া যায়।

যাই হোক সিমপ্লি, ল্যাঙ্গুয়েজ রেফারেন্স হিসেবে চিন্তা করলে Python -কে একটা ইন্টারফেসও বলা যেতে পারে। যারা প্রোগ্রামিং করেন তারা এই টার্ম সম্পর্কে জানেন - ইন্টারফেস হচ্ছে এক ধরণের অ্যাবসট্র্যাকশন যার মাধ্যমে নির্ধারণ করে দেয়া হয় একটা নির্দিষ্ট কাজ কিভাবে হবে কিন্তু ইমপ্লিমেন্টেশনটা যার যেমন ইচ্ছা সেভাবে করবে। তাই, CPython বস্তুত Python এর Language Reference মেনে C দিয়ে করা একটা ইমপ্লিমেন্টেশন। আবার, CPython -ই হচ্ছে Python প্রোগ্রামের ইন্টারপ্রেটার :P

একটা পাইথন প্রোগ্রামের এক্সিকিউশনের ধাপ গুলো এরকমঃ

> Python Source Code (.py) -> Compiler -> Bytecode (.pyc) -> Interpreter (VM/CPython) -> Output (Hello World!)

যখন নির্দিষ্ট কোন কাজের সাপেক্ষে বলা হয় Python স্লো বা ফাস্ট তখন আসলে দোষ বা বাহবা যেটাই দেয়া হোক, দিতে হবে CPython কে অথবা ওই নির্দিষ্ট ইমপ্লিমেন্টেশনকে :)

Python এর এরকম আরও অনেক ইমপ্লিমেন্টেশন আছে। যেমন- Jython, IronPython, PyPy ইত্যাদি। Jython এর ক্ষেত্রেও বিষয়টা একই। অর্থাৎ - Python নামক ফর্মাল ল্যাঙ্গুয়েজের ইমপ্লিমেন্টেশন করা হয়েছে Java তে। তাই Python এর এই ভার্সনেও স্বাভাবিক Python এর সিনট্যাক্স মোতাবেকই প্রোগ্রাম লেখা যাবে এবং প্রোগ্রাম রান করালে এর পিছনে আসলে কলকাঠি নাড়বে Java.

অন্যান্য প্রোগ্রামিং ল্যাঙ্গুয়েজের ক্ষেত্রেও বিষয়টা এরকম। যেমন, যদি বলা হয়ঃ C++ is implemented in C. এর মানে সহজ ভাবে বলতে গেলেঃ C++ এর কম্পাইলার C দিয়ে তৈরি :)

আরেকটা কথা, উপরের এক্সিকিউশনের ধাপ অনুযায়ী Python কে ইন্টারপ্রেটেড বা কম্পাইল্ড ল্যাঙ্গুয়েজ কোনটাই বলা যাবে না। আসলে Python কিন্তু ইন্টারপ্রেটেড বা কম্পাইল্ড না। যদি এই বৈশিষ্ট্য কাউকে দিতেই হয় তাহলে CPython কে দিতে হবে।

CPython ইন্টারপ্রেটেড কিন্তু তার আগে একটা কম্পাইলেশন স্টেপ আছে :P যেটা [প্রোগ্রাম টু প্রসেসর] কনসেপ্ট অনুযায়ী এখানে অগ্রাহ্য)

কিভাবে ডকুমেন্টেশন বুঝতে হয়

অনেকেই অফিসিয়াল ডকুমেন্টেশন থেকে বিভিন্ন ফাংশনের Signature পড়ার সময় বুঝে উঠতে পারে না আসলে এর প্যারামিটার গুলো কি। যেমন - `round(number[, ndigits])`
https://docs.python.org/3/library/functions.html#round

এটাকে বলে ফাংশন সিগনেচার।

এর মাধ্যমে বুঝে নিতে হয় এই ফাংশনের প্যারামিটার গুলো কি এবং কেমন। যেমন এই ফাংশনের প্রথম প্যারামিটার 'number' অবশ্যই দিতে হবে অর্থাৎ Compulsory. আবার `[, ]` দিয়ে বোঝানো হয় এই প্যারামিটার গুলো Optional. আর তাই, `[ ]` এর বাইরে যেগুলো থাকবে সেগুলো অবশ্যই দিতে হবে আর ভিতরে যেগুলো থাকবে সেগুলো না দিলেও ওই ফাংশন কাজ করবে। কিন্তু,

`[ ]` এর মধ্যে থাকা অপশনাল প্যারামিটার গুলোর মধ্যে কিছু ডিপেন্ডেন্সিও থাকতে পারে। যেমন - `RegexObject.match(string[, pos[, endpos]])` এখানে বোঝানো হয়েছে 'endpos' পাস করলে অবশ্যই 'pos' -ও পাস করতে হবে। তাই এগুলো একটা নেস্টেড ব্র্যাকেটের মধ্যে নির্দেশ করা হয়েছে। আবার শুধু 'pos' দিয়ে 'endpos' না দিলেও চলবে।

প্যাকেজ

পাইথনে প্যাকেজিং বলতে বুঝায়, আপনার লেখা বিভিন্ন মডিউল গুলোকে একটা স্ট্যান্ডার্ড ফরম্যাটে একত্র করে বান্ডেল করা যাতে অন্য ইউজাররা খুব সহজেই আপনার বানানো প্রোগ্রামকে ব্যবহার করতে পারে। যেমন ধরুন, ওয়েব থেকে ডাটা স্ক্র্যাপ করার জন্য যাবতীয় সব রকম ফাংশনালিটি গুলো বানিয়ে কিছু ডেভেলপার একটি প্যাকেজ রিলিজ দিয়েছে যার নাম `BeautifulSoup` . আর আমরা খুব সহজেই সেই প্যাকজেটি আমাদের প্রজেক্টে ইন্সটল করে তার বিভিন্ন রেডিমেড ফাংশনকে ব্যবহার করতে পারি।

নিচে আমরা ধাপে ধাপে দেখবো কিভাবে একটি কাস্টম প্যাকেজকে PyPI তথা Python Package Index -এ আপলোড করতে হয়। ধরে নিচ্ছি, আমাদের প্যাকেজের সোর্স কোড github এ হোস্ট করা আছে এবং প্যাকেজটির নাম `mypackage` .

> প্রথমেই, PyPI Live এবং PyPI Test -এ আপনার একাউন্ট খুলে নিতে পারেন।

`.pypirc` কনফিগারেশন ফাইল তৈরি
এই ফাইলে PyPI এর সাথে আপনার অথেন্টিকেশনের কনফিগারেশন গুলো উল্লেখ থাকবে।

```
[distutils]
index-servers =
    pypi
    pypitest

[pypi]
repository=https://pypi.python.org/pypi
username=your_username
password=your_password

[pypitest]
repository=https://testpypi.python.org/pypi
username=your_username
password=your_password
```

কাজের সুবিধার জন্য এই ফাইলটিকে আপনার হোম ফোল্ডারে রাখতে পারেন। অর্থাৎ যার লোকেশন হবে `~/.pypirc`

> যেহেতু এই ফাইলে আপনার কিছু সংবেদনশীল তথ্য যেমন পাসওয়ার্ড থাকবে তাই এই ফাইলের পারমিশন বদলে নিতে পারেন - `chmod 600 ~/.pypirc` এই কমান্ড ইস্যু করে।

প্যাকেজ কন্টেন্ট তৈরি

ইউজারের জন্য এভাবে প্যাকেজ তৈরি করতে গিয়ে `setuptools` এবং `distutils` মডিউলের দরকার পরে। প্রথমেই একটি ডিরেক্টরির মধ্যে সব গুলো প্রোগ্রাম ফাইলকে রাখতে হবে। ওই একই ডিরেক্টরির মধ্যে `__init__.py` নামের একটি ফাইলও রাখতে হবে। এটা নিয়ম। এর উপস্থিতির মাধ্যমে পাইথন এটাকে একটা প্যাকেজ হিসেবে ধরে নেয়। এই ফাইলটি ফাকাও হতে পারে।

এরপর, আপনার প্রোগ্রাম ফাইলগুলো এবং এই স্পেশাল ইনিসিয়ালাইজার ফাইল সহ ডিরেক্টরিটিকে আরও একটি রুট ডিরেক্টরির মধ্যে রাখতে হবে। এবং সেই রুট ডিরেক্টরির মধ্যে একটি readme ফাইল, একটি লাইসেন্স ফাইল এবং গুরুত্বপূর্ণ একটি ফাইল `setup.py` কে রাখতে হবে।

ডিরেক্টরি লিস্টিং উদাহরণ,

```
MyPackage/ # Name of this dir can be anything
   LICENSE.txt
   README.md
   setup.py
   setup.cfg
   mypackage/ # actual package name
      __init__.py
      myfile.py
      anotherfile.py
```

`setup.py` ফাইলে কিছু গুরুত্বপূর্ণ তথ্য যুক্ত করতে হয় যাতে করে আপনার ডেভেলপ করা প্যাকেজটি PyPI তথা Python Package Index এ আপলোড করার উপযোগী হয়। এতে করে ইউজাররা pip টুল ইউজ করে খুব সহজেই আপনার প্যাকেজটি ইন্সটল করে নিতে পারবে।

ফাইলটি হতে পারে নিচের মত,

```
from distutils.core import setup

setup(
    name = 'mypackage',
    packages = ['mypackage'], # this must be the same as the name above
    version = '0.1',
    description = 'A random test lib',
    author = 'Nuhil Mehdy',
    author_email = 'nuhil@nuhil.net',
    url = 'https://github.com/nuhil/mypackage', # use the URL to the github repo
    download_url = 'https://github.com/nuhil/mypackage/archive/0.1.tar.gz', # I'll explai
    keywords = ['testing', 'logging', 'example'], # arbitrary keywords
    classifiers = [],
)
```

`download_url` লিঙ্কের মাধ্যমে আপনার প্যাকেজের টারবল ভার্সন পয়েন্ট করে দিতে হয়। যদি আপনার সোর্স কোড github -এ হোস্ট করা থাকে এবং মুল রিপোজিটরি থেকে একটি ট্যাগ তৈরি করা থাকে তাহলে সেখানে এরকম টারবল রেডি থাকে।

> ট্যাগ তৈরি করতেঃ প্রথমে লোকাল রিপোজিটরিতে ঢুকে কমান্ড দিতে হবে `git tag 0.1 -m "Adds a tag so that we can put this on PyPI."` এরপর `git tag` কমান্ড দিয়ে ট্যাগ গুলো দেখা যাবে। আগের কমান্ড দিয়ে ঠিক ঠাক ট্যাগ তৈরি হয়ে থাকলে লিস্ট আসবে এমন - `0.1` . এরপর `git push --tags origin master` কমান্ড দিতে হবে যার মাধ্যমে github এ এই ট্যাগের তথ্য যুক্ত হয়ে যাবে। আর সেই নির্দিষ্ট ট্যাগ ওয়ালা টারবলটী ডাউনলোডের জন্য তৈরি থাকবে এই লিঙ্কে - `https://github.com/{username}/{module_name}/archive/{tag}.tar.gz`

setup.cfg ফাইল

যদি আপনার readme ফাইলটি মার্কডাউন ফরম্যাটে লেখা হয়ে থাকে তাহলে এই ফাইলটি দরকার পরে। এর কন্টেন্ট হবে নিচের মত।

```
[metadata]
description-file = README.md
```

> `.txt` ফরম্যাটে লেখা readme ফাইল হলে উপরোক্ত ফাইল যুক্ত করার দরকার পরে না।

PyPI Test এ প্যাকেজ আপলোড

প্রথমে নিচের কমান্ড ইস্যু করুন -

```
python setup.py register -r pypitest
```

এরপর,

```
python setup.py sdist upload -r pypitest
```

সব কিছু ঠিক ঠাক থাকলে আপনার প্যাকেজকে Test PyPI Repository তে দেখা যাবে।

PyPI Live এ প্যাকেজ আপলোড

প্রথমে নিচের কমান্ড ইস্যু করুন -

```
python setup.py register -r pypi
```

এরপর,

```
python setup.py sdist upload -r pypi
```