HUNT পাইথন

দ্বিতীয় সংস্করণ

ইয়াসির আরাফাত রাতুল

HUNT পাইথন

ইয়াসির আরাফাত রাতুল

প্রকাশকঃ

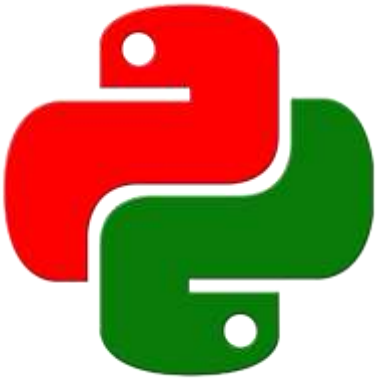
প্রচ্ছদঃ লেখক

পরিবেশকঃ

মূল্য

HUNT পাইথন

বাংলায় পাইথন প্রোগ্রামিং



ইয়াসির আরাফাত রাতুল

উৎসর্গ

আল্লাহ তা'আলার শ্রেষ্ঠ উপহার আব্বু-আম্মু কে

দৃষ্টি আকর্ষণ

বইটি লিখার ক্ষেত্রে বেশকিছু ইংরেজি শব্দ ব্যবহার করা হয়েছে। অনেক সময় বোঝানোর জন্য বাংলায় যথাযথ শব্দ খুঁজে না পাওয়ায় বা খুঁজে পেলেও সেটি খুবই বেমানান হওয়ায় পাঠকের বোঝার সুবিধার্থে ইংরেজি শব্দ ব্যবহার কেই বেশি প্রাধান্য দেয়া হয়েছে। যদি কোথাও কোন বাংলা শব্দের ভুল প্রয়োগ বা বানান ভুল পেয়ে থাকেন তার জন্য ক্ষমা প্রার্থী।

কোন ভুল পরিলক্ষিত হলে নির্দ্বিধায় জানাতে পারেন।

ইমেইলঃ arafatyeasir3@gmail.com

কাদের জন্য?

যারা অন্য একটি প্রোগ্রামিং ল্যাংগুয়েজের বেসিক জানেন এবং পাইথন শেখা শুরু করতে চাচ্ছেন বা যারা পাইথনের বেসিক জানেন এবং আরও বিশদ ভাবে পাইথন শিখতে চাচ্ছেন তারা এই বইটি পড়ে উপকৃত হবেন বলে আমার বিশ্বাস।

ভূমিকা

বর্তমান সময়ে পাইথন ল্যাঙ্গুয়েজের চাহিদা দিন দিন বেড়েই চলেছে। বিভিন্ন লাইব্রেরি দিয়ে সমৃদ্ধ এই ল্যাঙ্গুয়েজ শেখার জন্য বাংলা ভাষায় মান সম্মত বই এর বড়ই অভাব। সহজ ও সঠিক উপায়ে শেখার সেই চাহিদাই লেখক পূর্ণ করেছেন এই বইটির মাধ্যমে। পাইথনের গুরুত্বপূর্ণ বিষয়গুলো খুব সহজ ভাষায় এই বইটিতে লেখক আলোচনা করেছেন। নি:সন্দেহে যারা এই ল্যাঙ্গুয়েজে একেবারেই নতুন তাদের জন্য বইটি কার্যকারী ভূমিকা পালন করবে বলে আশা রাখি। পাইথনের কঠিন বিষয়গুলোকেও নতুনদের জন্য অনেক সহজভাবে উদাহরণ সহ লেখক তার নিজস্ব ঢঙ্গে উপস্থাপন করেছেন। আমি লেখক ইয়াসির আরাফাত রাতুলের সাফল্য কামনা করছি। আশা রাখছি ভবিষ্যতে প্রোগ্রামিং এর আরো অনেক বিষয় নিয়ে তিনি তার পাঠকদের সামনে হাজির হবেন। সবশেষে,আমি এই বইটির সার্বিক সাফল্য কামনা করছি ।

~ আতিফ উল আফতাব

গ্র্যাজুয়েট রিসার্চ অ্যাসিসট্যান্ট,

বায়োইনফরমেটিক্স ল্যাব

ইউনিভার্সিটি অফ ম্যানিটোবা,কানাডা

সূচি

এ ব্রীফ স্টোরি অফ পাইথন

পার্ট -১

এ ব্রিফ হিস্টোরি অফ পাইথন

গুইডো ভ্যান রসাম **Benevolent dictator for life** (**BDFL**)
১৯৮০'র শেষের দিকে পাইথন প্রোগ্রামিং ল্যাঙ্গুয়েজ তৈরি করেন। তখন তিনি একে "এবিসি" প্রোগামিং ল্যাঙ্গুয়েজ এর সাকসেসর বা পরবর্তী সংস্করণ হিসেবে প্রকাশ করেছিলেন।

পাইথন শুনতে যদিও মনে হয় এটি অজগরের নামে নামকরণ করা হয়েছে কিন্তু প্রকৃতপক্ষে গুইডো তার একজন প্রিয় কৌতুক অভিনেতা "মন্টি পাইথন" এর নামে এর নামকরণ করেন।

১৯৯১ সাল্ পর্যন্ত পাইথনকে অপ্রকাশিত রাখা হয় বিপুল সংখ্যক মডিউল ও প্যাকেজ সংযোজনের জন্য।

বর্তমানে পাইথনের দুটো ভার্সন প্রচলিত আছে ২.০ সিরিজ এবং ৩.০ সিরিজ (পাইথন ৩০০০ হিসেবে ও পরিচিত) ৩.০ সিরিজ ২.০ সিরিজ থেকে অনেক ক্ষেত্রেই অসামঞ্জস্যপূর্ণ, মূলত ৩.০ ইন্ট্রোডিউস করা হয় কিছু এক ঘেয়েমি কাজ পরিহার করতে এবং ক্লিন কোডিং এর জন্য।

পাইথন। "GUI" এপ্লিকেশন, গেম ডেভেলপমেন্ট, ওয়েব এপ্লিকেশন, ওয়েব স্ক্র্যাপিং, মলিকিউলার বায়োলজি, ডীপ লার্নিং ন্যাচারাল ল্যাঙ্গুয়েজ প্রোসেসিং, ডেটা সায়েন্স, মেশিন লার্নিং, আর্টিফিশিয়াল ইন্টিলিজেন্স সহ প্রায় সব ক্ষেত্রেই পাইথন প্রোগ্রামিং ল্যাঙ্গুয়েজ ব্যবহার করা হয় ।

পাইথনের "জি ইউ আই" টুলকিটের মধ্যে "**টারটল**", "**টিকইন্টার**", "**ডব্লিউ এক্স-পাইথন**" ইত্যাদি জনপ্রিয়। ওয়েব এপ ডেভেলপ মেন্টের জন্য **ফ্লাস্ক**, **জ্যাঙ্গো**, **পিরামিড, টর্নেডো** ইত্যাদি ওয়েব ফ্রেমওয়ার্ক বহুল ব্যবহৃত।

এছাড়া গেম ডেভেলপমেন্টের জন্য "**পাইগেম**", ওয়েব স্ক্র্যাপিং এর জন্য **"স্ক্র্যাপি" "বিউটিফুল সুপ"** ডীপ লার্নিং এর জন্য "**কেরাস**", ন্যাচারাল ল্যাঙ্গুয়রজ প্রোসেসিং এর জন্য "**এন এল টি কে**", মলিকিউলার বায়োলোজির জন্য "**বায়োপাইথন**", ডেটা সাইন্স, মেশিন লার্নিং এর জন্য "**টেন্সরফ্লো**","**পান্ডাস,নাম-পাই** র মত অনেক রকমের লাইব্রেরি,মডিউল,ফ্রেমওয়ার্কের বিশাল ভান্ডার রয়েছে।

অধ্যায়-১

পাইথন বেসিকস্

পাইথন একটি হাই-লেভেল, ইন্টারপ্রেটেড, ইন্টারেকটিভ, ডাইনামিক্যালি টাইপড এবং অব্জেক্ট ওরিয়েন্টেড প্রোগ্রামিং ল্যাংগুয়েজ। **ইন্টারপ্রেটেড ল্যাংগুয়েজ** মানে পাইথন প্রোগ্রাম রান করার জন্য প্রথমে কম্পাইল করতে হয় না। পাইথন রানটাইমে প্রোগ্রাম প্রোসেস করে। এ কারনে পাইথন অন্যান্য ল্যাংগুয়েজ থেকে কিছুটা ধীরগতির।
পাইথনের স্ট্যান্ডার্ড ফর্ম মূলত সি-ল্যাংগুয়েজের উপর ভিত্তি করে লিখা যাকে Cpython বলে। আমরা যখন পাইথনে কোন কোড লিখি ও রান করি ঠিক তখন পাইথন সে কোড কে বাইট কোডে কনভার্ট করে তারপর রান করে। এ জন্যই পাইথনের প্রোগ্রাম রান করতে সময় একটু বেশি লাগে। বর্তমানে Cpython ছাড়াও পাইথনের আরো কিছু ইমপ্লিমেন্টেশন আছে। যেমনঃ PyPy, Jython ইত্যাদি।

শুরুতে পাইথনের কিছু গাইডলাইন,এথিকস বা নীতি দেখে নেই যা "The Zen of Python" নামে পরিচিত। ইন্টারপ্রেটারে "*import this*" এই কমান্ড দিলে ২০ লাইনের একটা আউটপুট প্রিন্ট হয়।

```
>>> import this
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!
```

এখন প্রশ্ন থাকতে পারে যে এখানে তো ১৯ লাইন। আরেক লাইন কোথায়? এটা এখন পর্যন্ত অজানাই রয়ে গেছে, তবে অনেকের মতে অন্য লাইনটি হচ্ছে হোয়াইট স্পেস, যা দিয়ে পাইথনে স্পেসিং এর প্রয়োজনীয়তা বুঝানো হয়েছে।

ভ্যারিয়েবল (Variable)

ভ্যারিয়েবল হচ্ছে একটা পাত্র যেখানে আমরা কোন ডেটা রাখি। পাইথনে কোন ভ্যারিয়েবল ডিক্লেয়ার করার জন্য ডেটা টাইপ উল্লেখ করতে হয় না। পাইথন নিজেই কোন ডেটা টাইপের ভ্যারিয়েবল সেটা ডিফাইন করে নেয়।

```
a = 100          #An integer type variable
b = 253.64     #A float type variable
c = "Python" #A string type variable
```

পাইথনে একসাথে একাধিক ভ্যারিয়েবলও ডিক্লেয়ার করা যায়।

```
a, b, c = 100, 474.7, "Python"
```

কন্সট্যান্টস (Constants)

ভ্যারিয়েবল মান সবসময় পরিবর্তনীয়। কিন্তু আমরা যদি চাই যে কোন আইডেন্টিফাইয়ারের মান পরিবর্তন না হোক মানে ধ্রবক থাকুক তাহলে সে আইডেন্টিফাইয়ারের নামের প্রতিটি অক্ষর আপার কেজ লেটারে লিখতে হয়। তবে এটি একটি কনভেনশন মানে আপার কেজ লেটারে না লিখলে ও এরর দেখাবে না। পাইথন একে ভ্যারিয়েবল হিসেবেই নিবে। শুধু প্রোগ্রামকে রিডেবল করতে এই নিয়ম মানা জরুরি।

আইডেন্টিফায়ার (Identifier)

পাইথন আইডেন্টিফায়ার হচ্ছে ভ্যারিয়েবল, ক্লাস, ফাংশন ইত্যাদির জন্য ব্যবহৃত নাম। আইডেন্টি-ফায়ার লিখার ক্ষেত্রে আমরা a/A - z/Z, 1 - 9, এবং আন্ডারস্কোর " _ " ব্যবহার করি। পাইহনে আইডে ন্টিফায়ার লিখার ক্ষেত্রে @, $, % ইত্যাদি চিহ্ন ব্যবহার করা যায় না। পাশাপাশি পাইথন একটি কেজ সেন্সিটিভ ল্যাংগুয়েজ। python এবং Python দুটো আলাদা অর্থবহন করে।

পাইথন নেমিং কনভেনশন

নেমিং কনভেনশন হল, নাম রাখার নিয়ম। কোড কে রিডেবল করার জন্য নেমিং কনভেনশন মানা খুবই প্রয়োজন। পাইথনে ক্লিন কোড লিখার জন্য কিছু ভ্যারিয়েবল, ফাংশন্স ক্লাস ইত্যাদির নাম -করণের ক্ষেত্রে কিছু নিয়ম ফলো করে।

- গ্লোবাল অথবা লোকাল যেকোণ ভ্যারিয়েবল সব সময় লোয়ার কেজ লেটারে লিখতে হয়।

```
var = 0
```

- ফাংশনের নাম লোয়ার কেজ লেটারে এবং শব্দগুলোকে আন্ডারস্কোর দিয়ে আলাদা করে লিখতে হয়।

```
def func_name():
    pass
```

- প্যাকেজ ও মডিউলের নাম লোয়ার কেজ লেটারে লিখতে হয়।

```
math, os, sys
```

- ক্লাসের নাম আপারকেজ লেটার দিয়ে শুরু করতে হয়। এছাড়া বাকি সব আইডেন্টফায়ার লোয়ারকেজ লেটার দিয়ে শুরু করতে হয়।

```
class Car:
    pass
```

- কোন আইডেন্টিফায়ার যদি দুটি আন্ডারস্কোর দিয়ে শুরু ও শেষ হয় তার মানে হচ্ছে আইডেন্টিফায়ার টি একটা ল্যাংগুয়েজ ডিফাইন্ড স্পেশাল নেম।

```
__dict__, __future__, __all__
```

- নন-পাবলিক মেথডের আগে একটি আন্ডারস্কোর দিতে হবে।

```
def _protected():
```

```
pass
```

- প্রাইভেট মেথডের শুরুতে দুটি আন্ডারস্কোর দিতে হয়। যদিও এটি কনভেনশন নয় এটি সিন্ট্যাক্স।

```
def __private():
    pass
```

- কোন ফাংশনের নামের আগে ও পরে দুটিকরে আন্ডারস্কোর থাকলে ফাংশনটি একটি ম্যাজিক মেথড।

```
def __init__():
    pass
```

ইন্ডেন্টেশন (Indentation)

ইন্ডেন্টেশন হচ্ছে নির্দিষ্ট সংখ্যক স্পেস। সাধারণত এক ট্যাব বা চারটি স্পেস কে স্ট্যান্ডার্ড ধরা হয়। তবে এটি ভ্যারিয়েবল, আমরা চাইলে এর মান পরিবর্তন করে দিতে পারি । পাইথনে স্পেসিং বা ইন্ডেন্টেশন অনেক বেশি গুরুত্বপূর্ণ। কারণ পাইথনে কোড ব্লক বুঝানোর জন্য কোন ব্র্যাকেট ব্যবহার করা হয় না। তাই ক্লাস বা ফাংশন লিখার সময় কোড ব্লকের শুরু বা শেষ বুঝাতে ইন্ডেন্টেশনের ব্যবহার করা হয়। ইন্ডেন্টেশনের জন্য ট্যাবের বিপরীতে স্পেস ব্যবহার করাই উচিত।

```
def func():
__pass
```

indentation ⏎

কমেন্টস (Comments)

প্রোগ্রামিং এ কমেন্টিং এর অভ্যাস গড়ে তোলা খুবই গুরুত্বপূর্ণ। কোডকে রিডেবল করার জন্য কমেন্টিং করা খুবই জরুরি। তা না হলে দেখা যায় কিছু দিন পর নিজের লিখা কোড নিজেই বুঝতে বেগ পেতে হয়। আর কমেন্ট লিখার সময় ও খেয়াল রাখতে হবে যে কমেন্ট যেন স্পষ্ট হয়। একটা বিখ্যাত উক্তি আছে "Code never lies, comments sometimes do". পাইথনে কমেন্টিং এর জন্য (#) হ্যাস চিহ্ন ব্যবহার করা হয়।

```
# This is a comment
```

ডক্ স্ট্রিং

একাধিক লাইন কমেন্ট লিখার ক্ষেত্রে পাইথনে ট্রিপল সিঙ্গেল কোট (''' ''') অথবা ট্রিপল ডাবল কোট (""" """) ব্যবহার করা হয়। এদেরকে ডকস্ট্রিং বলা হয়। কমেন্টস ও ডক স্ট্রিং এর মাঝে অনেক পার্থক্য আছে।

```
"""
If I really hate pressing `enter` and
typing all those hash marks, I could
just do this instead
"""
```

কোন ফাংশন,ক্লাস বা প্যাকেজের ডকুমেন্টেশন লিখার জন্য ডকস্ট্রিং ব্যবহার করা হয়। কোন ফাংশন, ক্লাস বা প্যাকেজের ডকুমেন্টেশ্ন দেখার জন্য বিল্ট-ইন help() ও __doc__ ফাংশন ব্যবহার করতে পারি।

```
>>> def func():
        '''this is the function documentation'''
        pass

>>> print(func.__doc__)
this is the function documentation
>>> help(func)
Help on function func in module __main__:

func()
    this is the function documentation
```

রিজার্ভড ওয়ার্ডস / কি ওয়ার্ডস (key-words)

যে শব্দ গুলো পাইথনে আগে থেকেই কোন না কোন কাজের জন্য ডিফাইন করা আছে সেগুলো হচ্ছে কি-ওয়ার্ড বা রিজার্ভড ওয়ার্ডস। কোন প্রোগাম লিখার সময় খেয়াল রাখতে হবে যাতে কোন আইডেন্টিফায়ারের নাম আর রিজার্ভড ওয়ার্ডসের নাম একই না হয়।

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

Some Keywords of Python

ইনপুট-আউটপুট (input-output)

ইনপুট input()

ইউজার থেকে কোন ডেটা ইনপুট নিতে পাইথনে input() ফাংশন ব্যবহার করা হয়।

```
a = input()
b = input("Enter a number: ")
```

উপরের স্টেট্‌মেন্টের ক্ষেত্রে আমরা যদি নাম্বার ও ইনপুট দেই পাইথন সেটা **স্ট্রিং** হিসেবেই নিবে।

```
>>> a = input()
4
>>> a
'4'
>>> type(a)
```

```
<class 'str'>
```

তাই ইনপুট নেয়ার ক্ষেত্রে বলে দিতে হয় আমরা কোন টাইপের ডেটা ইনপুট হিসেবে চাই। প্রকৃত পক্ষে আমরা স্ট্রিং ই ইনপুট নেই এবং তাকে int(),str(),float() ফাংশনের মধ্যে পাঠিয়ে এর টাইপ পরিবর্তন করে নেই।

```
>>> a = int(input())
6
>>> a
6
>>> type(a)
<class 'int'>
>>> a = float(input())
2
>>> a
2.0
```

আউটপুট print()

পাইথনে কোন আউটপুট দেখানোর জন্য print() ফাংশন ব্যবহার করা হয়। প্রিন্ট ফাংশনের ফুল সিন্ট্যাক্স হচ্ছেঃ

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

প্যারামিটারসঃ

- objects – যে কোন অবজেক্ট (*) এর দেয়ার মানে হচ্ছে একাধিক অবজেক্ট।
- sep – একটি আউট পুট অপরটির মাঝখানে কি চিহ্ন দিয়ে আলাদা করা হবে। ডিফল্ট ভ্যালূঃ ' '(সিঙ্গেল স্পেস)

- end - end এর ভেতর আমরা যা পাস করব সেটা একদম সবার শেষে প্রিন্ট হবে।
- file – ফাইল অবজেক্ট নেয়।
- flush - False/True

নোট: sep, end, file এবং flush হল কি-ওয়ার্ড আর্গুমেন্টস। অর্থ্যাৎ, এই আর্গুমেন্টস গুলো ব্যবহার করার সময় এদের নাম বলে দিতে হবে।

```
print(*objects, sep = 'separator')
```

এভাবে দিলে ভূল হবেঃ

```
print(*objects, 'separator')
```

বিভিন্ন ধরনের আর্গুমেন্টস নিয়ে আমরা এই বইয়ে পরে আলোচনা করব।

উদাহরণঃ

```
a = 3
print("a =", a, sep=’ *****’, end='\n\n\n')
print("a =", a, sep='---', end='end')
```

আউটপুটঃ

```
a = *****3

a =---3end
```

flush কি-ওয়ার্ড আর্গুমেন্ট কিভাবে কাজ করে তা নিয়ে এখানে আমরা বিশদ আলোচনা করব না। তবে এটির কাজটা একটু দেখা যাক…

```
txtfile = open("myfile.txt",'w')
for i in range(100):
    print("I love Python",sep=' ',end = ' ',file = txtfile, flush = True)
```

প্রথমে যে ডিরেক্টরি / ফোল্ডারে আছো সেখানে myfile.txt নামে একটি ফাইল তৈরি কর। open() ফাংশনে ফাইলে নাম দিয়েছি এবং বলেছি একে 'w' write মুডে ওপেন করতে। এসব আমরা "পাইথন ফাইলস" অধায়ে দেখব। এবার এই একই প্রোগ্রাম একবার flush = False দিয়ে রান কর এবং myfile.txt ফাইল টি ওপেন করে দেখ। এবার flush = True দিয়ে রান কর তারপর ফাইল টি ওপেন কর। কিছু পরিবর্তন আছে? এ পরিবর্তনের পেছনে কি কারণ তা খোঁজার চেষ্টা কর। আমি একটু হিন্টস দিয়ে দেইঃ এর সাথে বাফারের একটা সম্পর্ক আছে।

পাইথন অপারেটরস (Operators)

যোগ, বিয়োগ বা অন্য যেকোন অপারেশন চালানোর জন্য কিছু বিশেষ চিহ্ন ব্যবহার করা হয়, এদেরকেই অপারেটর বলা হয়।কোন অপারেশনে জন্য যেসব ভ্যারিয়েবল বা ভ্যালু ব্যবহার করে তাদের অপারেন্ড বলে।

a+b একটি অপারেশন যেখানে a এবং b হল দুটি অপারেন্ড। পাইথনে ছয় ধরনের অপারেটর রয়েছেঃ

১। এরিথমেটিক অপারেটর

যোগ বিয়োগ সহ অন্যান্য গাণিতিক অপারেশনের জন্য যেসব অপারেটর ব্যবহার করা হয় তাদের এরিথমেটিক অপারেটর বলে।

অপারেটর	এপ্লিকেশন
+	দুটি সংখ্যা যোগ করার জন্য
-	দুটি সংখ্যা বিয়োগ করার জন্য
*	দুটি সংখ্যা গুণ করার জন্য

/	দুটি সংখ্যা ভাগ করার জন্য(দশমিক সংখ্যায়)
//	দুটি সংখ্যা ভাগ করার জন্য (পূর্ণ সংখ্যায়)
%	দুটি সংখ্যার ভাগশেষ নির্ণয়ের জন্য

ঊদাহরণঃ

```
>>> a = 4
>>> b = 3
>>> a + b
7
>>> a - b
1
>>> a * b
12
>>> a / b
1.3333333333333333
>>> a // b
1
>>> a % b
1
```

২। রিলেশনাল অপারেটরঃ

দুটো অপারেন্ডেকে পরস্পরের সাথে তুলনা করতে যেসব অপারেটর ব্যবহার করা হয় তাদের রিলেশনাল অপারেটর বলে।

অপারেটর	এপ্লিকেশন

>	বাম পাশের অপারেন্ড ডান পাশের অপারেন্ডের চেয়ে বড় হলে True
<	বাম পাশের অপারেন্ড ডান পাশের অপারেন্ডের চেয়ে ছোট হলে True
==	দুটো অপারেন্ড সমান হলে True
!=	দুটো অপারেন্ড সমান না হলে True
>=	বাম পাশের অপারেন্ড ডান পাশের অপারেন্ডের চেয়ে বড় বা সমান হলে True
<=	বাম পাশের অপারেন্ড ডান পাশের অপারেন্ডের চেয়ে ছোট বা সমান হলে True

উদাহরণঃ

```
>>> a = 4
>>> b = 3
>>> a > b
True
>>> b < a
True
>>> a == b
False
>>> a <= b
False
>>> a >= b
True
>>> a != b
True
```

৩। লজিকাল অপারেটরঃ

দুটো বা তার অধিক কন্ডিশন চেক করার জন্য লজিকাল অপারেটর ব্যবহার করা হয়।

অপারেটর	এপ্লিকেশন
and	True যখন সব কয়টি কন্ডিশন বা অপারেন্ড সত্য।
or	True যখন যে কোন একটি কন্ডিশন বা অপারেন্ড সত্য।
not	কন্ডিশন বা অপারেন্ড True হলে False কন্ডিশন বা অপারেন্ড False হলে True

উদারহণঃ

```
>>> a = 4
>>> b = 3
>>> c = 1
>>> a > b and b > c
True
>>> a > b or c < b
True
>>> not a > b
False
```

৪। বিট-ওয়াইজ অপারেটরঃ

বিট-ওয়াইজ অপারেটর প্রথমে অপারেন্ড কে বাইনারি ভ্যালু তে কনভার্ট করে। তাই এদের বাইনারি অপারেটর ও বলে।

অপারেটর	এপ্লিকেশন
& বিটওয়াইজ AND	দুটি অপারেন্ডের বাইনারি ভ্যালু হতে কমন বিট রিটার্ণ করে।
। বিটওয়াইজ OR	দুটি অপারেন্ডের বাইনারি ভ্যালু হতে কমন আন-কমন সব বিট রিটার্ণ করে।
~ বিটওয়াইজ NOT	অপারেন্ডের ১'র পরিপূরক রিটার্ণ করে,অর্থাৎ প্রতিটা বিটের বিপরীত বিট রিটার্ণ করে
^ বিটওয়াইজ XOR	দুটি অপারেন্ডের বাইনারি ভ্যালু হতে কেবল মাত্র সেসব বিট রিটার্ণ করে যে গুলো আন-কমন।
>> বিটওয়াইজ রাইট শিফট	অপারেন্ডের প্রতিটা বিটকে নির্দিষ্ট বিট ডানে সরিয়ে দেয়।
<< বিটওয়াইজ লেফট শিফট	অপারেন্ডের প্রতিটা বিটকে নির্দিষ্ট বিট বামে সরিয়ে দেয়।

উদাহরণঃ

```
>>>a = 60            # 60 = 0011 1100
>>>b = 13            # 13 = 0000 1101
>>>c = 0

>>>c = a & b      # 12 = 0000 1100
12
>>>c = a | b       # 61 = 0011 1101
61

>>>c = a ^ b;      # 49 = 0011 0001
49
>>>
```

```
>>>c = ~a;         # -61 = 1100 0011
-61
>>>c = a << 2      # 240 = 1111 0000 every bit is moved two bits left
240
>>>c = a >> 2      # 15 = 0000 1111 every bit is moved two bits right
15
```

৫। ইন প্লেইস বা এসাইনমেন্ট অপারেটর

কোন ভ্যারিয়েবলে ডেটা রাখতে ইনপ্লেস অপারেটর ব্যবহার করা।

অপারেটর	এপ্লিকেশন
=	ভ্যালূ অ্যাসাইন করার জন্য
+=	বাম পাশের অপারেন্ডের সাথে ডান পাশের অপারেন্ড যোগ কর এবং বাম পাশের অপারেন্ডে সেই ভ্যালু অ্যাসাইন কর।
-=	বাম পাশের অপারেন্ডের হতে ডান পাশের অপারেন্ড বিয়োগ কর এবং বাম পাশের অপারেন্ডে সেই ভ্যালু অ্যাসাইন কর।
*=	বাম পাশের অপারেন্ডের সাথে ডান পাশের অপারেন্ড গুণ কর এবং বাম পাশের অপারেন্ডে সেই ভ্যালু অ্যাসাইন কর।
/=	বাম পাশের অপারেন্ডকে ডান পাশের অপারেন্ড দিয়ে ভাগ কর(দশমিক সংখ্যা) এবং বাম পাশের অপারেন্ডে সেই ভ্যালু অ্যাসাইন কর।
%=	বাম পাশের অপারেন্ডকে ডান পাশের অপারেন্ড দিয়ে ভাগ করে ভাগশেষ বাম পাশের অপারেন্ডে অ্যাসাইন কর।

//=	বাম পাশের অপারেন্ডকে ডান পাশের অপারেন্ড দিয়ে ভাগ কর(পূর্ণ সংখ্যা) এবং বাম পাশের অপারেন্ডে সেই ভ্যালু অ্যাসাইন কর।
**=	ডান পাশের অপারেন্ডকে বাম পাশের অপারেন্ডের এক্সপোনেন্ট(পাওয়ার) হিসেবে বসিয়ে মান বের কর এবং বাম পাশের অপারেন্ডে সেই ভ্যালু অ্যাসাইন কর।
&=	বিট-ওয়াইজ AND বের করে বাম পাশের অপারেন্ডে ভ্যালু অ্যাসাইন কর।
^=	বিট-ওয়াইজ XOR বের করে বাম পাশের অপারেন্ডে ভ্যালু অ্যাসাইন কর।
>>=	বিট-ওয়াইজ right shift বের করে বাম পাশের অপারেন্ডে ভ্যালু অ্যাসাইন কর।
<<=	বিট-ওয়াইজ left shift বের করে বাম পাশের অপারেন্ডে ভ্যালু অ্যাসাইন কর।
|=	বিট-ওয়াইজ OR বের করে বাম পাশের অপারেন্ডে ভ্যালু অ্যাসাইন কর।

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

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

আউটপুট,

Python3

৬। আইডেন্টিটি অপারেটসঃ

is = যদি দুটো অপারেন্ড মেমোরি লোকেশন একই হয় তাহলে True

is not = যদি দুটো অপারেন্ড মেমোরি লোকেশন একই না হয় তাহলে True

```
>>> a = 4
>>> b = 4
>>> a is b
True
#WHY?
>>> id(a)
1612671840
>>> id(b)
1612671840
>>> a = ['a','b','c']
>>> b = ['a','b','c']
>>> a is b
False
#WHY!!!
>>> id(a)
2260705602248
>>> id(b)
2260705673544
#because list is mutable and they are located in different part of the memory
```

৭। মেম্বারশিপ অপারেটরসঃ

in = কোন ভ্যালু, ভ্যারিয়েবল যদি অন্য কোন ভ্যারিয়েবল বা অবজেক্টে থাকে তাহলে True

not in = কোন ভ্যালু, ভ্যারিয়েবল যদি অন্য কোন ভ্যারিয়েবল বা অবজেক্টে না থাকে তাহলে True

```
>>> 'a' in a
True
>>> 'a' not in a
False
```

অধ্যায়-২

ডেটা টাইপ্স (Data Types)

কোন ডেটার ধরন নির্ণয় করার জন্য প্রোগ্রামিং এ বিভিন্ন ডেটা টাইপ্স নির্ধারণ করা হয় থাকে। পাইথনে ও এমন কয়েক ধরনের ডেটা টাইপ্স আছে ।

ক। ইন্টিগ্রাল টাইপ্স

পাইথনে দু'ধরনের বিল্ট-ইন ইন্টিগ্রাল ডেটা টাইপ্স আছে। int(Integer) এবং bool(Boolean) এরা প্রত্যেকেই **ইমিউটেবল।**

ক.১। ইন্টিজার (int)

ইন্টিজার বা নাম্বার ডেটা টাইপ যেকোন ধরনের পূর্ণসংখ্যা। ইন্টিজার ভ্যালু কনভার্সনের জন্য পাইথনে কিছু বিল্ট-ইন ফাংশন আছে।

bin(x) – যে কোন সংখ্যা কে বাইনারি তে কনভার্ট করে
hex(x) – যে কোন ইন্টিজার কে হেক্সা ডেসিমালে তে কনভার্ট করে
int(x) – যে কোন ইন্টিজার কে ইন্টিজারে তে কনভার্ট করে
oct(x) – যে কোন ইন্টিজার কে অক্টালে তে কনভার্ট করে
pow(x,y) – x to the power y
pow(x,y,z) – (x to the power y) % z

এছাড়াও আরও অনেক ফাংশন আছে, পাইথনের ডকুমেন্টেশন থেকে সেসবের ব্যবহার দেখে নিতে পার। মজার বিষয় হল এসব ফাংশন ছাড়া ও পাইথন ইন্টারপ্রেটারে আমরা ইন্টিজারের পাশাপাশি বাইনারি,অক্টাল ও হেক্সাডেসিমাল সংখ্যা ও লিখতে পারি…

```
>>> 15                  #integer
15
>>> 0b101010            #binary
42
>>> 0o172356           #octal
62702
```

```
>>> 0xABCD          #hexa-decimal
43981
>>>
```

এখানে লক্ষণীয় যে বাইনারি বুঝানোর জন্য 0b, অক্টাল বুঝানোর জন্য 0o এবং হেক্সাডসিমাল বুঝানোর জন্য 0x হেডিং ব্যবহার করতে হয়।

ক.২। বুলিয়ান (bool)

বুলিয়ান ডেটা টাইপ কোন এক্সপ্রেশনে সত্যতা যাচাই করে। বুলিয়ান ডেটা টাইপের দুটো ভ্যালু আছে True এবং False । True এর মান 1 এবং False এর মান 0. এবং একটি ইন্টিজার/ফ্লোট টাইপের ডেটার সাথে অপারেশন যোগ্য।

```
>>> a = True
>>> type(a)
<class 'bool'>
>>> 2 == 2
True
>>> 5<7
True
>>> 5>5
False
>>> 5+True
6
>>> 1.2 + True
2.2
>>> 7*False
0
>>> (5<7) == True
True
>>> 8 < 9 == True
False
```

শেষের কন্ডিশন টা কেন False হলো তা খুঁজে বের করা চেষ্টা করবে।

খ। ফ্লোটিং পয়েন্ট টাইপ্স

দশমিক সংখ্যাকে ফ্লোটিং নাম্বার বলে। পাইথনে তিন রকমের ফ্লোটিং পয়েন্ট নাম্বার আছে, বিল্ট-ইন float এবং complex. এবং অন্যটি পাইথন স্ট্যান্ডার্ড লাইব্রেরির decimal.Decimal টাইপ।

খ.১ ফ্লোট (float)

কোন দশমিক সংখ্যাকে বোঝানোর জন্য ফ্লোট ডেটা টাইপের ব্যবহার করা হয়। সাইন্টেফিক E নোটেশন বোঝানোর জন্য e বা E ব্যবহার করা যায়।

```
>>> type(a)
<class 'float'>
>>> float(4)
4.0
>>> .4e9
400000000.0
>>> .5e2
50.0
>>> 4.4e2
440.0
>>> 2.e0
2.0
>>> 2.e1
20.0
>>> 2.e
SyntaxError: invalid syntax
>>> 2e
SyntaxError: invalid syntax
```

খ.২ কমপ্লেক্স (complex)

কমপ্লেক্স নাম্বার একটি ইমিউটেবল ডেটা টাইপ। কমপ্লেক্স নাম্বারের দুটো ভাগ থাকে, বাস্তব সংখ্যা (real) ও কাল্পনিক সংখ্যা(imaginary).

```
>>> 3 + 2j
(3+2j)
>>> type(3+2j)
<class 'complex'>
>>> a = 2.3 - 5.8j
>>> a.real
2.3
>>> a.imag
5.8
>>> a + 4j
(2.3+9.8j)
```

কমপ্লেক্স নাম্বারের একটি ফাংশন হচ্ছে conjugate() যা ইমাজিনারি অংশের চিহ্ন পরিবর্তন করে।

```
>>> a.conjugate()
(2.3-5.8j)
```

খ.৩ ডেসিমাল (decimal)

ডেসিমাল ডেটা টাইপ পাইথনের কোন বিল্ট ইন ডেটা টাইপ না। এটি পাইথন স্ট্যান্ডার্ড লাইব্রেরি থেকে ইম্পোর্ট করতে হয়। মূলত গানিতিক হিসেবে সূক্ষাতিসূক্ষ্য ভুল এড়ানোর জন্য এ ডেটা টাইপ ব্যবহার করা হয়।

```
>>> from decimal import *
>>> a = Decimal(3.4)
>>> a
Decimal('3.39999999999999991118215802998747676610946655273437 5')
```

3.4 এর প্রকৃত মান হচ্ছে ডেসিমাল ভ্যালুটি। এ সংখ্যাটিকে নির্দিষ্ট অংক পর্যন্ত আউট পুট পেতে quantize () ফাংশন ব্যবহার করতে হবে।

```
>>> a.quantize(Decimal('0.00'))
Decimal('3.40')
```

```
>>> from decimal import *
>>> Decimal(3.4) + Decimal(4.3)
Decimal('7.69999999999999997335464740900')
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999,
Emax=999999, capitals=1, clamp=0, flags=[Inexact, FloatOperation,
Rounded], traps=[InvalidOperation, DivisionByZero, Overflow])
```

এখানে ডেসিমাল ডেটা টাইপ একটি বিশাল সংখ্যা রিটার্ণ করেছে। কিন্তু সবসময় এত বড় সংখ্যার প্রয়োজন হয় না । getcontext() ফাংশনের prec প্যারামিটার ব্যবহার করে আমরা নিজের ইচ্ছে মত সংখ্যা নির্ধারণ করে দিতে পারি। তবে এক্ষেত্রে দশমিকের আগের সংখ্যা ও prec এর ভ্যালু'র ভেতরে গণনা করা হয়।

```
>>> getcontext().prec = 5
>>> Decimal(3.4) + Decimal(4.3)
Decimal('7.7000')
>>> getcontext()
Context(prec=5, rounding=ROUND_HALF_EVEN, Emin=-999999,
Emax=999999, capitals=1, clamp=0, flags=[Inexact, FloatOperation,
Rounded], traps=[InvalidOperation, DivisionByZero, Overflow])
```

লক্ষ্য কর getcontext() ফাংশনের prec প্যারামিটারের মান পরিবর্তন হয়ে গেছে।

গ। স্ট্রিংস্ (Strings)

পাইথনে তিন নিয়মে স্ট্রিং তৈরি করা যায়। সিঙ্গেল,ডাবল অথবা ট্রিপল কোট ব্যবহার করে স্ট্রিং লিখা যায়।

```
>>> my_string = 'Hello World'
>>> my_string_two = "Welcome to Python"
>>>my_string_three = '''I am writing a multiple line string'''
```

মাল্টিপল লাইন স্ট্রিং লিখতে আমরা চাইলে তিনটি ডাবল কোট ও ব্যবহার করতে পারি।

```
>>> my_string_four = """ String using
                          three double quotes """
```

আমরা যদি চাই স্ট্রিং এর ভিতরেই সিঙ্গেল/ ডাবল কোট ব্যবহার করব তাহলে আমরা পুরো স্ট্রিং টি সিঙ্গেল কোটে লিখলে ভেতরে ডাবল কোট ব্যবহার করব এবং পুরো স্ট্রিং টি ডাবল কোটে লিখলে ভিতরে সিঙ্গেল কোট ব্যবহার করব।

```
>>> my_string = 'My name is"Ratul" ’
>>>my_string = "My name is 'Ratul' "
```

অথবা আমরা চাইলে ট্রিপল কোট ও ব্যবহার করতে পারি।

স্ট্রিং কনকেটেনেশন(Concatenation)

পাইথনে একাধিক স্ট্রিং কে যোগ করাকে স্ট্রিং কনকেটেনেটিং বলে। এর জন্য আমরা ' + ' অপারেটর ব্যবহার করি।

```
>>>name = "Tommy is my dog."
>>>color = "His color is Black"
>>>my_string = name + color
>>>my_string
Tommy is my dog. His color is Black
```

স্ট্রিং মেথডস

স্ট্রিং হচ্ছে এক ধরনের অবজেক্ট। পাইথনে স্ট্রিং এর জন্য কিছু বিল্ট ইন মেথড/ ফাংশন আছে।
যেমন আমরা যদি স্ট্রিং কে সম্পূর্ণ আপার/লোয়ার কেজ এ রূপান্তর করতে চাই তাহলে আমরা পাইথনে upper() বা lower() মেথড ব্যবহার করতে পারি।

আবার যদি শুধু মাত্র প্রথম অক্ষর ক্যাপিটালাইজ করতে চাই তখন আমরা capitalize() ফাংশন ব্যবহার করি। স্ট্রিং এর খুবই গুরুত্বপূর্ণ দুটি মেথড হলঃ

split() ও join()

split() ফাংশন কোন স্ট্রিং এর প্রতিটি হোয়াইট স্পেস পর পর অংশকে একটি করে এলিমেন্ট হিসেবে নিয়ে লিস্টে ইনপুট করে।
join() একটি স্ট্রিং মেথড যা আর্গুমেন্ট হিসবে একটি মাত্র ইটারেবল অবজেক্ট নেয় এবং একটি স্ট্রিং ক্লাসের অবজেক্ট রিটার্ন করে। এটি split() ফাংশনের বিপরীত কাজ করে।

```
>>>a = ['Programmers','are',"today's","Saints"]
>>>string = " ".join(a)
>>>string
Programmers are today's Saints
>>>string.split()
['Programmers','are',"today's",'Saints']
>>> string = "-".join(a)
["Programmers-are-today's-Saints"]
```

এছাড়াও আরও কিছু স্ট্রিং ফাংশন হল…

capitalize()	স্ট্রিং এর প্রথম অক্ষর ক্যাপিটাল লেটারে পরিবর্তন করে।
len()	স্ট্রিং এর দৈর্ঘ্য নির্ণয় করে।
upper()	পুরো স্ট্রিং কে আপার কেজ লেটারে পরিবর্তন করে।
lower()	পুরো স্ট্রিং কে লোয়ার কেজ লেটারে পরিবর্তন করে।
swapcase()	আপার কেজ লেটার কে লোয়ার কেজে ও লোয়ার কেজ লেটার কে আপার কেজে রূপান্তর করে।
strip()	কোন স্ট্রিং এর শুরুর ও শেষের সব হোয়াইট স্পেস রিমোভ করে দিতে এই ফাংশন ব্যবহার করা হয়।

```
>>> string = 'hunt Python'
>>> len(string)
11
>>> string.capitalize()
'Hunt python'
>>>
>>> string.upper()
'HUNT PYTHON'
>>>
>>> string.lower()
'hunt python'
>>>
>>> string = 'HuNt PyThOn'
>>> string.swapcase()
'hUnT pYtHoN'
>>>
>>> string = '   a b c   '
>>> string
'   a b c   '
>>> string.strip()
'a b c'
```

পাইথনে অনেক স্ট্রিং মেথড আছে। সব গুলো মেথড দেখতে হলে ইন্টারপ্রেটারে

>>>dir(my_string) কমান্ড টাইপ করলেই হবে।

স্ট্রিং স্লাইসিং

রিয়েল ওয়ার্ল্ড প্রবলেমের ক্ষেত্রে স্ট্রিং স্লাইসিং অনেক বেশি ব্যবহৃত হয়। স্লাইসিং করে স্ট্রিং এর প্রত্যেকটা ক্যারেক্টর একসেস করা যায়।

```
>>> my_string = "I like Python!"
```

যেহেতু পাইথন জিরো বেজড্ বা জিরো থেকে কাউন্টিং শুরু করে আমরা প্রথম এলিমেন্ট কে জিরো থেকেই কাউন্ট করব।

```
I [space] l i k e [space] p y t h o n !
0  1   2 3 4 5   6   7 8 9 10 11 12 13
```

```
>>>my_string [0:1]
'I'
>>>my_string [0:11]
'I like Pyth'

>>>my_string [0:14]
'I like Python!'

>>>my_string [2:14]
'like Python'

>>>my_string [ : ]
'I like Python’
```

(এখানে লক্ষণীয় যে আমরা ম্যাক্সিমাম লিমিট যত দিচ্ছি পাইথন ঠিক তার আগের সংখ্যা পর্যন্ত কাউন্ট করছে)

স্ট্রিং ফরম্যাটিং

"স্ট্রিং ফরম্যাটিং অর্থ হচ্ছে স্ট্রিং কে "সুবিন্যস্ত বা গুছিয়ে লিখা। একটা বেজ স্ট্রিং এ অন্য একটি স্ট্রিং/ ইন্টিজার/ ফ্লোট ভ্যালু ইনসার্ট করা।ধরুন আমার ৫ টি আপেল আছে, এখন আমি সেটাকে প্রোগ্রামিং এ কিভাবে লিখব?
my_apples = 5
এবং আমি চাই স্ক্রিনে প্রিন্ট করুক "Ratul has 5 apples"

এখন এই 5 তো ইন্টিজার ভ্যারিয়েবল,যখন আমি ইউজার থেকে একেক বার একেক ডেটা ইনপুট নিব তখন কিভাবে দেখাব? ৫ না হয়ে যদি ১০ হয় কিংবা আমার নামের জায়গায় যদি আপনার নাম হয়? তখন? বার বার তো আর কোড পরিবর্তন করা যাবে নাএই ধরনের সমস্যা এড়ানোর জন্যই মূলত স্ট্রিং ... ...ফরম্যাটিং ব্যবহার করা হয়
পাইথনে অনেক ভাবেই স্ট্রিং কে ফরম্যাট করা যায়। সবচেয়ে বেসিকপুরোনো - যে পদ্ধতি সেটাঅনেকটাসি' ই-'র মতঃ

```
name = "Ratul"
apples_num = 5
print(" %s has %i apples" %(name, apples_num))
আউটপুটঃ
Ratul has 5 apples.
```

এখানে খেয়াল কর যে আমরা যতগুলো স্ট্রিং ইনপুট দিচ্ছি ততগুলো %s ব্যবহার করছি।
কিন্তু আমরা যদি সমান সংখ্যক ইনপুট না দেই তাহলে আমাদের কি এরর দেখাবে?

```
print(" %s has %i apples" %(name))

Traceback (most recent call last):
File "<string>", line 1, in <fragment>
TypeError; not enough arguments for formattig string
```

এছাড়াও এই পদ্ধতির একটা সমস্যা হচ্ছে আমাদের আলাদা ডেটা টাইপের জন্য আলাদা আলাদা সিনট্যাক্স ব্যবহার করতে হয়...

সিঙ্গেল ক্যারেক্টারে জন্য %c
স্ট্রিং এর জন্য %s
ইন্টিজারের জন্য %i এরকম আরো অনেকগুলো আছে, তো এভাবে মনে রাখা কষ্টসাধ্য, কিন্তু পাইথনের কাজই হচ্ছে ব্যাপার গুলো সহজ থেকে সহজতর করা। সে কাজটি করে .format() মেথড...
এর মাধ্যমে একটি স্ট্রিং এর মধ্যে থাকা কিছু আর্গুমেন্টকে রিপ্লেস বা সাবস্টিটিউট করা যায়।

```
>>>name = “Ratul”
>>>apples_num= 5
>>>final_string = “{} has {} apples” .format(name, apples_num)
>>>print(final_string)
Ratul has 5 apples
```

সহজ করে বলি, আমরা final_string ভ্যারিয়েবলটির ভেতর {} কার্লিব্রেসি দ্বিতীয়) দিয়ে কিছু শূন্যস্থান তৈরি করে করেছি (বন্ধনী, এরপর এর শেষে ফরম্যাট ফাংশনের ভেতর বলে দিয়েছি যে কোন শুন্যস্থান টাকে কোন ভ্যারিয়েবলের মান দিয়ে পূরণ করতে হবে।
সেটা কিভাবে হল?
আমরা যখন একের পর এক শূণ্যস্থান তৈরি করি পাইথন তখন এদেরকে বাই - (০ থেকে শুরু)ডিফল্ট ইন্ডেক্সিং করে। তার মানে আমাদের প্রথম শূন্যস্থানের ইন্ডেক্স ভ্যালু হচ্ছে ০, দ্বিতীয় শূন্যস্থানের ইন্ডেক্স ভ্যালু হচ্ছে ১...
আগের প্রোগ্রামটা ই যদি একটু অন্যভাবে-রান করি তাহলে দেখো কি হয়।

```
>>>name = “Ratul”
>>>apples_num= 5
>>>final_string = “{0} has {1} apples”.format(name,apples_num)
>>>print(final_string)
Ratul has 5 apples
```

এইবার ও আউটপুট আগের মতই আসবে। এবার আমরা শূন্যস্থানে বলে দিয়েছি কোথায় ফরম্যাট ফাংশনের কোন আর্গুমেন্টের ভ্যালু বসবে মানে এই ইন্ডেক্স এর মান হচ্ছে ফরম্যাট ফাংশনের ইন্ডেক্স এর মান। চল কোডেই বোঝা যাক...

```
>>>name = "Ratul"
>>>apples_num= 5
>>>final_string = "{0} has {0} apples".format(name,apples_num)
>>>print(final_string)
```

এর আউটপুট আসবে...
Ratul has Ratul apples

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

```
>>>name = "Ratul"
>>>apples_num= 5
>>>final_string = "{1} has {0} apples and {0}
oranges".format(apples_num,name)
>>>print(final_string)
```

এর আউটপুট আসবে...
Ratul has 5 apples and 5 oranges

কিন্তু এই শেষ ও শেষ না। পাইথন ৩৬ এ পাইথন একটি নতুন স্ট্রিং ফরম্যাটিং . " লিটারেল নিয়ে আসে। যাএফ "স্ট্রিংস ফরম্যাটিং-বা "স্ট্রিং ইন্টারপোলেশন " নামেও পরিচিত। এবার আমরা শুধু নরমাল স্ট্রিং লিখে যেখানে প্রয়োজন সেখানে {} এর ভেতর ভ্যারিয়েবলের নাম বসিয়ে দিলেই কাজ হয়ে যাবে।

```
>>>name = "Ratul"
>>>apples_num= 5
>>>print( f"{name} has {apples_num} apples")
```

এবার ও ঠিক এই আউট পুট টাই পাব।
Ratul has 5 apples

ফরম্যাটিং এক্সপ্রেশন্সঃ

এক্সপ্রেশন	অর্থ	উদাহরণ
{:d}	ইন্টিজার ভ্যালু	'{:d}'.format(10.5) → '10'
{:.2f}	ফ্লোটিং পয়েন্ট নাম্বার, দশমিকের পর ২ সংখ্যা বিশিষ্ট।	'{:.2f}'.format(0.5) → '0.50'
{:.2s}	২ ক্যারেক্টার বিশিষ্ট স্ট্রিং	'{:.2s}'.format('Python') → 'Py'
{:<6s}	বাম পাশে ৬টি স্পেস বিশিষ্ট স্ট্রিং	'{:<6s}'.format('Py') → 'Py '
{:>6s}	ডান পাশে ৬টি স্পেস বিশিষ্ট স্ট্রিং	'{:<6s}'.format('Py') → ' Py'
{:^6s}	বাম-ডান দুই পাশে ৩টি করে স্পেস বিশিষ্ট স্ট্রিং	'{:<6s}'.format('Py') →' Py '

কালেকশন ডেটা টাইপ্স

আমরা যে ডেটা টাইপ্স গুলো নিয়ে আলোচনা করছি সেগুলো সিঙ্গেল ভ্যালু ডেটা নিতে পারত। কিন্তু আমরা যদি একটি ভ্যারিয়েবলে একই টাইপের বা ভিন্ন টাইপের ডেটা রাখতে চাই তখন আমাদের এক স্পেশাল ডেটা টাইপ প্রয়োজন হবে। এগুলো কে কালেকশন ডেটা টাইপ্স বলে। সি বা সি++ এ যেমন অ্যারে। এগুলোকে আবার ডেটা স্ট্রাকচার ও বলা হয়। কিন্তু পাইথনে এদের কে ডেটা টাইপ হিসেবেই ধরা হয়। যে নামেই ডাকি না কেন জিনিস তো একটাই। চল শুরু করা যাক...

ক.সিকোয়েন্স টাইপ্স

এই ধরনের ডেটা টাইপে একই বা ভিন্ন ধরনের ডেটা কমা দিয়ে আলাদা করে রাখা হয়। একের পর এক ডেটা রেখে এই ডেটা টাইপ গুলো সাজানো থাকে বলে এদের সিকোয়েন্সিয়াল ডেটা টাইপ বলে।

ক.১ লিস্ট (list)

লিস্ট হচ্ছে পাইথনের এক ধরনের ডেটা টাইপ যা এক সাথে একাধিক টাইপের ডেটা স্টোর করতে পারে। পাইথনে লিস্ট লিখতে স্কয়ার ব্রাকেট "[ ]" ব্যবহার করা হয় এবং এলিমেন্ট গুলা কমা সেপারেটেড থাকে।

```
<list_name> =[value1,value2,....valueN];
```

এখানে আমরা চাইলে ভ্যালু ওয়ানে স্ট্রিং, টু তে ইন্টিজার, এবং যেকোন পজিশনে যেকোন "ডেটা টাইপের" এলিমেন্ট ইনপুট দিতে পারব। লিস্ট একটি **মিউটেবল** ডেটা স্ট্রাকচার অর্থাৎ আমরা চাইলে লিস্টের ডেটা পরিবর্তন করতে পারব এবং এর জন্য পাইথন নতুন একটা লিস্ট তৈরি করে ফেলবে না।

লিস্টের ইন্ডেক্সিং শুরু হয় "০" শূন্য থেকে অর্থাৎ লিস্টে যদি n সংখ্যক এলিমেন্ট থাকে তাহলে এর সর্বশেষ উপাদানের ইন্ডেক্স হবে n-1.

অর্থাৎ একটি লিস্ট যদি এমন হয়ঃ

```
list = ["python",67, 0.1, 'O' ]
```

এখানে ইন্ডেক্স ভ্যালু শুরু হবে ০ থেকে। প্রথম এলিমেন্ট থেকে ইন্ডেক্সিং করলে তাকে ফরোয়ার্ড ইন্ডেক্সিং বলে। আমরা চাইলে লিস্টকে ব্যাকওয়ার্ড ইন্ডেক্সিং ও করতে পারি সেক্ষেত্রে শেষ এলিমেন্ট থেকে ইন্ডেক্সিং শুরু হয় এবং মান হয় -১।

লিস্ট অপারেশনস

পাইথন লিস্টে আমরা এলিমেন্ট যোগ করা, বাদ দেয়া, আলাদা লিস্ট বানানো, দুটো লিস্ট যোগ করা সহ নানা রকমের অপারেশন চালাতে পারি।

কনকেটেনেটিং

দুটো লিস্ট একসাথে জুড়ে দেয়াকে লিস্ট কনকেটেনেশন বলে। পাইথনে লিস্ট কনকেটেনেশনের ক্ষেত্রে "+" অপারেটর ব্যবহার করা হয়। এখানে অবশ্যই অপারেন্ড দুটি লিস্ট টাইপের হতে হবে।

```
listA = ["a","b","c"]
listB = [1, 2, 3]
listADD = listA + listB
```

স্লাইসিং

একটি লিস্ট থেকে চাইলে একাধিক সাব লিস্ট তৈরি করা যায়।

```
>>> listA = [1,5,4,7,8,9]
>>>listA[2:]
[4,7,8,9]
>>>listA[:3]
[1,5,4]
>>>listA[1:4]
[5,4,7,8]
```

আপডেটিং

লিস্ট যেহেতু মিউটেবল বা পরিবর্তনীয় তাই আমরা চাইলে লিস্টের যেকোন জায়গায় যে কোন সময় এলিমেন্ট ইনপুট করতে পারি।

```
>>> listA=[6,8,9,3,5,1]
>>> listA[2] = 10
>>> listA
[6,8,10,3,5,1]
```

লিস্টের ২ নং ইন্ডেক্সে ভ্যালু ১০ ইনপুট কর। কিন্তু এক্ষেত্রে নতুন ডেটা লিস্টের আগের ডেটা কে রিপ্লেস কিরে ফেলবে।

এপেন্ডিং

এপেন্ড অর্থ যোগ করা। আমরা চাইলে লিস্টে একটি করে ডেটা ইনপুট করতে পারি তার জন্য এপেন্ড এক্সপ্রেশন ব্যবহার করা হয়।

```
>>>listA = [1,2,5,8,6]
>>>listA.append(7)
[1,2,5,8,6,7]
```

ডিলিটিং/ রিমোভিং

এপেন্ডের মত রিমোভ ফাংশন ব্যবহার করে লিস্ট থেকে একটি করে ডেটা রিমোভ/ডিলিট করা যায়।

```
>>>listA.remove(5)
[1,2,8,6,7]
```

listA তে যদি একাধিক 5 থাকত তাহলে সবগুলো 5 রিমোভ হওয়ার পরিবর্তে প্রথম যে 5 টি পাওয়া যেত পাইথন সেটি রিমোভ করে দিত।

```
>>>listA =[1,2,5,6,7,5,4]
>>>listA.remove(5)
[1,2,6,7,5,4]
```

লেনথ্

লিস্টের দৈর্ঘ্য বের করার জন্য len() ফাংশন ব্যবহার করা হয়।

```
>>>listA = (1,2,3)
>>>len(listA)
3
```

রিপিটিশন

একটি লিস্ট একাধিক বার আউটপুট হিসেবে চাইলে শুধু মাত্র সাধারণ গুণ করে দিলেই হয়। তবে লিস্ট গুলো আলাদা আলাদা লিস্ট আকারে প্রিন্ট না হয়ে একটি মাত্র লিস্ট আকারে প্রিন্ট হবে।

```
>>> li = ['T','C','O']
>>> li*2
['T', 'C', 'O', 'T', 'C', 'O']
```

ম্যাক্সিমাম-মিনিমাম এলিমেন্ট

একটি লিস্টের সবচেয়ে বড়/ছোট উপাদান বের করতে লিস্টে ও max() এবং min() ফাংশন ব্যবহার করা হয়ে থাকে।

```
>>> listA=( 2,6,7,9)
>>>max(listA)
9
>>>min(listA)
2
```

মেম্বারশিপ

কোন একটা উপাদান একটি লিস্টের মেম্বার কিনা তাও চেক করা যায়। তার জন্য in কী-ওয়ার্ড ব্যবহার করা হয়।

```
>>>listA = [1,2,3,4]
>>>3 in listA
True
>>> if 3 in listA:
        print("Present")
    else:
        print("Absent")
Present
```

ক.২ ট্যুপল/ টাপল(tuple)

টাপল হচ্ছে পাইথনের একটি বিল্ট ইন ডেটা স্ট্রাকচার। ট্যুপল আর লিস্টের মধ্যে বেসিক পার্থক্য হচ্ছে ট্যুপল ইমিউটেবল। মানে ট্যুপলে কোন পরিবর্তন করা যায় না।
ট্যুপল লিখতে আমরা ফার্স্ট ব্রাকেট না প্যারেনথিসিস ব্যবহার করি।
tup = () #empty tuple

লিস্টের মত ট্যুপলে ও আমরা এলিমেন্ট গুলো কমা সেপারেটেড করে লিখি।
tup = (1,3,4,7)

বলে রাখা ভাল ট্যুপলে চাইলে আমরা প্যারেনথিসিস ব্যবহার না ও করতে পারি। সেক্ষত্রে ও পাইথন একে ট্যুপল অবজেক্ট হিসেবেই নিবে।

tup = "y"," a", "r"

ট্যুপল অপারেশনস্

পাইথন ট্যুপলে আমরা এলিমেন্ট যোগ করা, বাদ দেয়া, দুটো টুপলের তূলনা করা সহ নানা রকমের অপারেশন চালাতে পারি।

আপডেট

টুপল যেহেতু অপরিবর্তনীয় (ইমিউটেবল) তাই টুপলে সিঙ্গলে ভ্যালু আপডেট/ইনসার্ট করতে পারব না।

তবে একাধিক টুপল যোগ/ কনক্যাট করা যায়।

```
tup1 = ("y","a","r")
tup2 = (1,2,3)
tup3 = tup1 + tup2
print(tup3)
```

আউটপুটঃ
("y","a","r",1,2,3)

ডিলিট

পাইথনে টুপল ডিলিট করার জন্য del স্টেটমেন্ট ব্যবহার করা হয় । তবে এতে পুরো টুপলই ডিলিট হয়ে যায়।

```
>>>del tup3
>>>print(tup3)
```

এই কোড রান করাতে গেলে এমন এরর আসবে…
NameError: 'tup3' is not defined

কারণ পাইথন পুরো ট্যুপল টা- ই ডিলিট করে দিয়েছে।

লেনথ্

ট্যুপলের লেনথ্ বের করার জন্য লিস্টের মত len() এক্সপ্রেশন ব্যবহার করা হয়।

```
>>>tup = (1,2,3)
>>>len(tup)
3
```

রিপিটিশন

একটি ট্যুপল একাধিক বার আউটপুট হিসেবে চাইলে শুধু মাত্র সাধারণ গুণ করে দিলেই হয়। তবে ট্যুপল গুলো আলাদা আলাদা ট্যুপল আকারে প্রিন্ট না হয়ে একটি মাত্র ট্যুপল আকারে প্রিন্ট হবে।

```
>>>tup*3
(1,2,3,1,2,3,1,2,3)
```

কম্পেয়ার

দুটো টুপলের উপাদান গুলোর মধ্যে আমরা চাইলে তুলনা করতে পারি। এর জন্য পাইথনে cmp() ফাংশন ব্যবহার করা হয়।

```
>>>tup1= (1,2,3)
>>>tup2= (2,3,5)
>>>cmp(tup1,tup2)
```

ম্যাক্সিমাম-মিনিমাম এলিমেন্ট

একটি ট্যুপলের সবচেয়ে বড়/ছোট উপাদান বের করতে লিস্টের মত ট্যুপলে ও max() এবং min() ফাংশন ব্যবহার করা হয়ে থাকে।

```
>>> tup=( 2,6,7,9)
>>>max(tup)
9
>>>min(tup)
2
```

মেম্বারশিপ

কোন একটা উপাদান একটি টুপলের মেম্বার কিনা তাও চেক করা যায়।

```
>>>tuple = (1,2,3,4)
>>>3 in tuple
True

>>> if 3 in tuple:
        print("Present")
    else:
        print("Absent")
```

এক্সেসেসিং, ইনডেক্সিং, স্লাইসিং ইন ট্যুপল

ট্যুপলের উপাদান গুলো এক্সসেস/ট্রাভার্স করার জন্য আমরা লিস্টের মত ইন্ডেক্সিং এর সাহায্য নিতে পারি। ট্যুপলের ক্ষেত্রে ও ফরওয়ার্ড ইন্ডেক্সিং শুরু হয় ০ থেকে এবং ব্যাকওয়ার্ড ইন্ডেক্সিং শুরু হয় -১ থেকে।

```
>>> tuple = (4,3,7,6)
>>> tuple[1]
3
>>>tuple[-1]
6
```

একটা ট্যুপলের এক বা একাধিক উপাদান কেটে/স্লাইস করে নিয়ে চাইলে অন্য আরেকটি টুপল তৈরি করা যায়। একে স্লাইসিং বলে।

```
>>> tuple = (2,5,7,9,5)
>>> tuple[1:]
(5,7,9,5)
```

এর মানে ইন্ডেক্স ১ থেকে শুরু করে পরের সব গুলো উপাদান।

```
>>>tuple[:3]
(2,5,7,9)
```

এর মানে হচ্ছে ০ শূন্য ইন্ডেক্স থেকে ৩ নং ইন্ডেক্স পর্যন্ত উপাদান গুলো।

```
>>>tuple[1:3]
(5,7)
```

full() ম্যাক্সিমাম সাইজ ও উপাদান সংখ্যা সমান হয়ে গেলে True রিটার্ন করে।

get() – এই ফাংশনের কাজ দুটো। প্রথমত, কিউ এর প্রথম উপাদান টি রিটার্ন করে এবং তা ডিলিট করে। অন্যদিকে LIFO কিউ এর ক্ষেত্রে লাস্ট উপাদানটি রিটার্ন করে ও ডিলিট করে।

qsize() কিউ এর সাইজ রিটার্ন করে।

এর মানে হচ্ছে ইন্ডেক্স নং ১ থেকে শুরু করে ৩ নং ইন্ডেক্সের আগ পর্যন্ত সকল উপাদান তবে ৩নং ইন্ডেক্স নয়।

ম্যাপিং টাইপস

ম্যাপিং টাইপ ডেটা টাইপের বৈশিষ্ট্য হল অন্যান ডেটা টাইপের মত এদের কোন ইনডেক্স নেই বরং প্রতিটি ডেটার সাথে একটি 'কি' দেয়া থাকে যার মাধ্যমে সে 'কি' তে থাকা ভ্যালু কে আমরা এক্সেস করতে পারি। এই 'কি' এবং 'ভ্যালু'র জোড় কে 'কি-ভ্যালু পেয়ার' বলে।

ডিকশনারি dict{}

ডিকশনারি পাইথনের একটি বিল্ট ইন ডেটা টাইপ। ডিকশনারি তে প্রতিটি এলিমেন্ট এর দুটো অংশ থাকে। "কি" এবং "ভ্যালু"। "কি" হচ্ছে এলিমেন্টের ইন্ডেক্স বা এলিমেন্ট নির্দেশক। আর ভ্যালু হচ্ছে এলিমেন্টের ভ্যালু। প্রতিটা "কি" অবশ্যই ইউনিক হতে হয় তবে একাধিক ভ্যালু একইরকম হতে পারে। ডিকশনারি প্রকাশ করতে পাইথনে কার্লিব্র্যাসি { } ব্যবহার করা হয়।
ডিকশনারি তে এলিমেন্ট গুলোর "কি" এবং "ভ্যালু" সেমিকোলন (:) দ্বারা সেপারেটেডে থাকে এবং প্রত্যেকটা এলিমেন্ট কমা (,) সেপারেটেড করে লিখা হয়।

```
dictA = {
"key1" : "value1",
"key2" : "value2",
"key3" : "value2",
}
```

আমরা চাইলে ডিকশনারিতে বিভিন্ন টাইপের ভ্যালু রাখতে পারি কিন্তু 'কি' গুলো অবশ্যই ইমিউটেবল ডেটা টাইপের হতে হবে। যেমনঃ স্ট্রিং,নাম্বার,টুপ্যল।

এক্সেসিং ভ্যালু

লিস্ট এবং ট্যুপলে ডেটা এক্সেস করার জন্য ইন্ডেক্সিং এর ব্যবহার করছি। কিন্ত ডিকশনারির ডেটা এক্সেস করার জন্য আমরা এর 'কি' গুলো ব্যবহার করি।

```
>>>dic = {"First" : "Red",
          "Second" : "Yellow",
          "Third" : "Green"
}
>>>print(dic['Second'])
"Yellow"
>>> dic.values()
dict_values(['Red', 'Yellow', 'Green'])
```

আবার যদি এমন কোন 'কি' কল করা হয় যে নামে কোন 'কি' ডিকশনারিতে নেই তখন
KeyError এক্সেপশন রেইজ করবে।
আমরা ডিকশনারির ভ্যালু থেকে ও এর key বের করতে পারব।

```
dic ={"a":1, "b":2, "c":3}

keys = list(dic.keys())
values = list(dic.values())

print(keys [values.index(1)])
print(keys [values.index(3)])
```

আউটপুটঃ
a
c

প্রথমে ডিকশনারির কী ও ভ্যালু গুলোকে লিস্ট অবজেক্টে রূপান্তর করে নিয়েছি। তারপর লিস্টের index() অ্যাট্রিবিউট ব্যবহার করে values লিস্টের এলিমেন্টের ইন্ডেক্স বের করেছি ও এ ভ্যালুকে keys লিস্টে ইনডেক্স হিসেবে পাস করেছি।

এই কাজটা অন্যভাবে ও করা যায়ঃ

```
def get_key(val):
    for key, value in dic.items():
        if val == value:
            return key
    return "key doesn't exist"

dic ={"a":1, "b":2, "c":3}

print(get_key(1))
print(get_key(2))
```

আউটপুটঃ

a
b

আমাদের যে ভ্যালুর key দরকার সে ভ্যালুটা আমরা get_key() ফাংশন আর্গুমেন্ট হিসেবে পাস করেছি।get_key() ফাংশন লিস্টের আইটেম গুলোর মধ্যে সেই ভ্যালু সার্চ করে, খেয়াল করে দেখ এখানে key এবং value দুটোই ইটারেটর তাই প্রতিবার লুপেই key-value জোড় সেটাই হবে যেটা ডিকশনারিতে আছে। value আর্গুমেন্টের সাথে মিলে গেলে get_key() সেই key-value জোড়ের key টি রিটার্ন করে দিbe আর না মিললে “key doesn’t exist” রিটার্ন করে দিবে।

আপডেট

ডিকশনারি একটি মিউটেবল ডেটা টাইপ,আমরা চাইলে যেকোন সময় ডিকশনারি আপডেট করতে পারি। নিচের নিয়মে ডিকশনারি তে নতুন এন্ট্রি বা পুরোনো এন্ট্রি আপডেট করতে হয়…

```
>>>dictionary = { 'Name' : 'Nadim', 'Roll' : '67', 'Dept' : 'CSE' }
```

```
>>>dictionary[ 'Name' ] = 'Chowdhury' #Updating existing entry
>>>dictionary[ 'Sec' ] = 'A' #Adding new entry
```

অথবা ডিকশনারির update() অ্যাট্রিবিউট ও ব্যবহার করা যায়।

```
car = {"brand":"ford",
       "model":"mustang",
       "year":1995
       }
car.update({"color":"red"})
```

ডিলিট

ডিকশনারি তে সিঙ্গেল এলিমেন্ট, সব এলিমেন্ট বা পুরো ডিকশনারি ই ডিলিট করে দেয়া যায়।
পুরো ডিকশনারি ডিলিট করতে
del স্টেটমেন্ট ব্যবহার করা হয়।

```
>>>dictionary = {'Name' : 'Nadim', 'Roll' : '67', 'Dept' : 'CSE' }
>>>del dictionary['Dept'] #removing single element
>>>dictionary.clear() #removing all the elements
>>>del dictionary[ ] #removing the total dictionary
```

নেস্টেড ডিকশনারি

যদি একটি ডিকশনারি ভিতরে আরো ডিকশনারি থাকে তাহলে আমরা কিভাবে সে ডিকশনারি থেকে ভ্যালু বের করে আনব? ধরি আমাদের নিম্নের মত একটি ডিকশনারি আছে…

```
dictionary = {"Class_One"  : {'roll_1':'Rabib','roll_2':'Rujhan','roll_3':'Ahabab'},
              "Class_Two"  : {'roll_1':'Yash','roll_2':'Shabab','roll_3':'Sadman'},
              "Class_Three": {'roll_1':'Awan','roll_2':'Arabi','roll_3':'Jaha'},
        }
```

আমরা যদি ক্লাস ওয়ানের রোল এক এর নাম জানতে চাই তাহলে কিভাবে জানতে পারবে? হয়তো ক্লাস ওয়ান কে আলাদা করে তারপর আবার তার "কি" দিয়ে এক্সেস করতে পারবে। কিন্তু আমরা একই লাইনের মাধ্যমেই ডেটা এক্সেস করতে পারবে।

```
>>> dictionary["Class_One"]['roll_1']
Rabib
```

এখানে প্রথম "কি" টা["Class_One"] মেইন ডিকশনারির "কি"। এই "কি" এর ভ্যালু হিসেবে যে ডিকশনারি টি রিটার্ন হবে সে ডিকশনারি "কি" হচ্ছে এর পরের "কি" ['roll_1'] এবং এই স্টেট মেন্ট টি এর ডেটা ই রিটার্ন করবে।

```
>>>world = {'Europe': {"France": {'Capital': "Paris"},
                      "Germany": {'Capital': "Berlin"},
                      "Spain": {'Capital': "Madrid"}},
           'Asia': {"Japan": {'Capital': "Tokyo"},
                    "Bangladesh":{'Capital': "Dhaka"},
                    "India":{'Capital': "New Delhi"}}
       }

>>>world['Asia']["Bangladesh"]['Capital']
Dhaka
```

এভাবে যত ইচ্ছা নেস্টেড ডিকশনারি এক্সেস করা যায়। জেসন JSON টাইপের ডেটা ও একই ভাবে এক্সেস করা হয়।

কাউন্টার Counter()

অন্যান্য কালেকশন ডেটা টাইপের (list, tuple, dict) মত পাইথনের collection মডিউলের একটি ডেটা টাইপ বা ডেটা স্ট্রাকচার হচ্ছে counter. counter মূলত count ফাংশনের কাজ করে। তবে এটি একটি পুরো ডেটা সেটের বিভিন্ন ডেটার আলাদা আলাদা সংখ্যা গণনা করে।এটি dict ক্লাসের একটি সাবক্লাস। counter

ডেটা টাইপে লিস্ট,টুপল,স্ট্রিং রাখা যায়। এটি একটি ডিকশিনারি রিটার্ণ করে। যার 'কি' হচ্ছে লিস্টের প্রতিটি ইউনিক ভ্যালু এবং 'ভ্যালু' হল সে এলিমেন্ট লিস্টে কত বার আছে তার মান।

```
>>>from collections import Counter
>>>s = Counter('sfsjsbfajs')
>>>s
Counter({'s': 4, 'f': 2, 'j': 2, 'b': 1, 'a': 1})
>>>
>>>c = Counter(['a','b','a','a',1,4,1,1,'b'])
>>>c
Counter({'a': 3, 1: 3, 'b': 2, 4: 1})
>>> t = Counter((1,3,1))
>>>t
Counter({1: 2, 3: 1})
>>>type(t)
<class 'collections.Counter'>
>>>issubclass(Counter,dict)
True
```

আমরা কাউন্টার ডেটা টাইপে একটি মাত্র আর্গুমেন্ট পাস করতে পারব। অর্থাৎ একটি লিস্ট বা একটি স্ট্রিং বা একটি ট্যুপল। এক সাথে একাধিক আর্গুমেন্ট পাঠানো যাবে না তাহলে এক্সেপশন দেখাবে।

```
>>> from collections import Counter
>>> c = Counter(['a','a','b'],[1,2,3,'a'])
TypeError: expected at most 1 arguments, got 2
```

এছাড়া ও পাইথন Counter() এ কি- ওয়ার্ড আর্গুমেন্ট পাস করা যায়।

আপডেটিং

অন্যান্য কালেকশন ডেটা টাইপের মত Counter ও আপডেট করা যায়।একটি এম্পটি কাউন্টার বা এক্সিস্টিং কোন কাউন্টারেও নতুন এলিমেন্ট যোগ করা যায়।

```
>>>c = Counter()
# defining an empty Counter
>>>c.update("a")
# adding values to "c"
>>>c
Counter({a:1})
>>>c.update("aab")
>>>c
Counter({a:3,b:1})
```

ডিলিটিং

কাউন্টার ডেটা টাইপ থেকে কোন ডেটা ডিলিট করার জন্য subtract() ফাংশন ব্যবহার করা হয়।

```
>>>c.subtract('a')
>>>c
Counter({'a':2, 'b':1})
>>>c.subtract('ab')
>>>c
Counter({'a':1, 'b':0})
```

এক্সেসিং

Counter এর গঠন পুরোপুরি ডিকশেনারির মতই। এর এলিমেন্ট গুলো এক্সেস ও করা যায় ডিকশনারির মত 'কি' ব্যবহার করে।
তবে ডিকশিনারির ক্ষেত্রে কোন কি যদি অনুপস্থিত থাকত আমরা KeyError পেতাম, এক্ষেত্রে কাউন্টার আমাদের "শূন্য" পাঠাবে।

```
>>>c['a']
3
```

```
>>>c['d']
0
```

আমরা যদি কোন 'কি' এর ভ্যালু বাড়াতে চাই মানে "a" এর মান "3" থেকে বাড়াতে বা কমাতে চাই তাহলে সে কি এর ভ্যালু রি-অ্যাসাইন করে দিতে হবে।

```
>>>c['a'] = 2
>>>c
Counter ({'a':2, 'b':1})
```

কাউন্টারের আরেকটি ফাংশন হল elements(). এ ফাংশন টি কাউন্টারের ভেতর থাকা প্রতিটি এলিমেন্ট একটি ইটারেটর অবজেক্ট আকারে আউটপুট দেয়। তবে এর হিউম্যান রিডেবল ফরম্যাট পেতে আমাদের এটিকে লিস্টে কনভার্ট কর নিতে হয়।

```
>>>c.elements()
<itertools.chain object at 0x7c07339860>

>>>list(c.elements())
Counter ('a', 'a', 'b')
```

এখানে লক্ষনীয় যে, কাউন্টারের ভেতর যে এলিমেন্ট(কি) এর মান(ভ্যালু) যত ছিল সেই এলিমেন্ট ঠিক ততবার আলাদা আলাদা ভাবে যুক্ত হয়েছে।

কাউন্টার একটি কন্টেইনার যা কোন ডেটাসেটে কোন ডেটা কত সংখ্যক বার রয়েছে তার ভ্যালু ধারন করে।

সেট টাইপস

ক। সেট set()

সেট পাইথনের একটি ইমিটেবল ডেটা টাইপ। ডিকশিনারির মত পাইথনে সেট লিখার জন্য কার্লি ব্রেসিস {} ব্যবহার করা হয়, তবে এখানে ডিকশিনারির মত "কি-ভ্যালু" পেয়ার থাকে না। ডেটা গুলো অবিন্যস্ত অবস্থায় থাকে। আবার set() মেথড দিয়ে ও সেট ক্লাসের অবজেক্ট তৈরি করা যায়। সেটের প্রতিটি ভ্যালু ই হয় ইউনিক।

মানে তুমি চাইলে একই ডেটা একাধিক বা রাখতে পার কিন্তু পাইথন সেসব বাদ দিয়ে প্রত্যেকটি ডেটা একবার করে রাখবে।

```
>>> a_set={'a','b','c','d','c'}
>>> type(a_set)
<class 'set'>
>>> a_set
{'c', 'd', 'b', 'a'}
```

এখানে আমি 'a' এবং 'c' দু'বার করে রাখলে ও পাইথন সেগুলোকে একবার করে সেটে জমা করেছে।

সেট অপারেশন্স

আমরা গণিতে সেটের সমাধান করার সময় সেটে নতুন উপাদান যোগ করা বা বাদ দেয়া ছাড়া ও ইউনিয়ন, ইন্টারসেকশন ইত্যাদি অপারেশন চালাতে পারতাম। পাইথন সেট ও এর ভিন্ন নয়।

আপডেট

কোন সেটে নতুন কোন মেম্বার যোগ করার জন্য add() মেথড ব্যবহার করা হয়।

```
>>>a_set = {'a','b','c','d'}
>>>a_set.add('e'}
>>>print(a_set)
{'a','b','c','d','e'}
```

তবে add() মেথড ব্যবহার করে আমরা প্রতিবারে একটির বেশি এলিমেন্ট যোগ করতে পারব না। একসাথে একাধিক এলিমেন্ট যোগ করার জন্য update() মেথড ব্যবহার করতে হয়।

```
>>>a_set = {'a','b','c','d'}
>>>a_set.update('e','f','g'}
>>>print(a_set)
{'a','b','c','d','e','f','g'}
```

ডিলিটং

সেটের এলিমেন্ট ডিলিট করার জন্য discard(),remove(),pop() এই তিনটি মেথড ব্যবহার করা হয়।
pop() মেথড সবসময় সেটের শেষ এলিমেন্ট কে ডিলিট করে।

```
>>>a_set = {'a','b','c','d'}
>>>a_set.pop()
>>>print(a_set)
{'a','b','c'}
```

discard() এবং remove() মেথড একই কাজ করে, তবে একটি পার্থক্য হলঃ

discard() মেথড ব্যবহার করে যদি কোন এলিমেন্ট ডিলিট করতে হয় তবে সে এলিমেন্ট যদি সেটে না থাকে তাহলে ও পাইথন কোন এরর মেসেজ দেখাবে না।

কিন্তু remove () মেথডের ক্ষেত্রে যে এলিমেন্ট আমরা ডিলিট করব সেটি সেটে অবশ্যই থাকতে হবে তা না হলে পাইথন এরর দেখাবে।

```
>>> a_set = {'a','v','d'}
>>> type(a_set)
<class 'set'>
>>> a_set.remove('a')
>>> a_set
{'d', 'v'}
>>> a_set.discard('a')
>>> a_set.remove('a')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    a_set.remove('a')
KeyError: 'a'
```

ইন্ডেক্সিং

সেট অন্যান্য ডেটা টাইপের মত ইন্ডেক্সিং সাপোর্ট করে না। তুমি যদি কোন সেটের এলিমেন্ট কে তার ইন্ডেক্স দিয়ে এক্সেস করতে চান তাহলে এরর দেখাবে।

```
>>>a_set = {'a','b',1,'c'}
>>>print(a_set[0])
TypeError: 'set' object does not support indexing
```

এ কারনেই আমরা যখন add() বা update() ফাংশন ব্যবহার করি তখন একেক বার একের জায়গায় এলিমেন্ট যোগ হয়।

ইউনিয়ন

দু'টো বা তার বেশি সেটের সব উপাদান গুলো যোগ করার পদ্ধতিকে সেট ইউনিয়ন বলে।

```
>>> a = {'a','b','c'}
>>> b = {'1','2','3'}
>>> a|b
{'3', '2', 'a', 'c', 'b', '1'}
>>> c = {1,2,3}
>>> a|b|c
{1, '3', '2', 2, 3, 'a', 'c', 'b', '1'}
```

এখানে লক্ষ করবে সেট গুলোতে ভিন্ন ধরনের ডেটা থাকা সত্ত্বেও সেট গুলোর মধ্যে ইউনিয়ন অপারেশন চালানো সম্ভব।

ইন্টারসেকশন

দুই বা ততোধিক সেটের উপাদান গুলো থেকে সব সেটেই বিদ্যমান এমন উপাদান বের করার পদ্ধতিকে সেট ইন্টারসেকশন বলে।

```
>>> a&b&c
set()
```

```
>>> a = {'a','b','c'}
>>> b = {'b','c','d'}
>>> a&b
{'c', 'b'}
```

সেট ডিফারেন্স

একটি সেট থেকে আরেকটি সেট বাদ দেয়াকে সেট ডিফারেন্স বলে।

```
>>> a-b
{'a'}
```

এছাড়া ও কোন উপাদান সেটে আছে কিনা। কোন সেটে অন্য কোন সেটের সাবসেট কিনা ইত্যাদি অপারেশন পাইথন সেট এ করা যায়।

```
>>> 'a' in a
True
>>> 's' in a
False
>>> a = {'a','b'}
>>> b = {'a'}
>>> a > b   #subset
True
>>> a < b
False
```

খ। ফ্রোজেন সেট

ফ্রোজেন অর্থ জমাটবদ্ধ। আমরা সাধারনত সেট'এ এলিমেন্ট যোগ করা,বিয়োগ করা, সহ নানা অপারেশন চালাতে পারি। কিন্তু ফ্রোজেন সেট হচ্ছে একটি জমাটবদ্ধ সেট যাকে পরিবর্তন করার অনুমতি পাইথন আমাদের দেয় না।

```
>>> a = frozenset()
```

```
>>> a.add(3)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    a.add(3)
AttributeError: 'frozenset' object has no attribute 'add'
```

অধ্যায় ৩

পাইথন কন্ট্রোল স্ট্রাকচারস

প্রোগ্রাম কে নিয়ন্ত্রন করার জন্য কন্ট্রোল স্ট্রাকচার ব্যবহার করা হয়।

ইফ – এলস (if – else)

পাইথনে কন্ডিশনাল স্টেটমেন্ট হিসেবে if/ elif /else ব্যবহার করা হয়েছে। অন্যান্য ল্যাঙ্গুয়েজ switch/case স্টেটমেন্ট ও ব্যবহার করা হয়। কিন্তু পাইথনে সেসব নেই। পাইথনের if স্টেটমেন্টের ব্যবহার খুবই সিম্পল এবং সহজ। কয়েকটা উদাহরণ দেখলেই সহজে বোঝা যাবে।

```
>>> if 8 > 7:
print("True")
```

যদি একাধিক কন্ডিশন থাকে তখন আমরা elif এবং else ব্যবহার করি।

```
a = 10
b = 9

if (a>b):
    print("a is bigger than b")
```

```
elif(b>a):
    print("b is greater than a")
else:
    print( "Both are equal")
```

বুলিয়ান অপারেশন্স

পাইথন কন্ডিশনাল স্টেটমেন্ট ব্যবহারের সময় আমরা প্রায়ই (and, or, not) এই তিনটি শব্দ ব্যবহার করি। এগুলোই হচ্ছে বুলিয়ান অপারেশন্স। পাইথন ডকুমেন্টেশন অনুযায়ী এদের প্রায়োরিটি অর্ডার হচ্ছে, or, and তারপর not.

- or - যখন কন্ডিশনে দেয়া স্টেটমেন্ট গুলোর মাঝে কমপক্ষে একটি সত্য হয় তখন আউটপুট পাওয়া যায়।
- and- কন্ডিশনের সব স্টেটমেন্টস সত্য হলেই কেবল মাত্র আউটপুট পাওয়া যায়।
- not- যখন কন্ডিশনাল টা মিথ্যা হয় তখন আউটপুট পাওয়া যায়, কন্ডিশনাল সত্য হলে আউটপুট পাওয়া যায় না।

```
a = 5
b = 8
c = 7
if(a>b or b>c):
print(b)

if(a>b and b>c):
   print(b)
else:
   print("condition false")


my_list = [1,6,8,9,7,5]
n = 4
if n not in my_list:
```

```
print(" n is absent in a_list")
```

if __name__ = "__main__":

খুবই কমন কন্ডিশনাল স্টেটমেন্ট। এটি সবসময় একটা ফাইলের সবার শেষে বা নিচে লিখা হয়। আমরা জানি প্রত্যেকটি ফাংশনের __name__ একটি অ্যাট্রিবিউট থাকে। আমরা যখন কোন একটি প্রোগ্রাম রান করি তখন সেটি আমাদের __main__ ফাংশন হিসেবে এক্সিকিউট হয়। এই কন্ডিশনের মানে হচ্ছে আমরা যখন ঠিক এই প্রোগ্রাম টা ই রান করব শুধু মাত্র তখন এই প্রোগ্রামের সকল ফাংশন্স ক্লাস গুলো এই কন্ডিশন অনুযায়ী কাজ করবে।
তাছাড়া আমরা যদি এই প্রোগ্রামের ক্লাস বা ফাংশন্স গুলো অন্য কোন প্রোগ্রামে ইম্পোর্ট করে ব্যবহার করি তখন এই কন্ডিশনের কোন স্টেটমেন্ট বা কোন ডেটা ওই ক্লাস বা ফাংশন্স গুলোতে কোন প্রভাব ফেলবে না।

ব্যাপার টা কিছুটা কনফিউজিং হতে পারে। আমরা ইম্পোর্টিং নিয়ে পরে বিস্তারিত আলোচনা করব।

পাইথন লুপ (Loop)

কম্পিউটার প্রোগ্রামিং এ লুপ হচ্ছে কোন স্টেট্মেন্ট বার বার করা যতক্ষণ পর্যন্ত না এটা একটা নির্দিষ্ট শর্ত (কন্ডিশন) পূরণ করে। অন্যান্য সব প্রোগ্রামিং প্রোগ্রামিং ল্যাংগুয়েজের মত পাইথনেও লুপ ব্যবহার করা হয়। পাইথন দু'রকমের লুপ আছেঃ

*for loop

*while loop

ফর লুপ (for loop)

ফর লুপ হচ্ছে একটা ইটারেটর মেথড।যার কাজ হচ্ছে শর্ত সাপেক্ষে একটা কাজ বারবার করা। পাইথনে 'সি' এর মত for (i=0; i<n; i++) লুপ স্টাইল ব্যবহার না করে 'for in' স্টাইল ব্যবহার করা হয়েছে যা অন্যান্য ল্যাঙ্গুয়েজে 'for each' এর ন্যায়

কাজ করে। ফর লুপ দিয়ে আমরা প্রতিবারে এক এক করে ভ্যালু বাড়াতে পারি অর্থাৎ আমরা চাইলেই i += 2 করে মান বাড়াতে পারব না।

ফর লুপ সিন্ট্যাক্সঃ

```
for iterator_variable in sequence:
    statements(s)
```

উদাহরণঃ

```
>>> for i in range(5):
        print(i)

0
1
2
3
4
```

```
num=2
for a in range (1,6):
    print(num * a)
```

```
2
4
6
8
10
```

নেস্টেড লুপ

একটি লুপের ভেতরে যদি অন্য একটি লুপ থাকে তাহলে তাকে নেস্টেড লুপ বলে।

```
for i in range(1,6): #loop 1
    for j in range (1,i+1): #loop 2
        print (i, sep=(' '),end=(' '))
    print()
```

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

এই উদাহরণে প্রোগ্রামটি যখন প্রথম লুপে প্রবেশ করে তখন i এর মান 1 এবং তা 6 এর আগে পর্যন্ত মানে 5 পর্যন্ত চলবে।

i এর মান 1 এর জন্য যখন দ্বিতীয় লুপে প্রবেশ করবে তখন j এর মান শুরু হবে 1 থেকে এবং চলবে i+1 =1+1 =2 এর ঠিক আগে পর্যন্ত অর্থাৎ 1 বার , এভাবে i এর মান যখন 1 তখন দ্বিতীয় লুপটি ও 1 বার এক্সিকিউট হবে, i এর মান যখন 2 তখন দ্বিতীয় লুপ টি ও 2 বার এক্সিকিউট হবে। কারন ফর লুপের ম্যাক্সিমাম লিমিটের মান যদি n হয় তাহলে লুপ টি (n-1) সংখক বার এক্সিকিউট হয়।
এভাবে প্রত্যেক i এর মানের জন্যে দ্বিতীয় লুপ টি i এর মানের সমান বার এক্সিকিউট হবে এবং ঠিক তত বার i এর মান প্রিন্ট করবে।

range()

range() ফাংশন হচ্ছে জেনারেটর। যা একটি জেনারেটর অবজেক্ট রিটার্ন করে। সিনট্যাক্স:

```
range( start_value, end_value, increament_value)
```

ফাংশন তিনটি প্যারামিটার নিতে পারে। start_value ফাংশন কত থেকে কাউন্টিং শুরু করবে তা ডিফাইন করে, end_value ডিফাইন করে ফাংশনের শেষ সীমা এবং increment_value ডিফাইন করে প্রতি ধাপে কত করে মান বৃদ্ধি হবে। start_value এবং end_value হচ্ছে অপশনাল। মানে ফাংশনে এই দুটি প্যারামিটার পাস না করলে ও ফাংশন এক্সিকিউট হবে। সেক্ষেত্রে বাই-ডিফল্ট start_value এর মান হবে 0 এবং incremental_value এর মান হবে 1।

```
>>>range(5)
range(0, 5)
>>> a = range(5)
>>>print(type(a))
< class 'range'>
```

for লুপের সাথে ব্যবহারের ক্ষেত্রে আমরা range() ফাংশনের অবজেক্ট গুলো কে ট্রাভার্স করার জন্য একটি ভ্যারিয়েবল ব্যবহার করতে হয়।

```
<for variable in range(Iterable_object):>
```

```
>>> for i in range(5):
print(i)
```

```
0
1
2
3
4
```

যেহেতু আমরা কোন start_value এবং incremental_value দেই নি তাই বাই-ডিফল্ট range() ফাংশন এদের মান যথাক্রমে 0 এবং 1 ধরে নিয়েছে। যদিও end_value এর মান 5 দেয়া হয়েছে কিন্তু range() এখানে 4 পর্যন্ত আউটপুট দিয়েছে কারণ end_value এর মানের চেয়ে ছোট প্রিন্ট করতে পারবে, সমান নয়।

```
for i in range(1,10,2)
print(i)
```

```
1
3
5
7
9
```

range. ফাংশন এর মাধ্যমে স্বয়ংক্রিয় ভাবে একটি লিস্ট তৈরি করা যায়, যার এলিমেন্ট গুলো হয় একটি নির্দিষ্ট ক্রম অনুযায়ী।

```
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।

enumerate()

enumerate() হচ্ছে পাইথনের একটি বিল্ট ইন ফাংশন, এটি এক ধরনের জেনারেটর যা কোন ইটারেবল অবজেক্ট কাউন্ট করে এবং সে কাউন্টারের ভ্যালু রিটার্ন করে যাকে enumerate অবজেক্ট ও বলা হয়।

enumerate() ফাংশনে দুটি প্যারামিটার পাস করা যায়। প্রথমটি হল যে কোন Iterable অবজেক্ট (যেমন, নাম্বার, স্ট্রিং, লিস্ট, টুপল ইত্যাদি) এবং দ্বিতীয়টি হল Start ভ্যালু,যেখান থেকে কাউন্টিং শুরু করা হবে।

```
enumerate(iterable_object,start_value)
```

```
items = [ 'apple','orange','mango']
enu_Items = enumerate(items)

print(type(enu_Items))     #Output: <class 'enumerate'>

print (enu_Items)
#Output: <enumerate object at 0x00000224BAD80750>

# converting to list
print (list(enu_Items))
#Output:[(0, 'apple'), (1, 'orange'), (2, 'mango')]



# changing the default counter by passing START parameter
enu_Items = enumerate(items, 10)
print (list(enu_Items))
```

```
#Output:
[(10, 'apple'),
(11, 'orange'),
(12, 'mango')]
```

```
#use a for loop over a collection
Months = ["Jan","Feb","Mar","April","May","June"]
for i, m in enumerate (Months):
        print(i,m)
```

```
0 Jan
1 Feb
2 Mar
3 April
4 May
5 June
```

zip()

একাধিক ইটারেবল অবজেক্টে একই সাথে ফর লুপ চালাতে zip() ফাংশন ব্যবহার করা হয়। zip() আর্গুমেন্ট হিসেবে দুই বা ততোধিক ইটারেবল অবজেক্ট নেয় এবং প্রতিটি ইটারেবল অবজেক্টের একই ইন্ডেক্স এর ভ্যালু গুলোকে একটি টুপল রিটার্ন করে।

```
Men = ['Romeo','Forhad','Majnu','Shajahan']
Women = ['Juliet','Siri','Laily','Momtaj']

for couple in zip(Men,Women):
    print(couple)
```

আউটপুটঃ

```
('Romeo', 'Juliet')
('Forhad', 'Siri')
('Majnu', 'Laily')
('Shajahan', 'Momtaj')
>>> type(couple)
<class 'tuple'>
```

এখানে আমরা একটি মাত্র ইটারেটর ব্যবহার করেছি কিন্তু প্রতিটি ইটারেবল এর জন্য আলাদা ইটারেটর ব্যবহার করা যায়। মনে করি, একটি কম্পানির তিনটি ডেটাসেট আছে যার একটিতে কর্মচারীর নাম,একটি তে পদবী ও অন্যটিতে বেতন ক্রমানুযায়ী রাখা আছে। আমাদের সে তিনটি ডেটা সেট থেকে ডেটা গুলোকে ফরম্যাট করে সহজে বোঝার মত একটি আউটপুট দেখাতে হবে।

```
names = ('Richard','Danesh','Gylfoyl','Jarad','Erlick')
positions = ('CEO','Software Eng','Tester','Legal Adviser','Sales Executive')
salaries = (10000,8000,8000,5000,6000)


for name,position,salary in zip(names,positions,salaries):
    print(f"{name} is the '{position}' and he is paid {salary}$ per week")
```

আউটপুটঃ

```
Richard is the 'CEO' and he is paid 10000$ per week
Danesh is the 'Software Eng' and he is paid 8000$ per week
Gylfoyl is the 'Tester' and he is paid 8000$ per week
Jarad is the 'Legal Adviser' and he is paid 5000$ per week
Erlick is the 'Sales Executive' and he is paid 6000$ per week
```

হোয়াইল লুপ (while loop)

while লুপ হচ্ছে একটি কন্ডিশন্ড লুপ। আপাতত দৃষ্টিতে একে ফর লুপের মত মনে হলে ও এটির কাজ ভিন্ন। ফর লুপে আমরা কোন ইটারেবল অবজেক্টের উপর

ইটারেশন চালাই। কিন্তু while লুপ এক্সিকিউ করার জন্য আমরা কন্ডিশন চেক করি। while লুপ ততক্ষন পর্যন্ত একটি কোড ব্লক কে এক্সিকিউট করবে যতক্ষন condition == True.

while লুপ এক্সপ্রেশনঃ

```
while [a condition is True]:
      [do something]
```

উদাহরণঃ

```
a=10
while a>0:
   print ("Value of a is",a)
   a=a-2
   print ("Loop is Completed")
```

```
Value of a is 10
Value of a is 8
Value of a is 6
Value of a is 4
Value of a is 2
Loop is Completed
```

উদাহরণ ২ঃ

```
n=153
total=0
while n>0:
   r = n%10
```

```
    total += r
    n = n/10
print (total)
9 #output
```

অনুশীলনী

গত তিনটি অধ্যায়ে আমরা যা শিখেছি সেগুলো ব্যবহার করে আমরা কিছু সমস্যা সমাধান করব। এতে করে প্রোগ্রাম লিখার অভ্যাস হবে ও কোথায় কি ব্যবহার করতে হবে তার ধারনা লাভ করা যাবে।

সমস্যা ১ একটি প্রোগ্রাম লিখতে হবে যে দুটো লাইন ইনপুট নিবেপ্রথম লাই,নের ভ্যালু হবে দ্বিতীয় লাইনের ভ্যালু গুলোর ইন্ডেক্সের মান। আমাদের এই উপাদান গুলো কে ইন্ডেক্স অনুযায়ী সাজিয়ে আউটপুট দেখাতে হবে। যেমনঃ

3 1 2
44 20 12

এখানে 44 এর ইন্ডেক্স হচ্ছে 3, 20 এর ইন্ডেক্স হচ্ছে 1 এবং 12 এর ইন্ডেক্স হচ্ছে 2।
অর্থাৎ আউটপুট হবেঃ
20
12
44

সমাধানঃ

```
line1 = list(map(int,input().split()))
line2 = list(map(str,input().split()))
```

```
dictionary = {}
for i in range(len(line1)):
    dictionary[line1[i]] = line2[i]

for i in sorted(line1):
    print(dictionary[i])
```

প্রোগ্রামটিতে প্রথম লাইন ইনপুট দেয়ার সময় খেয়াল রাখতে হবে যাতে একই সংখ্যার পুনরাবৃত্তি না হয়।কারণ, প্রথম লাইন হচ্ছে ইন্ডেক্স ভ্যালু। একই ইন্ডেক্সে একাধিক মান রাখা যাবে না। যদিও কোন এরর দেখাবে না কিন্তু প্রত্যাশিত আউটপুট ও আসবে না। এবং দুটো লাইনেই সমান সংখ্যক উপাদান ইনপুট দিতে হবে।
এবার আসা যাক কিভাবে সমাধান করলাম? প্রথমে আমরা একটি ডিকশনারি তৈরি করেছি। তারপর একটি লুপ চালিয়েছি ০ শূণ্য থেকে line1 এর লেনথ পর্যন্ত যার ভেতর আমরা ডিকশনারির 'কি' হিসেবে সেট করেছি line1[i] নং এলিমেন্ট কে এবং 'ভ্যালু' হিসেবে সেট করেছি line2[i] নং এলিমেন্ট কে। এর ফলে আমাদের যে ভ্যালুর জন্য যে ইনডেক্স অ্যাসাইন করার কথা আমরা তা করে ফেলেছি। এখন আর শুধু সেগুলো কে সর্ট করে প্রিন্ট করতে হবে। কিন্তু আমরা জানি ডিকশনারির কোন sort() ফাংশনেই। তাই আমরা আমদের line1 লিস্ট অর্থাৎ ইনডেক্সের লিস্ট কে সর্ট করব এবং সে সর্টেড লিস্টের ভ্যালু গুলোর সাথে ম্যাচ করে ডিকশনারির 'কি' ধরে এক্সেস করব তাহলে ডিকশনারি তে ভ্যালু গুলো যেভাবেই থাকুক না কেন আমাদের কে সর্টেড আউটপুট ই দেখাবে।

সমস্যা ২: এ সমস্যা টি সামান্য জটিল। আমাদের দুটো সংখ্যা ইনপুট দেয়া হবে। প্রথম সংখ্যা টি হল অ্যামপ্লিচিউড এবং দ্বিতীয় সংখাটি হল ফ্রিকোয়েন্সি।

```
4
2
#output
1
22
333
4444
```

```
333
22
1

1
22
333
4444
333
22
1
```

এখানে আমাদের ১ থেকে শুরু করে n (অ্যামপ্লিচিউড) পর্যন্ত প্রিন্ট করতে হচ্ছে এবং আবার উলটো ভাবে ১ এ ফেরত আসতে হচ্ছে। তাহলে আমাদের টোটাল লাইন সংখ্যা কত হচ্ছে? টোটাল লাইন সংখ্যা হচ্ছে (n+(n-1)) . তার মানে আমাদের লুপ টি এত বার চালাতে হবে। এবার লক্ষ করুন প্রতিটি লাইনের অংক গুলোর সংখ্যা তাদের মানে সমান। মানে ১ নং লাইনে ১ টি এক, ২ নং লাইনে ২ টি দুই, ৩ নং লাইনে ৩ টি ৩, ৪ নং লাইনে ৪টি ৪, কিন্তু ৫ নং লাইনে ৩ টি ৩। সমস্যা শুরু হচ্ছে তাহলে n এর মান অতিক্রম করার পর। এখন আমরা n এর মানের সাথে আউটপুট গুলোর একটা সম্পর্ক স্থাপনের চেষ্টা করব।

আমরা যখন লুপ চালাবো, লুপের ইটারেটর i এর মান n থেকে যখন বড় হচ্ছে তখন আমরা লক্ষ করলে দেখব n এর মান যখন প্রিন্ট হচ্ছে (4444) তখন থেকে পরের প্রতি লাইনের আউটপুট গুলো মান ১ করে কমছে। তার মানে আমরা যদি প্রতিবারে এই ১ কে ইনক্রিমেন্ট করে n এর মান থেকে বিয়োগ করে দেই তাহলে আমরা আউটপুট টা পেয়ে যাব।
আর ফ্রিকোয়েন্সির মান হচ্ছে এই পুরো প্রসেস টা কত বার রিপিট হবে তার মান।

আমরা ইতিমধ্যে সমাধান করে ফেলেছি। এবার তোমার দায়িত্ব হল এই আলোচনা অনুযায়ী নিচের সমাধানের প্রতিটি লাইন কোড বুঝে নেয়া।

সমাধান

```
def sequence(n):
    return (str(n)*n)
```

```
amplitude = int(input())
frequency = int(input())
loop_times = amplitude+(amplitude-1)

for f in range(frequency):
    to_minus = 1
    for i in range(1,loop_times+1):
        if i > amplitude:
            i = amplitude - to_minus
            print(sequence(i))
            to_minus += 1
        else:
            print(sequence(i))

    if f < frequency-1:
            print()
```

সমস্যা ৩: আমাদের একটি প্রোগ্রাম লিখতে হবে যা একটি লাইন ইনপুট নিবে এবং লাইনের প্রতিটি শব্দ কে আলাদা আলাদা ভাবে রিভার্স করে সেই রিভার্সড শব্দ গুলো দিয়ে একটি লাইন প্রিন্ট করবে। খুবই সহজ একটি প্রোগ্রাম।

Hello World

olleH dlroW

সমাধান

```
while True:
    try:
        sentence = list(input().split(" "))
        reverse = []
        for i in sentence:
            reverse.append(i[::-1])
```

```
    print(*reverse)
except:
    break
```

প্রোগ্রাম টি অতি সহজ একটি প্রোগ্রাম। এ প্রোগ্রামের শুধু একটি লাইন তোমাদের চিন্তায় ফেলতে পারে। তা হল *reversed এই অ্যাস্টেরিস্ক চিহ্ন কেন ব্যবহার করা হয়েছে তা তোমরা এটি সরিয়ে একবার রান করলেই কারন টা জেনে যাবে।

অধ্যায় ৪

পাইথন কম্প্রিহেনশন্স (Comprehensions)

পাইথন কম্প্রিহেনশন্স খুবই মজার একটি বিষয়। কম্প্রিহেনশন্স লিখায় তুমি যত পারদর্শী হবে নিজেকে ততটা স্পেশাল প্রোগ্রামার মনে হবে।কারণ কম্প্রিহেনশন লিখা টা একটু জটিল। এই অধ্যায়ে আমরা পাইথন কম্প্রিহেনশন্স সম্পর্কে আলোচনা করব।

লিস্ট কম্প্রিহেনশন্স

বেসিক সিনট্যাক্সঃ

(values) = [(expression) **for** in (data) **in** (collection)]

এই এক্সপ্রেশনকে বোঝার জন্য আমরা এর সাধারন পদ্ধতি দেখিঃ

```
(values) = []
for (data) in (collection):
        (values).append((expression))
```

এখানে expression এর ভ্যালু data হতে পারে বা যে কোন অপারেশন হতে পারে।

```
>>> numbers = [1,2,3,4,5,6]
>>> numbers_list = []
>>> for n in numbers:
        numbers_list.append(n)

>>> print(numbers_list)
```

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

এই প্রোগ্রামটাকে যদি কম্প্রিহেনশন ব্যবহার করে লিখা হয় তাহলে এমন হবে,

```
>>> numbers_list = [n for n in numbers]
>>> numbers_list
[1, 2, 3, 4, 5, 6]
```

এখানে প্রথম n হল expression এবং দ্বিতীয় n হল data এর ভ্যালু আবার যখন if ব্যবহার করব তখন আমাদের সিনট্যাক্স টা হবে এমনঃ

```
[(expression) for (data) in (collection) if (condition)]
```

এর সাধারন পদ্ধতি হলঃ

```
(values) = []
for (data) in (collection):
    if (condition):
        (values).append((expression))
```

উদাহরণ ২

```
>>>even_numbers = []
>>>even_numbers = [ n for n in numbers if n%2 == 0 ]
[2, 4, 6]      #output
```

এই সম্পূর্ন কোড কে আরও ছোট করে ১ লাইনে লিখা যায়।

```
>>>even_numbers = print(list([n for n in numbers if n%2 == 0]))
[2, 4, 6]      #output
```

আমরা প্রথমে যে আউটপুট পাচ্ছি সেগুলোকে লিস্ট অবজেক্টে পরিবর্তন করে নিয়েছি এবং print() ফাংশনে পাস করে দিয়েছি।

উদাহরণ ৩

```
my_list = []
```

```
for letter in 'abcd':
    for num in numbers:
        my_list.append((letter,num))

print(my_list)
```

এই প্রোগ্রাম লিখার জন্য নেস্টেড লিস্ট কম্প্রহেনশন ব্যবহার করতে হবে।

```
my_list=[(letter,num) for letter in 'abcd' for num in numbers ]
print(my_list)
```

ফর লুপ দুটোর স্থান পরিবর্তন করে আউটপুটের ভিন্নতার কারণ বিশ্লেষণ করার চেষ্টা কর।

ডিকশনারি কম্প্রিহেনশন্স

লিস্ট কম্প্রিহেনশনের মত একইভাবে ডিকশনারি কম্প্রিহেনশন ও ব্যবহার করা যায়। এটি পাইথন ৩ তে নতুন সংযোজন করা হয় এবং পরে পাইথন২ তে ও সংযোজন করা হয়। এর গঠনও লিস্ট কম্প্রিহেনশনের মতই । লিস্টের ক্ষেত্রে আমরা এক্সপ্রেশনের জায়গায় শুধুমাত্র ভ্যালু নিয়ে কাজ করেছি এখন আমাদের **কি-ভ্যালু** নিয়ে কাজ করতে হবে।

```
>>> dictionary = {'1':'a','2':'b','3':'c'}
>>> dictionary.keys()
dict_keys(['1', '2', '3'])
>>> dictionary.values()
dict_values(['a', 'b', 'c'])
>>> dictionary.items()
dict_items([('1', 'a'), ('2', 'b'), ('3', 'c')])

>>> sqaure_of_values = { key : value*2 for (key, value) in dictionary.items()}
>>> sqaure_of_values
{'1': 'aa', '2': 'bb', '3': 'cc'}
```

'কী'র মান পরিবর্তন করা এবং লিস্ট কম্প্রিহেনশনের মত if স্টেট্‌মেন্ট ব্যবহার করার চেষ্টা কর। এখন আমরা দুটো লিস্ট থেকে কম্প্রিহেনশনের মাধ্যমে ডিকশনারি তৈরি করা দেখবঃ

```
>>> a = [1,2,3,4,5]
>>> b = ['a','b','c','d','e']
>>> dictionary = {key : value for key,value in zip(a,b)}
>>> dictionary
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
```

এখানে a লিস্টের এলিমেন্টকে key এবং b লিস্টের এলিমেন্ট কে ভ্যালু হিসেবে নেয়া হয়েছে। এবং কি-ভ্যালুকে zip() ফাংশনে পাঠানো হয়েছে। zip() ফাংশন প্রতিবার একটি ইটারেবল অবজেক্ট রিটার্ন করে যা ডিকশনারি তে কনভার্ট হচ্ছে। যদিও এখানে এটি আমাদের আলোচনার বিষইয় না তারপর ও জেনে রাখা ভাল যে, zip() ফাংশন সবসময় নুন্যতম সংখ্যক এলিমেন্ট নিয়ে কাজ করে। অর্থ্যাৎ,যেকোন লিস্টের মধ্যে সবচেয়ে কমসংখ্যক এলিমেন্ট যে লিস্টে থাকবে zip() শুধু ততবার ই ইটারেশন চালাবে।

```
>>>numberList = [1, 2, 3]
>>> strList = ['one', 'two', 'three','four']
>>> dict(zip(numberList,strList))
{1: 'one', 2: 'two', 3: 'three'}
>>> strList = ['one', 'two']
>>> dict(zip(numberList,strList))
{1: 'one', 2: 'two'}
```

zip() ফাংশনে দুইয়ের অধিক ইটারেবল অবজেক্ট ও পাঠানো যায় এবং unzip ও করা যায়।

সেট কমপ্রিহেনশন্স

সেট কম্প্রিহেনশন্স অনেকটাই ডিকশনারি কম্প্রিহেনশনের মত। এটি সেট অবজেক্ট রিটার্ন করে।

সিন্ট্যাক্সঃ

{expression **for** variable **in** collection}

উদাহরন

```
>>> {s for s in [1, 2, 1, 0]}
set([0, 1, 2])
>>> {s**2 for s in [1, 2, 1, 0]}
set([0, 1, 4])
>>> {s**2 for s in range(10)}
set([0, 1, 4, 9, 16, 25, 36, 49, 64, 81])

>>> {s for s in [1, 2, 3] if s % 2}
set([1, 3])
>>> {(m, n) for n in range(2) for m in range(3, 5)}
set([(3, 0), (3, 1), (4, 0), (4, 1)])
```

অধ্যায় ৫

ফাংশনাল প্রোগ্রামিং

একটি সুবিন্যস্ত কোড ব্লক যা একটি নির্দিষ্ট কাজ করার জন্য লিখা হয় তাকে ফাংশন বলে। কোডকে হালকা, রিডেবল এবং একই ধরনের কোড বারবার লিখা পরিহার করার জন্য ফাংশন লিখা হয়। কোডের একই ধরনের কাজ এক্সিকিউট করার ক্ষেত্রে সে কাজের জন্য যে ফাংশন লিখা হয়েছে তা কল করে কাঙ্ক্ষিত আউটপুট পাওয়া যায়।

পাইথনে দু ধরনের ফাংশন ব্যবহার করা হয়।

- বিল্ট-ইন (built in)
- ইউজার ডিফাইন্ড (user defined)

পাইথনে যেসব ফাংশন ডিফাইন করা আছে সেগুলো হচ্ছে বিল্ট-ইন ফাংশন্স। যেমনঃ print(), map(),input()

অন্যদিকে যেসব ফাংশন প্রোগ্রামার রা নিজের কোন নির্দিষ্ট কাজ করার জন্য তৈরি করে সেগুলো হচ্ছে ইউজার ডিফাইন্ড ফাংশন্স।

ফাংশন ডিক্লেয়ারেশন

পাইথনে ফাংশন লিখার শুরুতে def কিওয়ার্ড ব্যবহার করা হয় এবং এরপর ফাংশনের নাম ও সাথে প্যারেন্থিসিস "()" ও শেষে কোলন (:) দিতে হয়। প্যারে-ন্থিসিসের ভেতর প্যারামিটারস/আর্গুমেন্টস লিখতে হয়।

ফাংশনের প্রথম লাইন হয় ডক স্ট্রিং যেখানে লিখা থাকে যে এই ফাংশন কি কি কাজ করবে। এটি যদিও অত্যাবশ্যক না তবুও

"It is a good practice"

এরপর ফাংশনের প্রধান ব্লক যেখানে কোড লিখা হয়। এবং সবার শেষে return স্টেটমেন্ট তাকে যা ফাংশনের আউটপুট রিটার্ন করে।

ফাংশন এক্সপ্রেশন

```
def function (parameters):
    """ function docstring'"""
    function body
    return (expression)
```

ফাংশন কলিং

একটি ফাংশন লিখার পর তা থেকে আউটপুট পাওয়ার জন্য ফাংশন কল করতে হয়। এবং ফাংশন এক্সিকিউট করার জন্য নুন্যতম যে আর্গুমেন্টস দরকার তা পাস্ করতে হয়। ফাংশন চাইলে ওই একই প্রোগ্রামের যেকোন যায়গায় বা অন্য ফাংশনে বা অন্য প্রোগ্রামে মেথড হিসেবে কল করা যায়। আবার পাইথন প্রম্পট এ রানটাইমেও কল করা যায়।
উদাহরণঃ

```
def addition(a,b):
   """ This function will return sum of two digits"""
      result =(a+b)
      return(result)
addition(5,6)
```

এত টুকু আমরা যখন রান করাব তখন আমরা কোন আউটপুট স্ক্রিনে দেখতে পাব না। কারণ এখানে কোন print() ফাংশন নেই।

কিন্তু এই ফাংশনকে যদি print() ফাংশনের ভেতর কল করি তাহলে আমরা আউটপুট দেখতে পাব

```
print (addition (5+6))

call_function = addition (5,6)
print(call_function)
```

এই দু ক্ষেত্রেই আমরা আউটপুট দেখতে পাব।

আমরা যখন কোন ফাংশন নিজে লিখি তখন সে ফাংশনের সব কিছুই আমরা জানি,কিন্তু যখন কোন লাইব্রেরি কল করি বা অন্য কারো ফাংশন ইম্পোর্ট করি তখন সে ফাংশন সম্পর্কে আমরা সব কিছু জানি না। তবে সেটা জানার ও উপায় আছে। আমরা চাইলে ফাংশন কে প্রশ্ন করতে পারি যে তোমার নাম কি? তোমার আর্গুমেন্টস কয়টা? তোমার ডিটেইলস বল। শুনলে মজার শোনালে ও আসলেই ফাংশন এগুলোর উত্তর দেয়। ইন্টারপ্রটারে আমরা এই addition ফাংশন টা ই ইম্পোর্ট করে দেখাচ্ছি…

```
>>> from myfunction import addition #myfunction is the filename

>>> addition.__name__
'addition'
>>> addition.__doc__
'docstring of the addition function'

#how many local variables do you have?
>>> addition.__code__.co_nlocals
3

>>> addition.__code__.co_code
b'|\x00|\x01\x17\x00}\x02|\x02S\x00' #byte code of the function



>>> addition.__code__
<code object addition at 0x000002F94C24BD20, file "C:/Users/ASUS-
PC/AppData/Local/Programs/Python/Python36\myfunction.py", line 1>
#physical address of the function.
```

এছাড়াও ভ্যারিয়েবলের নাম, ফাইলনেম সহ আরও অনেক কিছুই জানা যায়।

আর্গুমেন্টস্ অফ ফাংশন

আর্গুমেন্ট বা প্যারামিটার, ফাংশন কল করার সময় ফাংশন এক্সিকিউশনের জন্য প্রয়োজনীয় যে ডেটা পাঠাতে তাকে আর্গুমেন্টস বলে। পাইথন ফাংশনে চার ধরনের আর্গুমেন্টস ব্যবহার করা যায়।

- ডিফল্ট আর্গুমেন্টস
- রিকোয়ার্ড বা পজিশনাল আর্গুমেন্টস
- কি-ওয়ার্ড আর্গুমেন্টস
- ভ্যারিয়েবল লেনথ আর্গুমেন্টস

ডিফল্ট আর্গুমেন্টস

পাইথন ফাংশনে যখন একাধিক আর্গুমেন্টস ব্যবহার করা হয় তখন কিছু আর্গুমেন্টসের ডিফল্ট ভ্যালু দেয়া হয়।সে আর্গুমেন্ট পাস না করলে ও ফাংশন সে আর্গুমেন্টের জন্য ডিফল্ট ভ্যালুটি নিয়ে কোড এক্সিকিউট করে। একেই ডিফল্ট আর্গুমেন্টস বলা হয়।

```
>>>def defaultArg( name, msg = "Hello!"):
    return (name + msg)

>>> print(defaultArg('ratul'))
ratulHello!
```

আমরা যদি বাই-ডিফল্ট msg প্যারামিটারের একটি ভ্যালু না অ্যাসাইন করে দিতাম তাহলে পাইথন এটিকে "রিকোয়ার্ড আর্গুমেন্ট" হিসেবে ধরে নিত এবং এরর দেখাত।

```
>>> def defaultArg(name, msg):
        return (name + msg)
```

```
>>> print(defaultArg('ratul'))
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    print(defaultArg('ratul'))
TypeError: defaultArg() missing 1 required positional argument: 'msg'
```

রিকোয়ার্ড আর্গুমেন্টস

যে আর্গুমেন্টস ফাংশন কল করার সময় অবশ্যই পাস করতে হয় তাকে রিকোয়ার্ড আর্গুমেন্টস বলে। এবং একটি ফাংশনে একাধিক রিকোয়ার্ড আর্গুমেন্টস থাকলে তা ঠিক একই পজিশনে পাস করতে হয়।

```
def string (arg1, arg2):
     print(str)

string(6)
```

```
TypeError: string() missing 1 required positional argument: 'arg2'
[Program finished]
```

কি-ওয়ার্ড আর্গুমেন্টস

ফাংশন কল করার সময় যে আর্গুমেন্ট গুলোকে তাদের নাম সহ মেনশন করে পাস করা হয় তাদের কি-ওয়ার্ড আর্গুমেন্টস বলে।

```
>>>def keywordArg( name, role ):
        return(name+role)
>>># 2 keyword arguments
>>>keywordArg(name = "Tom", role = "1111")
```

কি-ওয়ার্ড আর্গুমেন্টের পজিশন চেঞ্জ করে পাঠালে ও পাইথন কোন এরর দেখায় না।তবে কোন পজিশনাল আর্গুমেন্টের আগে কোন কি-ওয়ার্ড আর্গুমেন্ট পাস করলে তখন পাইথন সিন্ট্যাক্স এরর দেখায়।

```
>>># 2 keyword arguments (out of order)
>>>keywordArg("role = "1111",name = "Tom)
><<# 1 positional, 1 keyword argument
>>>keywordArg("Tom", role = "1111")
>>>keywordArg(role = "1111","Tom")
SyntaxError: non-keyword arg after keyword arg
```

ভ্যারিয়েবল লেনথ আর্গুমেন্টস

অনেক প্রোগ্রাম লিখার সময় আমাদের জানা থাকে না যে ঠিক কতটা আর্গুমেন্টস পাঠানো হবে। আমরা যে সাধারন ক্যাল্কুলেটর ব্যবহার করি সেখানেও আমরা আগে থেকে বলে দেই না যে আমরা ঠিক কতটা সংখ্যার ক্যালকুলেশন করব। আমরা একের পর এক ইনপুট দিতে থাকি এবং ক্যালুটরের ফাংশন ফলাফল রিটার্ণ করতে থাকে। এরকম অনেক ক্ষেত্রেই অজানা সংখ্যক আর্গুমেন্টসের প্রয়োজন পড়তে পারে। তখনই আমরা পাইথনের ভ্যারিয়েবল-লেন্থ্ আর্গুমেন্ট ব্যবহার করি।

```
def greet(*names):
      """This ‘ * ’ converts the argument
            into variable length argument """
  # names is a tuple with arguments
  for name in names:
     print("Hello",name)

greet("Ayub","Meraj","Rashed","Tonmoy","Younus","Imran",)
```

আউটপুট

```
Hello Ayub
```

```
Hello Meraj
Hello Rashed
Hello Tonmoy
Hello Younus
Hello Imran
```

এবার সেই ছোট্ট প্রোগ্রাম টা লিখি যা অজানা সংখ্যক সংখ্যার যোগফল রিটার্ন করবে।

```
>>> def add(*nums):
        total = 0
        for num in nums:
                total += num
        return total

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

একইভাবে কি-ওয়ার্ড আর্গুমেন্ট ও ভ্যারিয়েবল লেন্থ্ হিসেবে ডিফাইন করা যায়। তখন একটি স্টারের '***args***' জায়গায় দুটি স্টার দিতে হয় '****kwargs***' দিতে হয়।

রিকার্শন ইন পাইথন

রিকার্শন বলতে বোঝায় নিজের কাজ নিজে করা। ব্যাপারটা এমন যে তুমি আয়নার সামনে দাঁড়িয়ে নিজেকে প্রশ্ন করছ এবং নিজেই তার উত্তর দিচ্ছ। আমরা একটা নির্দিষ্ট করার জন্য একটি ফাংশন তৈরি করি। যদি এমন হয় যে সে একই কাজটি ওই ফাংশনের ভিতর কোন একটা জায়গায় আবার করা লাগছে তখন আমরা ওই ফাংশনের ভেতরেই আবার তাকে-ই কল করি। একেই বলে রিকার্শন।

উদাহরণঃ

```
def factorial(n):
    print("factorial function has been called with n = " + str(n))
    if n == 1:
```

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

print("Final result %s" %(factorial(5)))
```

আউটপুটঃ

```
factorial function has been called with n = 5
factorial function has been called with n = 4
factorial function has been called with n = 3
factorial function has been called with n = 2
factorial function has been called with n = 1
final result 120
```

এ **factorial** ফাংশন টি প্রথমে **5** এর জন্য এক্সিকিউট হওয়া শুরু হয় এবং যেহেতু **n == 1** নয় তাই প্রোগ্রামটি **else** স্টেট্‌মেন্টে ঢুকে। তারপরের লাইনে **result** কাউন্ট করার সময় **factorial** ফাংশন নিজেকেই আবার কল করে এবং তখন আর্গুমেন্ট **n** এর মান হল (**n-1)=(5-1)=4** এভাবে একই ভাবে **1** পর্যন্ত এক্সিকিউট করে এবং রেজাল্ট রিটার্ন করে। এখানে আমাদের ফাংশনটি **n=1** এর জন্য শেষ হয়েছে। এইকন্ডিশন কে বলা হয় বেজ কন্ডিশন। কোন রিকার্সিব ফাংশনে যদি বেজ কন্ডিশন না থাকে তাহলে ফাংশনটি ইনফাইনাইটলি চলতে থাকবে।

অ্যানোনিমাস/ল্যাম্বডা ফাংশন (lambda)

পাইথনে অ্যানোনিমাস ফাংশন বলতে বোঝান নামহীন ফাংশন। পাইথনে ফাংশন ডিফাইন করার জন্য **def** কি ওয়ার্ড ব্যবহার করা হয়। তবে অ্যানোনিমাস ফাংশন ডিফাইন করার জন্য **lambda** কী-ওয়ার্ড ব্যবহার করা হয়।

এক্সপ্রেশনঃ

```
lambda arguments: expression
```

```
# Program to show the use of lambda functions
>>>double = lambda x : x * 2
>>>print(double(5))
10
```

lambda ফাংশনে যত ইচ্ছা আর্গুমেন্ট থাকতে পারে তবে কেবল মাত্র একটি এক্সপ্রেশন থাকে।

গ্লোবাল ভ্যারিয়েবল

কোন ভ্যারিয়েবল যদি কোন ফাংশন বা কোন নির্দিষ্ট কোড ব্লকের বাইরে অর্থাৎ গ্লোবাল স্কোপে ডিক্লেয়ার করা হয় তাকে গ্লোবাল ভ্যারিয়েবল বলে।একটি গ্লোবাল ভ্যারিয়েবল কোডের যেকোন জায়গায়,যেকোন ফাংশনে ব্যবহার করা যায়।

লোকাল ভ্যারিয়েবল

কোন ভ্যারিয়েবল যদি কোন ফাংশন বা কোন নির্দিষ্ট কোড ব্লকের ভিতরে ডিক্লেয়ার করা হয় তাকে লোকাল ভ্যারিয়েবল বলে। একটি লোকাল ভ্যারিয়েবল শুধু মাত্র এর নিজস্ব কোড ব্লকের ভিতরেই ব্যবহার করা যায়।

উদাহরণঃ

```
x = "global" #global variable

def foo():
    global x
    y = "local" #local variable
    x = x * 2
    print(x)
    print(y)

foo()

#output
globalglobal
local
```

```

```

এখানে global কি-ওয়ার্ড ব্যবহার করার কারণ হল আমরা x কে একটি গ্লোবাল ভ্যারিয়েবল হিসেবে ডিক্লেয়ার করেছি এখন আমরা যদি একে কোন কোড ব্লকের ভিতরে এর মান পরিবর্তন করতে চাই তখন গ্লোবাল ভ্যারিয়েবল এক্সেস করতে global কি-ওয়ার্ড ব্যবহার করতে হয়। global কি-ওয়ার্ড ব্যবহার না করলে আমা দের এই এরর টি দেখাবে।

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

নন-লোকাল ভ্যারিয়েবল

যদি কোন লোকাল ভ্যারিয়েবল কে এর নিজের স্কোপের বাহিরে ও ব্যবহার করতে হয় তখন 'nonlocal' কি-ওয়ার্ড ব্যবহার করে একে নন-লোকাল ভ্যারিয়েবল হিসেবে ডিক্লেয়ার করা হয়।

```
def myfunc1():
  x = "local variable "
  def myfunc2():
    nonlocal x
    x = " local variable value changed "
  myfunc2()
  return x

print(myfunc1())

output:
local variable value changed
```

এখানে যদি nonlocal কি-ওয়ার্ড ব্যবহার না করা হয় তবে আউট পুট আসবেঃ
local variable

কন্সট্রাক্টরস

কন্সট্রাক্টরস হল স্পেশাল টাইপ ফাংশন/ মেথড যা কিনা কল করা ছাড়াই এক্সি - কিউট হয়। কোন ক্লাসে এই কন্সট্রাক্টর ব্যবহার করলে এবং পরবর্তীতে সেই ক্লাসের

ইন্সট্যান্স তৈরির সময় এই মেথডটি স্বয়ংক্রিয় ভাবেই কল হয় যাতে করে এর মাধ্যমে কিছু সেটআপ রিলেটেড কাজ করে নেয়া যায়।কন্সট্রাক্টরে চাইলে প্যারামিটার পাস্ করা যায় আবার না ও করা যায়। একটি কন্সটাক্টর ফাংশন তৈরি করার ক্ষেত্রে ফাংশনের নামের প্রথমে ও শেষে দুটি করে আন্ডারস্কোর দিতে হয়। কন্সট্রাক্টর ফাংশনের নাম হয় __init__().

এই মেথড গুলোকে ভাষায় প্রকাশ করার সময় "dunders" বা "ডাণ্ডার ইনিট" এভাবে বলা হয়ে থাকে। আমরা যখন কন্সট্রাক্টর ছাড়া কোন ক্লাসের অবজেক্ট তৈরি করি তখন পাইথন একটি ডিফল্ট কন্সট্রাক্টর তৈরি করে যা আসলে কিছুই করে না।

```
class greeting():
    def __init__(self):
        print("Hellow")
    def user(name):
        print(name)

new_user = greeting()
new_user.user("Boss")
```

Hellow
Boss

এখানে Hellow আউটপুট পেতে আমরা শুধু ক্লাসের একটি অবজেক্ট তৈরি করতে হয়েছে। কিন্তু Boss আউটপুট পেতে সে ক্লাসের প্রপার্টি এক্সেস করতে হয়েছে।

পাইথনে কিছু বিশেষ ধরনের বিল্ট ইন মেথড আছে যেগুলোকে ম্যাজিক মেথড বলা হয়। এগুলার চেনার খুব সহজ উপায় হচ্ছে এদের নামের দুই ই দুটো করে আন্ডারস্কোর থাকে। অর্থাৎ, __init__ মেথডের মত। এই মেথডের সঙ্গে ইতোমধ্যে আমাদের পরিচয় হয়েছে।

এই __init__ মেথড বাদেও অনেক গুলো ম্যাজিক মেথড আছে পাইথনে। ম্যাজিক মেথডের খুব বহুল ব্যবহার দেখা যায় অপারেটর অভারলোডিং এর সময়। প্রত্যেকটি অপারেটর এর জন্যই একটি ম্যাজিক মেথড আছে। যেমন, + অপারেটর এর জন্য ম্যাজিক মেথডটি হচ্ছে __add__

অর্থাৎ, যদি আমাদের এমন একটি এক্সপ্রেশন থাকে x+y তখন এই মেথড টি কল হ্য x.__add__(y)

কিছু কমন অপারেটরের ম্যাজিক মেথড

__sub__	-
__mul__	*
__truediv__	/
__floordiv__	//
__mod__	%
__pow__	**
__and__	&
__xor__	^
__or__	|
__lt__	<
__le__	<=
__eq__	==
__ne__	!=
__gt__	>
__ge__	>=

এছাড়াও আরও অনেক ম্যাজিক মেথড আছে পাইথনে যেমন - __len__ __getitem__ __setitem__ __delitem__ __iter__ __contains__ ইত্যাদি

জেনারেটরস

জেনারেটরস হল এক ধরনের ফাংশন যে ভ্যালু জেনারেট করে বা তৈরি করে। জেনারেটর ফাংশন কে চাইলে একটা নির্দিষ্ট জায়গায় থামিয়ে ঠিক সে জায়গা থেকে আবার শুরু করতে পারি। **range()** একটি জেনারেটর ফাংশন। আমরা চাইলে নিজেরা ও জেনারেটর ফাংশন তৈরি করতে পারি। এর জন্য **yield** কি-ওয়ার্ড ব্যবহার করা হয়।

```
>>> a = range(3)
>>> a
range(0, 3)
>>> for i in a:
        print(i)

0
1
2
```

উদাহরণঃ

```
>>> def gen(end):
        for i in range(0,end+1,2):
            yield i

>>> data = gen(5)
>>> data
<generator object gen at 0x000001800794D410>
>>> next(data)
0
>>> next(data)
2
>>> next(data)
4
>>> next(data)
```

```
Traceback (most recent call last):
  File "<pyshell#73>", line 1, in <module>
    next(data)
StopIteration
```

কিছু বিল্ট-ইন ফাংশন

পাইথনে অনেক বিল্ট ইন ফাংশন্স আছে। সব গুলো লিখতে গেলে হয়তো কয়েক হাজার পৃষ্ঠার বই লিখা যেতে পারে তাও শেষ করা যাবে বলে মনে হয় না। আমরা এখানে কিছু ফাংশন নিয়ে আলোচনা করব যেগুলো সচরাসচর আমাদের কাজে লাগবে এবং এদের কাজ খুবই চমৎকার ।

chr()

এই ফাংশন কেবল একটি মাত্র ইন্টিজার আর্গুমেন্ট নেয়,এবং একটি ক্যারেক্টার ভ্যালু রিটার্ণ করে। ইন্টিজার ভ্যালু হচ্ছে ক্যারেক্টারটির ইউনিকোড ভ্যালু।

```
>>> print(chr(279), chr(222), chr(345), chr(334))
>>> print(chr(340),chr(194),chr(358),chr(364),chr(321))
Ŕ Â Ŧ Ŭ Ł
```

map() filter()

map()

মনে কর, আমাদের একটি লিস্ট আছে যেখানে আমাদের এক সপ্তাহের তাপমাত্রা দেয়া আছে। এখন আমাদের সে সাত দিনের তাপমাত্রা গুলোকে ফারেনহাইটে নিতে হবে। আমরা সাধারনত প্রোগ্রামটি যেভাবে লিখব সেটি হল প্রথমে একটি সেলসিয়াস-ফারেনহাইটে নেয়ার ফাংশন লিখব তারপর লিস্টের উপাদান গুলোর মধ্যে লুপ চালিয়ে ফাংশন টাকে কল করব। এই পদ্ধতি কে সহজতর উপায়ে করা যায় map() ফাংশন দিয়ে। map() আর্গুমেন্ট হিসেবে একটি ফাংশন নেয় ও

একাধিক ইটারেবল(লিস্ট,টুপল) অবজেক্ট নেয় এবং সে ইটারেবল অবজেক্টের প্রতিটি উপাদানের জন্য ফাংশন টি কল করে এবং একটি ম্যাপ অবজেক্ট রিটার্ন করে যা আমরা list,set ইত্যাদিতে ডেটা টাইপ্সে পরিবর্তন করে নেই।

```
def celcius_to_fahrenheit(temp_in_celcius):
    temp = (temp_in_celcius*1.8)+32
    return temp
list_of_temp = [20,22,24,32,35,28,30]

for i in list_of_temp:
    print(celcius_to_fahrenheit(i))

print(list(map(celcius_to_fahrenheit,list_of_temp)))
```

আউটপুটঃ

```
68.0
71.6
75.2
89.6
95.0
82.4
86.0
#output for map()
[68.0, 71.6, 75.2, 89.6, 95.0, 82.4, 86.0]
```

যেহেতু map() ফাংশন আর্গুমেন্ট হিসেবে ফাংশন নেয় তাই আমরা চাইলে lambda ফাংশন ও ব্যবহার করতে পারি। উপরে সম্পূর্ন প্রোগ্রামটি চাইলে এক লাইনে লিখা যাবে।

```
print(list(map(lambda x: (x*1.8)+32, list_of_temp)))
#output
[68.0, 71.6, 75.2, 89.6, 95.0, 82.4, 86.0]
```

এবার তাহলে দেখি একাধিক ইটারেবল অবজেক্টের জন্য map() কিভাবে ব্যবহার করতে হয়।

```
a = (3,4,5,6)
b = (4,5,6,7,8)

#without map()
for i in range(min(len(a),len(b))):
    result = a[i]+b[i]
    print(result)

#using map()
print(list(map(lambda x,y: x+y,a,b)))
```

আউটপুটঃ

```
7
9
11
13
[7, 9, 11, 13]
```

filter()

ডেটা সেট থেকে কোন কন্ডিশনের ভিত্তিতে নির্দিষ্ট কিছু ডেটা নিয়ে আসার জন্য filter() ফাংশন ব্যবহার করা হয়। filter() ও map() এর মত একটি ফাংশন ও একটি ইটারেবল অবজেক্ট প্যারামিটার হিসেবে নেয়। এবং ইটারেবল অবজেক্টের প্রতিটি উপাদানের ওপর ফাংশনটি প্রয়োগ করে।

```
a = range(1,100) #less memory consuming than a list

print(list(filter(lambda x: x%2==0, a)))
```

আউটপুটঃ

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]
```

filter() ফাংশনের জায়গায় map() দিয়ে দেখতে পার আউটপুট কি আসে।

এবার আমরা এই দুটি ফাংশনের একসাথে ব্যবহার শিখব। আমরা এমন একটি প্রোগ্রাম তৈরি করব যে ১ থেকে ১০০ পর্যন্ত প্রতিটি বিজোড় সংখ্যার বর্গের মান একটি লিস্টে রাখবে।

```
a = range(1,100)
print(list(map(lambda x:x**2,filter(lambda x:x%2==1,a))))
```

এখানে প্রথমে আমরা ১ থেকে ১০০ এর ভেতরের বিজোড় সংখ্যা গুলো ফিল্টার করেছি। তারপর সে ফিল্টার করা সংখ্যা গুলো র বর্গ করেছি এবং লিস্টে ইনপুট করেছি। এখানে একটা বিষয় হল, আমরা ফিল্টার করার পর বিজোড় সংখ্যা গুলোকে কোন লিস্ট বা টুপল কিছুতেই রাখি নি বরং সেগুলো একটি ফিল্টার অবজেক্ট ছিল। তারপর ও ম্যাপ ফাংশন ইটারেশন করতে পেরেছে তার কারণ হল

ফিল্টার অবজেক্ট ও ইটারেবল অবজেক্ট।এই প্রোগ্রামটাকেই যদি আমরা ভেঙ্গে করি তাহলে সহজেই বোঝা যাবে।

```
a = range(1,100)
b = filter(lambda x:x%2==1,a)
print(b)
c = map(lambda x:x**2,b)
print(c)
print(list(c))
```

আউটপুটঃ

```
<filter object at 0x0000018BB1EB3A58>
<map object at 0x0000018BB1EA1F60>
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361, 441, 529, 625, 729, 841, 961,
1089, 1225, 1369, 1521, 1681, 1849, 2025, 2209, 2401, 2601, 2809, 3025,
3249, 3481, 3721, 3969, 4225, 4489, 4761, 5041, 5329, 5625, 5929, 6241,
6561, 6889, 7225, 7569, 7921, 8281, 8649, 9025, 9409, 9801]
```

eval() এবং exec()

আমরা পাইথন ইন্টারপ্রেটারে অনেক গুলো ইংরেজি শব্দ লিখে দেই। ইন্টারপ্রেটার সেগুলে পড়ে বুঝে নেয় যে তাকে কি করতে হবে। eval() এবং exec() এমন দুটি ফাংশন যাদের প্যারামিটার হিসেবে একটি স্ট্রিং পাঠানো হয়। এবং তারা সেটা পড়ে পাইথন কোডে রূপান্তর করে আউটপুট দেখায়। তবে সে স্ট্রিং টি অবশ্যই পাইথনের ভ্যালিড এক্সপ্রেশন বা স্টেট্মেন্ট হতে হবে।

```
>>> eval("print('i love programming')")
i love programming
>>> exec("print('i love programming')")
i love programming
```

এখন মনে হতে পারে, যদি দুটো ফাংশন একই কাজ ই করে তাহলে দুটো ফাংশনের কি দরকার। দুটো ফাংশন আসলে একই কাজ করে না। eval() প্রধানত কোন এক্সপ্রেশনের মান নির্ণয় করে। অন্যদিকে exec() স্টেট্মেন্ট এক্সিকিউট করে। এক্সপ্রেশন এবং স্টেট্মেন্টের প্রধান পার্থক্য হচ্ছে এক্সপ্রেশনে কোন ভ্যারিয়েবলের মান অ্যা সাইন করা হয় বা কোন এরিথমেটিক অপারেশন চালানো হয় অন্য দিকে স্টেট্মেন্ট হল কোন এক্সপ্রেশন কে পরিবর্তন করতে প্রভাবক হিসেবে কাজ করে(লুপ, কন্ডিশনাল)

```
>>> exec("for i in range(10): print (i)")
0
1
2
3
4
5
6
7
8
9

```

exec() ফাংশন যে স্ট্রিং এক্সিকিউট করতে পেরেছে eval() সে একই স্ট্রিং কে এক্সিকিউট করতে পারবে না কারন এটি একটি স্টেট্মেন্ট।

```
>>> eval("for i in range(10): print(i)")
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    eval("for i in range(10): print(i)")
  File "<string>", line 1
    for i in range(10): print(i)
      ^
SyntaxError: invalid syntax
```

স্ট্রিং কে ভ্যারিয়েবলে রেখে সে ভ্যারিবলকে আর্গুমেন্ট হিসেবে পাস করলে ও একই আউটপুট পাওয়া যায়।

```
>>> a = 'for i in range(10):print(i)'
>>> exec(a)
0
1
2
3
4
5
6
7
8
9
>>> eval(a)
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    eval(a)
  File "<string>", line 1
    for i in range(10):print(i)
      ^
SyntaxError: invalid syntax
```

কোন ফাংশনকে ও চাইলে প্যারামিটার হিসেবে পাস করা যায়। সেক্ষেত্রে একই ফাংশনের জন্য eval() ও exec() ভিন্ন ভিন্ন আউটপুট দিবে।

```
>>> def function(argument):
        print(f"Argument = {argument}")
        return argument*2

>>> eval('function(10)')
```

```
Argument = 10
20
>>> exec('function(10)')
Argument = 10
```

str() এবং repr()

দুটো ফাংশনই ডেটাকে 'স্ট্রিং' টাইপ করে রিপ্রেজেন্টেশনের জন্য ব্যবহার করা হয়। str() *ব্যবহার করা হয় এন্ড ইউজার দের আউটপুট দেয়ার জন্য যা সহজেই রিডেবল অন্যদিকে* repr() *ব্যবহার করা হয় ডিবাগিং এবং ডেভেলপিং এর সময় যা প্রোগ্রামারকে ডেটা/অবজেক্টের পুরো ডিটেইলস (কোন ক্লাসের,কোন ফাংশনের অবজেক্ট) সহ আউটপুট দেয়।*

str() কে __str__ (dunder str method) এবং repr() কে __repr__ (dunder repr method) দ্বারা রিপ্রেজেন্ট করা হয়।

উদাহরণঃ

```
string =  'hello world'
print (str(string))
print (repr(string))
```

আউটপুটঃ

hello world

'hello world'

ঊদাহরনঃ

```
from datetime import datetime
today = datetime.now()

# prints "readable" format for date-time object
print str(today)
```

```
# prints the "official" format of date-time object
print repr(today)
```

আউটপুটঃ

2018-09-24 10:57:14.216514

datetime.datetime(2018, 9, 24, 10, 57, 14, 216514)

অনুশীলনী

সমস্যা ১ একটি পরিক্ষায় নাম্বার দেয়ার পদ্ধতি হল যদি একাধারে একাধিক উওর সঠিক হয় তাহলে প্রতিটি সঠিক উত্তরের জন্য পূর্ববর্তী নাম্বারের সাথে এ **1** যোগ হবে এবং ভুল হলে এই এখন পর্যন্ত হওয়া মোট নাম্বার মূল নাম্বারের সাথে যোগ হয়ে টেম্পোরারি নাম্বার আবার 0 হয়ে যাবে। **'O'** = সঠিক উত্তর এবং **'X'** = ভুল উত্তর। অর্থাৎ, যদি আমাদের স্কোর কার্ড **OOXXOXXOOO** এমন হয় তাহলে আমাদের নাম্বার গণনা হবে **1+2+0+0+1+0+0+1+2+3"** এভাবে। অর্থাৎ, এই ইনপুটের জন্য আমাদের মোট নাম্বার হবে **10.**

আমরা, একটি ফাংশন ব্যবহার করে এর সমাধান করব। যে ফাংশন টি একটি স্ট্রিং ইনপুট নিবে যা হল আমাদের উত্তর গুলোর কম্বিনেশন। আমরা ফাংশন টির নাম দিয়েছি counter() এ ফাংশনের একটি ভ্যারিয়েবল হল total যাতে আমরা আমাদের মোট নাম্বার রাখব। এর পরের ভ্যারিয়েবল হল টেম্পোরারি নাম্বার রাখার জন্য যা আমাদের প্রতিটি সঠিক উত্তরের জন্য ১ করে বাড়বে ও ভুল উত্তরের জন্য আবার ০ শূণ্য তে ফিরে আসবে।

এখন স্ট্রিং থেকে প্রতিটি উত্তর আলাদা করে চেক করার জন্য আমরা একটি লুপ চালাবো। লুপে **i** হল ইটারেটর এবং **string** হল ইটারেবল। এখন **i == 'O'** (সঠিক উত্তর) পেলেই আমরা **increament** ভ্যারিয়েবলের সাথে এক যোগ করছি। এবং **increament** এর মান মোট নাম্বারের সাথে যোগ করে

দিচ্ছি। অন্যদিকে **i=='X'** (ভুল উত্তর) পেলেই আমরা **increament** ভ্যারিয়েবলের মান আবার **0** শূণ্য করে দিচ্ছি। তাহলেই আমরা পরের বার আবার যখন **i=="O"** পাব তখন আমরা আবার ১ থেকে নাম্বারিং করা শুরু করতে পারব।

সমাধান

```
def counter(string):
    total = 0
    increament = 0
    for letter in string:
        if letter == 'O':
            increament += 1
            total += increament
        else:
            increament = 0
    return total

if __name__ == "__main__":
        string = str(input())
        print(counter(string))
```

সমস্যা ২ এখন আমরা "ডাইসারিয়াম নাম্বার" বের করার জন্য একটি প্রোগ্রাম লিখব। কোন একটি নাম্বারের ডিজিটের ইন্ডেক্সের মানকে স্ব স্ব ডিজিটের পাওয়ার হিসেবে বসিয়ে তাদের মান গুলোকে যোগকরলে যোগফল যদি নাম্বারের সমান ই হয় তাহলে সে নাম্বার কে "ডাইসারিয়াম নাম্বার বলে" মনে কর একটি নাম্বার হল "**175**" এখন **1** এর ইন্ডেক্স হল **1,** (শূণ্য প্রযোজ্য নয়) **7** এর ইন্ডেক্স হল **2** এবং **5** এর ইন্ডেক্স হল **3.** তাহলে $\mathbf{1+7^2+5^3 = 175}$ হয় তাহলে এটি একটি "ডাইসারিয়াম নাম্বার" **1 + 49 + 125 = 175.** মানে এটি একটি ডাইসারিয়াম নাম্বার।

এই সমস্যা সমাধান করার জন্য আমরা একটি ফাংশন লিখব যা আমাদের বলে দিবে কোন একটি সংখ্যা ডাইসারিয়াম সংখ্যা কি না। **isDisarium()** ফাংশনের প্যারমিটার **num** এর মধ্যে আমরা আমাদের সংখ্যা টি পাঠাবো। এরপর আমাদের প্রথম কাজ হচ্ছে সংখ্যা টি তে কতটি ডিজিট আছে তা জেনে নেয়া। এর জন্য আমরা এর লেনথ জেনে নিলেই পারি। কিন্তু, সংখ্যা টি যেহেতু একটি ইন্টিজার অবজেক্ট এবং পাইথনে ইন্টিজার অবজেক্টের জন্য কোন **len()** অ্যাট্রিবিউট নেই তার জন্য আমাদের একটু ঘুরিয়ে এর লেন্থ জেনে নিতে হবে। আমরা ইন্টিজার কে প্রথমে স্ট্রিং টাইপে পরিবর্তন করে তারপর এর লেন্থ জেনে নিয়েছি এবার **num** কে আমরা একটি ভ্যারিয়েবল **temp** এ রেখেছি এবং যোগফল রাখার জন্য একটি **result** ভ্যারিয়েবল নিয়েছি।

আমরা সংখ্যাটির শেষ ডিজিট থেকে ক্যাল্কুলেশন শুরু করব। এখানে আমাদের সংখ্যা **175** এর শেষ ডিজিট হল **5**, এবং তা পাওয়ার জন্য আমরা সংখ্যা কে **10** দিয়ে ভাগ করলে এর ভাগশেষ হিসেবে শেষ ডিজিট টি পেয়ে যাব। এবার এই শেষ ডিজিটে এর ইন্ডেক্সের ভ্যালু কে পাওয়ার হিসেবে বসিয়ে এর মান বের করতে হবে। এখানে **5** এর ইন্ডেক্স হল **3.** আমাদের **power** ভ্যারিয়েবলের মান ও **3** কারন আমরা এতে পুরো সংখ্যা টির লেনথ এর মান রেখেছি।

(length == index_of_last_digit)

এবার **temp_result** ভ্যারিবলে **last_digit**power** এর মান রেখে **result** ভ্যারিয়েবলে **temp_result** এর মান যোগ করে দিয়েছি। এতটুকু পর্যন্ত আমাদের একটা ডিজিট(লাস্ট ডিজিটের কাজ শেষ)। এরপরে প্রতিটি ডিজিটের জন্য আমরা **power** এর মান **1** করে কমিয়েছি এবং আমাদের নাম্বার(**temp)** থেকে প্রতিবার লাস্টের ডিজিট বাদ দিয়ে **(temp//10)** নাম্বার কে আপডেট করেছি। এভাবে যতক্ষন পর্যন্ত সংখ্যায় ডিজিট শেষ না **temp != 0** হবে ততক্ষন আমরা এই লুপ চালাবো। এরপর কন্ডিশন চেক করে যদি **result == num** হয় তাহলে **True** অন্যথায় ফাংশন আমাদের **False** রিটার্ন করবে।

সমাধানঃ

```
def isDisarium(num):
    power = len(str(num))
    temp = num
```

```
    result = 0
    while temp != 0:
        last_digit = temp%10
        temp_result = last_digit ** power
        result += temp_result
        power -= 1
        temp = temp//10


    if result == num:
        return True
    else:
        return False

num = int(input())
if isDisarium(num):
    print(f"{num} is a Disarium Number")
else:
    print(f"{num} is not a Disarium Number")
```

সমস্যা ৩ একটি লিস্ট **a_list** এবং একটি একটি ইন্টিজার **k** দেয়া থাকবে। লিস্টের উপাদান গুলোর মধ্যে এমন জোড়া বের করতে হবে যাদের যোগফল **k** দ্বারা নিঃশেষে বিভাজ্য। এবং এমন জোড়ার সংখ্যা কয়টি তা বের করতে হবে। লিস্টে জোড়া গুলো পাশাপাশি না থাকলে ও হবে। ধর, লিস্টের **a_list[0] + a_list[3]** এর যোগফল যদি **k** দিয়ে নিঃশেষে বিভাজ্য হয় তাহলে ও সেই জোড়াকে গণনা করতে হবে।

a_list = [1,2,3,5,1,2]

k = 4

এখানে ৪ দিয়ে নিঃশেষে বিভাজ্য এমন জোড়া গুলো হলঃ

a_list[0] + a_list[2] = 4

a_list[1] + a_list[5] = 4

a_list[2] + a_list[3] = 8

a_list[2] + a_list[4] = 4

তার মানে আমাদের আউটপুট হবে 4। কারন চারটি জোড়া এমন যাদের যোগফল k দ্বারা নিঃশেষে বিভাজ্য।

সমাধানঃ

```
# Complete the divisibleSumPairs function below.
def divisibleSumPairs(a_list,k):
    count = 0

    for i in range(len(n_list)+1):
        for j in range(i+1,len(a_list)):
            if (a_list[i] + a_list[j])% k == 0:
                count += 1
    return count

if __name__ == '__main__':
    ar = list(map(int, input().strip().split()))
    k = int(input())
    result = divisibleSumPairs(ar, k)

    print(result)
```

এই প্রোগ্রাম টা নিজে নিজে বোঝার চেষ্টা কর।

অধ্যায় ৬

এক্সেপশন্স

প্রোগ্রাম লিখার সময় যদি ভুল হয় এবং প্রোগ্রামটি যদি পাইথন ইন্টারপ্রেটার পড়তে না পারে বা রান করার সময় ক্র্যাশ করে তখন পাইথন সেটা আমাদের মেসেজের মাধ্যমে জানিয়ে দেয়। এই ভুল গুলোকেই এক্সেপশন্স বলে। এই অধ্যায়ে আমরা বেশ কিছু এক্সেপশন্স এর সাথে পরিচিত হব এবং এদের কবলে পড়লে কিভাবে মুক্তি পেতে হয় সে পদ্ধতি সম্পর্কে জানব।

কমন এক্সেপশন্স (exceptions)

পাইথন প্রোগ্রামিং করার সময় সচরাচর যে এরর বা এক্সেপশন গুলো প্রায়শই দেখতে পাব সেগুলো কি এবং কেন হয় তার বেশ কয়েকটা এখানে দিলামঃ

TypeError: কোন ফাংশন বা অপারেশন যে টাইপের ডেটা দেয়ার কথা যদি ভিন্ন টাইপের কোন ডেটা দিয়ে অপারেশন চালানোর চেষ্টা করা হয় তখন TypeError দেখায়।

AttributeError: যখন কোন ফাংশনের "এট্রিবিউট" এসাইন বা পাস করা না হয় বা কোন কারনে ফেইল করে তখন এই এরর রেইজ হয়।

IOError: যখন I/O(ইনপুট/আউটপুট) অপারেশন যেমনঃ প্রিন্ট ফাংশন, বিল্ট-ইন open() ফাংশন কোন কারণে ঠিকমত এক্সিকিউট না হয় তখন এমন এরর দেখায়।

ImportError: অন্য কোন মডিউল বা মেথড ইম্পোর্ট না হলে ইম্পোর্ট-এরর মেসেজ দেখায়।

IndexError: ইটারেবল অবজেক্টের ইন্ডেক্সের মান উপাদান সংখ্যার চেয়ে বেশি হয়ে গেলে।

KeyError: দেখায় যদি এমন কোন 'কি' কল করা হয় যা ডিকশনারি তে এক্সিস্টিং 'কি' সেটের মধ্যে নেই ।

KeyboardInterrupt: প্রোগ্রাম রান করার সময় যদি কিবোর্ড থেকে অপ্রোজনীয় কি প্রেস (সাধারনত Ctrl+C) করা হয়।

NameError: যদি লোকাল বা গ্লোবাল কোন ভ্যারিয়েবলের নাম না পাওয়া যায়

OSError: যদি সিস্টেম-সম্পর্কিত কোন এরর থাকে

SyntaxError: যদি যথাযথ সিন্ট্যাক্স ব্যবহার না করা হয় কিংবা ভুল সিন্ট্যাক্স ব্যবহার করা হয়।

ValueError: ভ্যালু যদি ঠিক ভাবে ইনপুট দেয়া না হয়।

এক্সেপশন হ্যান্ডেলিং

আমরা যে এক্সেপশন গুলো নিয়ে আলোচনা করেছি এগুলো হ্যান্ডেল করার জন্য try-except স্টেট্মেন্ট আছে। হ্যান্ডেল মানে এই না যে এরর দেখাবে না। সাধারণত পাইথন আমাদের প্রকৃত যে এরর তার সাথে আরও অনেক ডেটা প্রিন্ট করে। পাশাপাশি অনেক সময় প্রকৃত এরর টাকে ট্রেস করা যায় না। এক্সেপশন হ্যান্ডেলিং প্রোগ্রামের প্রকৃত এরর নির্ণয় করা হয় এবং প্রোগ্রামকে ক্র্যাশ করা থেকে রক্ষা করে।

try-except

try ব্লকে মূল কোড লিখা হয় এবং সে কোড থেকে যে এরর আশার সম্ভাবনা আছে সেগুলো লিখা হয় except ব্লকে।

```
# import module sys to get the type of exception
import sys

randomList = ['a', 4, 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        ent = int(entry)
```

```
        r = 1/ent
        print("The result:",entry,"is",r,"\n")
    except:
        print("Oops!",sys.exc_info()[0],"occured.")
        print("Next entry.")
        print()
```

আউটপুটঃ

```
The entry is a
Oops! <class 'ValueError'> occured.
Next entry.

The entry is 4
The result: 4 is 0.25

The entry is 0
Oops! <class 'ZeroDivisionError'> occured.
Next entry.

The entry is 2
The result: 2 is 0.5
```

যদি try-except স্টেট্‌মেন্ট ব্যবহার করা না হতো তাহলে প্রথম ইটারেশনে একবার এরর দেখিয়ে প্রোগ্রাম ক্র্যাশ করত। আমরা বুঝতে পারতাম না কোন ডেটার জন্য কি এরর দেখাচ্ছে। এভাবে আমরা প্রতিটি এরর আলাদাভাবে হ্যান্ডেল করতে পারি।

```
my_dict = {"1": "Bangladesh"," 2": "Japan"," 3": "Canada"}
try:
    value = my_dict["4"]
except KeyError:
    print("No key of this name")
```

finally

ফাইনালি স্টেটমেন্ট try-except এর পরে ব্যবহার করা হয়। try বা except যেকোন স্টেটমেন্ট এক্সিকিউট হওয়ার পর অবশ্যই finally স্টেটমেন্ট এক্সিকিউট হবে।

```
my_list = [1,2,3,4,5]

try:
    print(my_list[4])
except IndexError:
    print("Index value exceeded")
finally:
    print("Whatever happens i will be executed")
```

আউটপুটঃ
5
Whatever happens i will be executed

try-except-else

try-except এর সাথে আরেকটি স্টেটমেন্ট ব্যবহার করা হয় "else"। যখন কোডে কোন এরর পায় না তখনই কেবল else স্টেটমেন্ট এক্সিকিউট হয়।

উদাহরণ

```
my_list = [1,2,3,4,5]

try:
    print(my_list[4])
except IndexError:
    print("Index value exceeded")
else:
    print("Excuted Successfully")
```

```
finally:
    print("Whatever happens i will be executed")
```

এখানে যদি ইন্ডেক্স ভ্যালু চেঞ্জ করে 6 করে দেই তাহলে else স্টেটমেন্ট এক্সিকিউট হবে না।

অধ্যায়ঃ ৭

অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং ইন পাইথন

পাইথন একটি অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং ল্যাংগুয়েজ। প্রকৃতপক্ষে পাইথনে সবকিছুই(ডেটা,লিস্ট,সেট,টুপল,ইন্টিজার,ফাংশন...) একেকটি অবজেক্ট। এই অধ্যায়ে আমরা ক্লাস, অবজেক্ট ও অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং এর ভিত্তি গুলো নিয়ে আলোচনা করব।

ক্লাস (class)

ক্লাস হচ্ছে একটি প্রোটোটাইপ যা একটি অবজেক্টের আচরণ নির্ধারণ করে। একটি ক্লাসের অবজেক্ট গুলো কেমন হবে বা তাদের কেমন বৈশিষ্ট্য থাকবে তা একটি ক্লাসে ডিফাইন করা থাকে। অর্থাৎ ক্লাস হচ্ছে একটি ব্লু-প্রিন্ট যার প্যাটার্ন অনুযায়ী সব গুলো অবজেক্ট তৈরি হয়।
ধরা যাক, আমাদের একটা বাড়ি বানানোর নকশা আছে। এখন ঐ নকশা দেখে আমরা যতগুলো বাড়ি বানাবো সব গুলো হবে সেই নকশার একেকটা অবজেক্ট এবং সেই নকশাটি হবে এদের ক্লাস।
পাইথনের ক্লাস হচ্ছে C++ এবং Modula-3 এর ক্লাস মেকানিজমের সংমিশ্রণ।পাইথনে ক্লাস ডিফাইন করার জন্য "class" কি-ওয়ার্ড ব্যবহার করা হয়।

ক্লাস এক্সপ্রেশনঃ

```
class className:
    'Optional class documentation string'
    class_suite
```

ক্লাসের ডকুমেন্টেশন স্ট্রিং কে চাইলে className.__doc__ দিয়ে এক্সেস করা যায়। class_suite ব্লকে ক্লাসের সব কম্পোনেন্ট স্টেটমেন্টস লিখা থাকে।

অবজেক্ট(object)

কোন ক্লাসের অবজেক্ট হল সেই ক্লাসের সকল প্রপার্টি ধারন করা একটি ভ্যারিয়েবল। অবজেক্ট কে ইন্সট্যান্স ও বলা হয়। প্রত্যেকটি অবজেক্ট ই একটি নির্দিষ্ট টাইপের। ক্লাসের অবজেক্ট-কে দুই ভাবে তৈরি করা যায়। অ্যাট্রিবিউট

রেফারেন্স ও ইন্সটেন্সিয়েশন। অ্যাট্রিবিউট হল কোন ক্লাসের ভেতরে থাকে এর প্রোপার্টিস, মেথডস ইত্যাদি।

অ্যাট্রিবিউট রেফারেন্স এর ক্ষেত্রে আমরা যখন কোন ক্লাসের অবজেক্ট তৈরি করি তখন সেটি একটি ব্ল্যাংক অবজেক্ট থাকে যতক্ষন পর্যন্ত না আমরা সে ক্লাসের কোন ফাংশন / মেথড / অ্যাট্রিবিউট কে কল না করি।

উদাহরণঃ

```
>>>class Home()
        length = 6
        width = 5
        area = length*width
>>>new_home = Home()    is an object of 'Home' class

>>>print(new_home)
<__main__.Home object at 0x711b8384a8>
>>>print(new_home.area)
30
>>>print(new_home.length)
6
```

+প্রথমে যখন আমরা শুধুমাত্র (new_home) প্রিন্ট করেছি তখন আমাদের অবজেক্টটির এড্রেস আউটপুট দিয়েছে। এরপর আমরা ওই ক্লাসের বিভিন্ন অ্যাট্রিবিউট এক্সেস করেছি। আমরা এতক্ষণ যে অ্যাট্রিবিউট গুলো নিয়ে কাজ করলাম সেগুলো ছিল ডেটা মেম্বার কিন্তু প্রয়োজনে ক্লাসের ভেতর ফাংশনও (মেথড) ডিক্লেয়ার করা যায়।

```
class Student:
    name = "John"
    roll = 67
    def admission():
        print("Student has been admitted")
```

এখানে আমরা একটি স্টুডেন্ট ক্লাস তৈরি করেছি, যার ক্লাস ভ্যারিয়েবল বা ডেটা মেম্বার হচ্ছে name এবং roll এবং একটি মেথড তৈরি করলাম admission().
এখন আমরা এই ক্লাসের প্রপার্টি গুলো ব্যবহার করতে পারব।

```
>>>Student.name
'John'
>>>Student.roll
67
>>>Student.admission()
Student has been admitted
```

আমরা এখনো এই ক্লাসকে ইন্সট্যানশিয়েট করিনি অর্থাৎ ক্লাসের কোন অবজেক্ট তৈরি করি নি। এবার দেখা যাক আমরা একটা ইন্সট্যান্ট বা অবজেক্ট তৈরি করলে কি হয়।

```
>>> new_student = Student()
>>> new_student
<__main__. Student object at 0x000001E8BF8D0BA8>
>>> type(new_student)
<class '_main_.Student'>
```

এই অবজেক্ট দিয়ে আমরা Student ক্লাসের অ্যাট্রিবিউট এবং মেথডস গুলো ব্যবহার করার চেষ্টা করি।

```
>>>new_student.name = "Don"
>>>new_student.roll = 94
```

চল দেখি এই ছাত্র কে ভর্তি করানো যায় কিনা মানে admission() মেথড কে কল করা যায় কিনা।

```
>>>new_student.admission()
```

তখন ই পাইথন আমাদের এমন একটি এরর দেখাবে।
"TypeError: admission() takes 0 positional arguments but 1 was given"

এখন প্রশ্ন হচ্ছে এই আর্গুমেন্ট আসলো কোথা থেকে আমরা তো কোন আর্গুমেন্ট ই পাঠাই নি।

এর পেছনের ঘটনা হল, আমরা যখন কোন ক্লাসের অবজেক্ট দিয়ে কোন মেথড কে কল করি তখন সেই মেথডের প্রথম আর্গুমেন্ট হিসেবে সেই অবজেক্ট টি পাঠানো হয়। এখন আমাদের কি করা উচিত। একটি প্যারামিটার দেয়া উচিত । তুমি চাইলে যে কোন কিছুই দিতে পারো। তবে পাইথনে স্ট্যান্ডার্ড হিসেবে self ব্যবহার করা হয়। অর্থাৎ self কোন কি ওয়ার্ড নয়।পাইথন ক্লাসের কোন ফাংশন যখন কোন অবজেক্ট দ্বারা কল করা হয় তখন সেটি মেথড হয়ে যায়। এখানে ফাংশন এবং মেথডের তফাৎ টা হচ্ছে মেথড ফাংশনের চাইতে একটি আর্গুমেন্ট বেশি নেয় এবং সেটি হচ্ছে সেই অবজেক্ট।ইন্টারপ্রেটারে ক্লাস এবং অবজেক্ট থেকে আলাদা আলাদা করে একই ফাংশনের টাইপ যাচাই করলেই ভিন্নতা দেখতে পাবে।

```
>>>type(Student.admission)
<class 'function'>
>>>type(new_student.admission)
<class 'method'>
```

মেথডের এক্সপ্রেশন

Class.method(object, *args)

এই ব্যপারটাই ঘটে যখন আমরা কোন অবজেক্ট থেকে কোন মেথড কে কল করি।

object.method(*args)

অর্থাৎ, কোন একটা ক্লাসের মেথডের প্রথম প্যারামিটার হচ্ছে সেই অবজেক্ট নিজেই।
কোন অবজেক্টে ক্লাস অ্যাট্রিবিউট কল করার সময় যদি সে অ্যাট্রিবিউটি একটি ফাংশন(মেথড) হয় তখন যে আর্গুমেন্ট গুলো পাস করা হয় সেগুলো সে অবজেক্টের মধ্যে অ্যাসাইন হয় না যথক্ষণ না আমরা self ব্যবহার করে তার মধ্যে ভ্যালু টা পাস করি। খুব গোল মেলে লাগছে নিশ্চয়ই!
উদাহরণের সাহায্যে বোঝার চেষ্টা করি…

```
>>>class Student:
        def admission(self,name,roll):
            self.name = name
            return self.name, roll

>>>new_student = Student()
>>>new_student.admission("John",67)
'John', 67
>>>new_student.name
John
>>>new_student.roll
Traceback (most recent call last):
AttributeError: 'Student' object has no attribute 'roll'
```

আমরা একটি Student ক্লাস তৈরি করেছি এবং new_student নামে তার একটি অবজেক্ট তৈরি করেছি।
new_student অবজেক্ট এর সাথে আমরা যখন Student ক্লাসের admission() মেথড কে কল করছি তখন এর প্যারামিটার এ name এবং roll আর্গুমেন্ট পাস করছি।
admission() মেথডের দিকে খেয়াল করলে দেখব আমরা name আর্গুমেন্ট টি কে self.name এর ভেতর রাখছি। যেহেতু self হচ্ছে new_student অবজেক্ট নিজেই তার মানে আমরা name আর্গুমেন্ট এর ভ্যালুকে new_student অবজেক্টের name হিসেবে অ্যাসাইন করছি। কিন্তু roll আর্গুমেন্ট এর ক্ষেত্রে তা করিনি।

তাই আমরা যখন new_student থেকে admission() মেথডের আর্গুমেন্ট গুলো পাস করছি তখন সেগুলো ঠিকই পাস হচ্ছে, কিন্তু শুধু মাত্র name আর্গুমেন্টের ভ্যালু টা ই সে অবজেক্টের মধ্যে অ্যাসাইন হচ্ছে। তাই যখন new_student.name কুয়েরি চালাই তখন পাইথন আমাদের অবজেক্টটির name আর্গুমেন্ট এর ভ্যালু দেখাচ্ছে কিন্তু যখন
new_student.roll জানতে চাচ্ছি তখন পাইথন আমাদের বলছে।

" Student ক্লাসের অবজেক্টের roll নামে কোন অ্যাট্রিবিউট নেই। "

এর কারণ আমরা roll আর্গুমেন্ট পাস করেছি ঠিকই কিন্তু সে ভ্যালু টাকে অবজেক্টের roll হিসেবে অ্যাসাইন করে দেই নি।

ক্লাস ভ্যারিয়েবল

ক্লাসের যে ভ্যারিয়েবল গুলো ওই ক্লাসের সকল অবজেক্ট ব্যবহার করতে পারে সেগুলোই হল ক্লাস ভ্যারিয়েবল।
ক্লাস ভ্যারিয়েবল ক্লাসে ভেতরে ডিফাইন করা হলে ও যেকোন মেথডের বাহিরে ডিফাইন করা হয়।

ইন্সটেন্স ভ্যারিয়েবল

যে ভ্যারিয়েবল ক্লাসের কোন মেথডের ভেতর ডিফাইন করা হয় তাকে ইন্সট্যান্স ভ্যারিয়েবল বলে।

ডেটা মেম্বার

ক্লাস ভ্যারিয়েবল এবং ইন্সট্যান্স ভ্যারিয়েবল কে ওই ক্লাসের ডেটা মেম্বার বলে।

অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং এর ভিত্তি

ইনহেরিটেন্স (Inheritence)

অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং এর সবচেয়ে গুরুত্বপূর্ণ বিষয়টি হচ্ছে ইনহেরি - টেন্স। ইনহেরিটেন্স অর্থ উত্তরাধিকার। প্রোগ্রামিং এ একটি ক্লাসের ডেটা অন্য ক্লাস উত্তরাধিকার সূত্রে ব্যবহার করার পদ্ধতিকে ইনহেরিটেন্স বলে। যে ক্লাসের ডেটা ব্যবহার করা হয় তাকে প্যারেন্ট ক্লাস বা বেজ ক্লাস বলে। আর যে ক্লাস ডেটা ব্যবহার করে তাকে চাইল্ড ক্লাস বা ডিরাইভড ক্লাস বলে। ইনহেরিটেন্স প্রোগ্রামে রিডান্ডেন্সি কমায়, প্রোগ্রামকে লাইট ওয়েট করে।

```
class <parent_class>:
    <class_suite>

class <child_class>(parent_class):
    <class_suite>
```

উদাহরণঃ

```
class Parent:
   def assets(self):
      print("Assets of parents are inherited by child")

class Child(Parent):
   pass



child1 = Child()
child1.assests()
```

আউটপুটঃ

Assets of parents are inherited by

child

এখানে আমরা Child ক্লাসে কোন মেথড ব্যবহার করি নি। তবুও আমরা Parent ক্লাসের মেথড ব্যবহার করতে পারছি। কারণ এখানে Parent ক্লাসের প্রপার্টি গুলো Child ক্লাস ইনহেরিট করেছে।

মাল্টিপল ইনহেরিটেন্স

একটি ক্লাস একই সাথে একাধিক ক্লাস কে ইনহেরিট করতে পারে। সেক্ষেত্রে ডিরাইভড ক্লাসের প্যারেনথিসিস এর ভেতর প্রতিটা বেজ ক্লাসের নাম উল্ল্যেখ কর দিতে হয়।

```
class <parent_class_one>:
    <class_suite>

class <parent_class_two>:
    <class_suite>

class <child_class>(<parent_class_one>,<parent_class_two>):
    <class_suite>
```

একে মাল্টিপল ইনহেরিটেন্স বলে।

উদাহরণঃ

```
class Father:
        def hero(self):
                print( "Every father is a hero")

class Mother:
        def warrior(self):
```

```
                print("Every mother is a warrior")

class Child(Father,Mother):
        def life(self):
                print("Child's life is inherited from Father and Mother")

we = Child()
we.life()
we.hero()
we.warrior()
```

আউটপুটঃ

Child's life is inherited from Father and Mother

Every father is a hero

Every mother is a warrior

মাল্টিলেভেল ইনহেরিটেন্স

যদি একটি ডিরাইভড ক্লাসের ও আরও ডিরাইভড ক্লাস থাকে তখন একে মাল্টি লেভেল ইনহেরিটেন্স বলে।

```
class A:
   <class_suite>

class B(A):
  <class_suite>

class C(B):
   <class_suite>
```

উদাহরণঃ

```
class Grand:
        def ancestor(self):
                print( "I am the Grand Class")

class Parent(Grand):
        def second_generation(self):
                print("I am inherited from Grand Class")

class Child(Parent):
        def new_generation(self):
                print("We are third generation")

we = Child()
we.new_generation()
we.second_generation()
we.ancestor()
```

কোন ক্লাস কোন ক্লাসের চাইল্ড ক্লাস বা কোনটি কোন ক্লাসের অবজেক্ট তা চেক করার জন্য issubclass() ও isinstance( ) দুটো বিল্ট ইন ফাংশন ব্যবহার করা হয়।

isinstance( ) ফাংশনটি দুটো প্যারামিটার নেয়। প্রথমটি হল অবজেক্ট,দ্বিতীয়টি একটি ক্লাস এবং এটি বুলিয়ান ভ্যালু রিটার্ন করে। যদি অবজেক্ট টি ওই ক্লাসের হয় তাহলে True অন্যথায় False রিটার্ন করে।

```
>>> isinstance(we,Child)
True
>>> isinstance(we,Parent)
True
```

Issubclass() মেথড ব্যবহার করা হয়, ক্লাস ইনহেরিটেন্স চেক করার জন্য।

```
>>> issubclass(Child,Parent)
True
```

```
>>> issubclass(Parent,Child)
False
```

পলিমরফিজম (Polymorphism)

পলি অর্থ একাধিক এবং মরফিজম অর্থ কাঠামো। কোন কিছু কে একাধিক কাঠামো দেয়া কে পলিমরফিজম বলে। প্রোগ্রামিং এ পলিমরফিজম বলতে বোঝায় একই ফাংশন কে ভিন্ন ভাবে ব্যবহার করা বা ভিন্ন আচরণ করা।

মনে কর + অপারেটর বা এর ফাংশন __add__() ইন্টিজার যোগ করার সময় এক রকম কাজ করে আবার স্ট্রিং যোগ করার সময় অন্য রকম

```
>>>4 + 5
9
>>> '4' + '5'
'45'
>>> a = 4
>>> a.__add__(5)
9
>>> a = '4'
>>> b = '5'
>>> a.__add__(b)
'45'
```

পাইথনে দু'ভাবে পলিমরফিজম পদ্ধতি ব্যবহার করা যায়। মেথড ওভাররাইডিং ও মেথড ওভারলোডিং।

মেথড ওভাররাইডিং

কোন মেথড যদি প্যারেন্ট ক্লাসে ইম্পলিমেন্টেশন করা থাকে। তারপরও চাইল্ড ক্লাসে নিজের মত করে সে মেথড কে পরিবর্তন করে নেয়াকে মেথড ওভার রাইডিং বলে।

```
class Animal:
    def sound(self):
        print("Each animal has an unique sound")

class Dog(Animal):
    def sound(self):
        print("Dog Barks")

puppy = Dog()
puppy.sound()
#output
Dog Barks
```

আমরা প্যারেন্ট ক্লাস Animal এর sound() মেথড কে ইমপ্লিমেন্ট করেছি। কিন্তু চাইল্ড ক্লাসে Dog মধ্যে আবার সে মেথড করে ওই ক্লাসের উপযোগী করে পরিবর্তন করে নিয়েছি এই প্রক্রিয়া কেই মেথড ওভাররাইডিং বলে।

মেথড ওভারলোডিং

একটি মেথড যখন ভিন্ন ভিন্ন কন্ডিশনে ভিন্ন ভিন্ন ভ্যালু রিটার্ন করে তখন তাকে মেথড ওভারলোডিং বলে।

```
class Student:
    def __init__(self,name,roll,*args):
        self.name = name
        self.roll = roll


    def cgpa(self, grade=None):
        self.grade = grade
        if grade == None:
            print(f"Ow!{self.name} \nYou are backbencher right?")
        elif grade >= 3.5:
```

```
        print(f"Welcome TOPPER!! \nRoll: {self.roll}Your CGPA is {self.grade}")
    else:
        print(f"Welcome {self.name} \nRoll: {self.roll} \nYour CGPA is
{self.grade}")
me = Student('Ratul',94)
ayub = Student("Khan",98)
ayub.cgpa(3.5)
rashed = Student('Rashed',109)
rashed.cgpa(3)
```

আউটপুটঃ

```
Ow!Ratul
You are backbencher right?

Welcome TOPPER!!
Roll: 98
Your CGPA is 3.5

Welcome Rashed
Roll: 109
Your CGPA is 3
```

এখানে আমরা Student একটি ক্লাস তৈরি করেছি এবং __init__() একটি কন্সট্রাক্টর ডিক্লেয়ার করেছি । এখন প্রত্যেক ছাত্রের ই নাম ও রোল থাকে। কিন্তু সব ছাত্রে হাই-সিজি থাকেনা। শিক্ষক রা ও সবার সাথে সমান আচরন করেন না। একদম পলিমরফিজমের জীবন্ত উদাহরণ। এখানে ও ঠিক একই কাজ হচ্ছে। আমাদের

Student ক্লাসের cgpa মেথডটি একেক অবজেক্টের সাথে একেক রকম আচরণ করবে।

তোমার যদি grade == None হয় তাহলে এক কথা বলবে,grade যদি 3.5 এর নিচে হয় তাহলে এক রকম আউটপুট দিবে ar যদি গড়পত্তা গ্রেড হয় তাহলে অন্যরকম আচরণ করছে। আশাকরি, পলি মরফিজম বুঝতে পেরেছ।

এনক্যাপসুলেশন (Encapsulation)

এনক্যাপসুলেশন অর্থ কোন ক্লাসের প্রপার্টি এবং ডেটা গুলো একসাথে প্যাকেট করে ফেলা। অন্য কোন ক্লাসের অব্জেক্ট থেকে কোন ক্লাসের ডেটা লুকোনো বা সরাসরি ব্যবহার থেকে বিরত রাখা ই এনক্যাপসুলেশন। অর্থাৎ কোন ক্লাসের ডেটা বা মেথড কে প্রাইভেট করে রাখার প্রক্রিয়া-ই এনক্যাপসুলেশন । প্রাইভেট অ্যাট্রিবিউট ক্লাস নিজে যেকোন সময় এক্সেস করতে পারে তবে অন্য কোন ক্লাস সরাসরি এক্সেস করতে পারেনা বা পরিবর্তন করতে পারে না। এনক্যাপসুলেশন ক্লাসের ডেটার নিরাপত্তা বাড়ায়।

পাইথন ও অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং এর এনক্যাপসুলেশনকে এন্টারটেইন করে না যদিও পাইথনে ডেটা প্রাইভেট করার সিন্ট্যাক্স আছে।

- একটি আইডেন্টিফায়ার যদি একটি আন্ডারস্কোর দিয়ে শুরু হয় তার মানে হচ্ছে আইডেন্টিফায়ার টি প্রাইভেট তবে চাইলে সহজেই এক্সেস করা যাবে।
- কোন আইডেন্টিফায়ার যদি দুটি আন্ডারস্কোর দিয়ে শুরু হয় তাহলে আইডেন্টিফায়ার টি প্রাইভেট তবে এক্সেস করা একটু কঠিন।

উদাহরণঃ

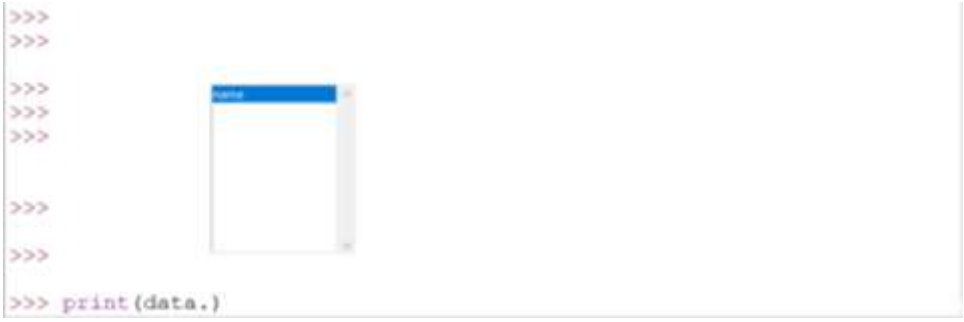
```
>>> class Person:
        def __init__(self):
                self.name = 'name of person'
                self._age = '20'
                self.__salary = '50'
```

```
>>> data = Person()
>>> print(data.name)
Ratul
>>> print(data._age)
20
```

আমরা একটা Person ক্লাস তৈরি করেছি। এবং __init__() মেথডের তিনটি ভ্যারিয়েবলের ভেতরে name একটি পাব্লিক ভ্যারিয়েবল _age প্রাইভেট, __salary আরও বেশি প্রাইভেট ভ্যারিয়েবল

```
>>> print(data.__salary)
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    print(data.__salary)
AttributeError: 'Person' object has no attribute '__salary'
```

__salary এতই প্রাইভেট যে পাইথন আমাদের বলছে যে এ নামে কোন অ্যাট্রিবিউটই নেই। কিন্তু _age তো ঠিকই আউটপুট দেখাচ্ছে। একে তো প্রাইভেট মনে হচ্ছে না। এটি যে প্রাইভেট তা অনুভব করতে পারবে যখন ইন্ট্রারপ্রেটারে _age এক্সেস করতে যাবে।



পাইথন ইন্টারপ্রেটার আমাদের শুধু মাত্র data অবজেক্টের একটি অ্যাট্রিবিউট সাজেস্ট করছে। যেন তার আর কোন অ্যাট্রিবিউট ই নেই!এই প্রাইভেট অ্যাট্রিবিউট

গুলো পরোক্ষ ভাবে এক্সেস বা পরিবর্তন করতে পাইথন getter এবং setter ব্যবহার করে।

```
class Person:
        def __init__(self):
                self.__salary = 50
        def getSalary(self):
                print(self.__salary)
        def setSalary(self,salary):
                self.__salary = salary

>>> data = Person()
>>> data.getSalary()
50                          #initial value is default and accessible
>>> data.setSalary(60)      # manipulating data
>>> data.getSalary()
60                          # data manipulation is been done
```

আমরা প্রথমে Person ক্লাসের দুটো মেথড ডিক্লেয়ার করেছি। getSalary() মেথড টি আমাদের __salary ডেটা পাঠাবে এবং setSalary() মেথডটি তে আমরা __salary ভ্যারিয়েবলে নতুন করে পাঠানো salary আর্গুমেন্টের ডেটা রেখে দিচ্ছি যা __salary ভ্যারিয়েবলের ডিফল্ট ডেটা পরিবর্তন করছে।

এতক্ষণ আমরা প্রাইভেট ভ্যারিয়েবল কিভাবে ডিক্লেয়ার করে এক্সেস করে তা দেখলাম। এবার আমরা প্রাইভেট মেথডস নিয়ে আলোচনা করব। পাইথনে প্রাইভেট মেথড ডিক্লেয়ার করার জন্য নামের শুরুতে দুটো আন্ডারস্কোর দিতে হয়।

```
__privateMethod():

        pass
```

উদাহরণঃ

```
>>> class Person:
        def publicMethod(self):
                return "public data"
        def __privateMethod(self):
                return "private data"


>>> data = Person()
>>> data.publicMethod()
'public data'
>>> data.__privateMethod()
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    data.__privateMethod()
AttributeError: 'Person' object has no attribute '__privateMethod'
```

প্রাইভেট মেথড একাধিক ভাবে এক্সেস করা যায়। সাধারণত এভাবে কল করা যায়…

object._className.__privateMethodName()

```
>>> data._Person__privateMethod()
'private data'
```

এ মেথড পাইথন নিজেই তৈরি করে নেয়। ইন্টারপ্রেটারে ক্লাসের ডিরেক্টরি দেখলে প্রথমেই মেথড দেখবে।

```
>>> dir(data)
['_Person__privateMethod', '__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'publicMethod']
```

এছাড়া ও ক্লাসের ভেতরে পাব্লিক মেথড লিখে সেখানে প্রাইভেট মেথড কে কল করে পরোক্ষ ভাবে প্রাইভেট মেথড এক্সেস করা যায়।

```
>>> class A:
        def __private(self):
                return "I won't say anything"
        def public(self):
                return self.__private()
>>> obj = A()
>>> obj.public()
"I won't say anything"
```

পাইথনে আন্ডারস্কোর ব্যবহার করা ছাড়া `accessify` লাইব্রেরির `@private`, `@protected` ডেকোরেটরস ব্যবহার করে মেথড প্রাইভেট বা প্রোটেক্টেড করা যায়। এদের এক্সেস মডিফাইয়ারস বলে।

অ্যাবস্ট্রাকশন (Abstraction)

অর্থ অবাস্তব। প্রোগ্রামিং এ অ্যাবস্ট্রাকশন বলতে বোঝায়, প্রোগ্রামের কমপ্লেক্সিটি কে আড়াল করে ব্যবহার কারির সামনে শুধুমাত্র গুরুত্বপূর্ণ ফিচার গুলো উপস্থাপন করা। যেকোন সিস্টেমের পেছনের কাজ গুলো সাধারণ ব্যবহারকারীর জানার প্রয়োজন থাকে না। পেছনের কঠিন সব মেকানিজম লুকিয়ে ব্যবহারকারীর জন্য সহজতর উপায়ে সিস্টেম টাকে উন্মুক্ত করে দেওয়ার পদ্ধতি ই হল অ্যাবস্ট্রাকশন এখানে ব্যবহারকারী এমন একটা ইন্টারফেসের সাথে কমিনিকেশন করে যার মূল কাজ গুলো প্রকৃত পক্ষে হচ্ছে অন্য কোন নিয়মে-অন্য কোথাও। অর্থাৎ, ব্যবহারকারী একটা অবাস্তব মাধ্যমের সাহায্য সিস্টেম টা ব্যবহার করছে। তাই একে অ্যাবস্ট্রাকশন বলে।

পাইথনে অ্যাবস্ট্রাকশন মেনে কাজ করে জন্য অ্যাবস্ট্রাক্ট ক্লাস ও ইন্টারফেস ব্যবহার করতে হয়।

অ্যাবস্ট্রাক্ট ক্লাস হল এক বা একাধিক অ্যাবস্ট্রাক্ট মেথড নিয়ে গঠিত ক্লাস যার মেথড গুলো সম্পূর্ন রুপে ইমপ্লিমেন্টেড থাকে না। অ্যাবস্ট্রাক্ট ক্লাসের মেথড ইমপ্লিমেন্টেশনের জন্য ওই ক্লাসের সাব ক্লাস তৈরি করতে হয়। পাইথনে বিল্ট-ইন কোন অ্যাবস্ট্রাক্ট বেজ ক্লাস নেই তবে (Abstract Base Classes) abc মডিউলের মাধ্যমে প্রোগ্রামে অ্যাবস্ট্রাকশন ব্যবহার করা যায়।

```
from abc import ABC, abstractmethod

class Month (ABC):
    @abstractmethod
    def weeks(self):
        pass
    @abstractmethod
    def days(self):
        pass

class January(Month):
    def weeks(self):
        print("four weeks")
    def days(self):
        print("thirty one days")


first = January()
first.days()
first.weeks()
```

আমরা যদি অ্যাবস্ট্রাক্ট ক্লাসের days () মেথডটি January ক্লাসে ইমপ্লিমেন্ট না করি তাহলে আমাদের এরর দেখাবে। কারণ অ্যাবস্ট্রাক্ট ক্লাসের সবকটি মেথড ডিরাইভড অভাররাইড করা না হলে ডিরাইভড ক্লাসটির অবজেক্ট তৈরি করা যায় না। তাই আমরা এমন এরর পাব…

TypeError: Can't instantiate abstract class January with abstract methods days

তবে আমরা চাইলে আলাদাভাবে একাধিক ক্লাস ইমপ্লিমেন্ট করতে পারি।

```
class January(Month):
    def weeks(self):
        print("four weeks")

class JanuaryDays(January):
    def days(self):
        print("thirty one days")

first = JanuaryDays()
first.days()
first.weeks()
```

```
thirty one days
four weeks
```

আমরা অ্যাবস্ট্রাক্ট মেথড কে ও ইমপ্লিমেন্ট করতে পারি এবং সে মেথড কে ডিরাইভড ক্লাসের ভেতর ব্যবহার করার জন্য super() মেথড ব্যবহার করতে হয়। কেন করতে হয় সেটা পরে আলোচনা করা যাবে।

```
from abc import ABC,abstractmethod

class Month(ABC):
    @abstractmethod
    def weeks(self):
        pass
    @abstractmethod
    def days(self):
        print("thirty one days")
```

```
class January(Month):
    def weeks(self):
        print("four weeks")
    def days(self):
        super().days()
        print("now i can use my abstractmethod")


first = January()
first.days()
first.weeks()
```

```
thirty one days
now i can use my abstractmethod
four weeks
```

অ্যাবস্ট্রাক্ট ক্লাসের অবজেক্ট তৈরি করা যায় না। কারণ অ্যাবস্ট্রাক্ট ক্লাসের মেথড গুলো অসম্পূর্ণ। যদি কোন অবজেক্ট তৈরি করা হয় তাহলে প্রকৃতপক্ষে তার কোন অ্যাট্রিবিউট থাকছে না যা ব্যবহারযোগ্য।

```
a = Month()
TypeError: Can't instantiate abstract class Month with abstract methods
days, weeks
```

super() ফাংশন

আমরা যখন কোন ক্লাসের মেথড ওভারাইড করি তখন চাইলে ও সে ক্লাসের প্যারেন্ট ক্লাসের ওভারাইড করা মেথড টি কল করতে পারি না। super() ফাংশন ব্যবহার করে আমরা চাইল্ড ক্লাসে মেথড ওভারাইড করার পরও প্যারেন্ট ক্লাসের মেথড টি কে ও কল করতে পারি ও ব্যবহার করতে পারি।

```
class Parent:
    def main(self):
        print("Parent Class Function")

class Child(Parent):
    pass

obj = Child()
obj.main()
```

এই উদাহরনে আমরা চাইল্ড ক্লাসের যে অবজেক্ট তৈরি করেছি সেটি প্যারেন্ট ক্লাসের প্রপার্টি গুলো ইনহেরিট করছে। আমরা তাই প্যারেন্ট ক্লাসের মেইন ফাংশন টি কল করতে পারছি।

```
class Parent:
    def main(self):
        print("Parent Class Function")

class Child(Parent):
    def main(self):
        pass
obj = Child()
obj.main()
```

এবার আমরা প্যারেন্ট ক্লাসের মেইন ফাংশনটি চাইল্ড ক্লাসে ওভারাইড করেছি কিন্তু এর ভেতর কোন কিছুই নেই। তাই এখন প্রোগ্রামটি আমাদের কোন কিছুই দেখাবে না। কারন সে চাইল্ড ক্লাসের ফাংশন টি কে কল করছে। এখন আমরা দেখব super() ফাংশন ব্যবহার করে কিভাবে একই সাথে চাইল্ড ও প্যারেন্ট ক্লাসের একই নামের ফাংশন কল করা যায়।

```
class Parent:
    def main(self):
        print("Parent Class Function")

class Child(Parent):
    def main(self):
```

```
        print("Child Class Function")
        super().main()

obj = Child()
obj.main()
```

আউটপুট

Child Class Function

Parent Class Function

এখানে অবজেক্ট টি প্রধানত আমাদের চাইল্ড ক্লাসের main() কেই কল করেছে. সেই main() এর ভেতরে আবার super() ফাংশন প্যারেন্ট ক্লাসের main() কে কল করেছে।

মেটাক্লাস (MetaClass)

আমরা জানি, পাইথনের সবকিছুই অবজেক্ট।তাহলে ক্লাস নিজেও একটি অবজেক্ট। ক্লাস তাহলে কার অবজেক্ট? অন্য কোন হায়ার-লেভেল ক্লাসের? সেই হায়ার ক্লাস-ই হল মেটাক্লাস। কোন ক্লাসের ক্লাসকে মেটাক্লাস বলে।

অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং এর একটি রহস্যমূলক বিষয় হচ্ছে এi মেটাক্লাস। নিচের কোডগুলোর দিকে লক্ষ্য করো…

```
>>> class Main:
        pass

>>> print(Main)
<class '__main__.Main'>
>>> print(Main())
<__main__.Main object at 0x00000202857788D0>
>>> print(type(Main))
<class 'type'>
>>> print(type(Main()))
<class '__main__.Main'>
```

```
>>> Main2 = type("Main2",(),{})
>>> print(Main2())
<__main__.Main2 object at 0x0000020285729D68>
>>> print(type(Main2))
<class 'type'>
>>> print(type(Main2()))
<class '__main__.Main2'>
>>>
```

```
>>> Main2 = type('Main2',(),{})
>>> print(Main2())
<__main__.Main2 object at 0x0000025A09E89DD8>
>>> print(type(Main2))
<class 'type'>
>>> print(type(Main2()))
<class '__main__.Main2'>
>>>
```

আমরা দুই ভাবে ক্লাস ডিফাইন করেছি। class কিওয়ার্ড দ্বারা ও type ব্যবহার করে। type এর আর্গুমেন্ট গুলো হলো Name, (Parent Class), {dict}

খেয়াল করে দেখো class একটি টাইপ অবজেক্ট। **<class 'type'>** এবং type ও টাইপ অবজেক্ট।

```
>>> class A:
        pass

>>> type(A) == type(type)
True
>>>
```

type একটি মেটাক্লাস। type "C" ল্যাংগুয়েজে ইমপ্লিমেন্ট করা। দেখতে কাজ এক হলে ও type ও class সম্পূর্ণ এক না। class মিউটেবল, type মিউটেবল না।

type ব্যবহার করে আমরা একটি ক্লাস তৈরি করা দেখবঃ

```
>>> def add(self,a,b):
        return a+b

>>> def sub(self,a,b):
        return a-b

>>> maxim = 100
>>>
>>> klass = type('klass',(),{'add':add,'sub':sub,'maxim':maxim})
>>>
>>> obj = klass()
>>> obj.add(4,5)
9
>>> obj.maxim
100
>>>
```

আমাদের add, sub ফাংশন গুলো ক্লাসের বাহিরে হয়েও ক্লাসের অ্যাট্রিবিউট হিসেবে কাজ করছে কারণ আমরা ডিকশনারি তে সেগুলোকে অ্যাসাইন করেছি।

একই ভাবে type এর কিছু ফাংশন হচ্ছে __call__ , __new__ , __init__। যখন আমরা কোন ক্লাস তৈরি করি তখন সে type এর __call__ ফাংশন কে কল করে। কারন বাই-ডিফল্ট সব ক্লাসের প্যারেন্ট ক্লাস হলো type type. __call__ ফাংশন তখন __new__ , __init__ ফাংশন কে কল করে। এরপর আমরা চাইলে আমাদের ক্লাসে এসব মেথড কে ওভাররাইড করতে পারি। তবে যখন আমরা কোন মেটাক্লাস তৈরি করব তখন আমরা সে ক্লাসকে type ক্লাস থেকে ইনহেরিট করব।

```
class newMeta(type):
  def __init__(self,name,parent,dictionary):
    self.attribute = "I am a new attribute"


class A(metaclass = newMeta):
  pass
class B(A):
  pass

print(B.attribute)
```

মেটাক্লাস যে কাজ করে তা আমরা ইনহেরিটেন্স / ডেকোরেটরস ব্যবহার করে ও করতে পারি। মেটাক্লাস ব্যবহার করা নিয়ে অনেক মত বিভেদও আছে।

ইম্পোর্টিং (import)

পাইথনে বিভিন্ন কাজের জন্য বিভিন্ন প্রোগ্রাম আগে থেকেই লিখা আছে। এগুলোকে একেকটা প্যাকেজ বলে। প্যাকেজ এ এক বা একাধিক মডিউল থাকে। পাইথনে প্রায় ২ লক্ষের ও বেশি প্যাকেজ আছে। কেউ যদি কোন প্যাকেজ ব্যবহার করতে চায় তাহলে সে প্যাকেজটি তার প্রোগ্রামে আমদানি (import) করতে হয়। পাইথন স্ট্যান্ডার্ড লাইব্রেরি ছাড়া ও আরও অনেক প্যাকেজ আছে সেগুলো import করতে হলে প্রথমে সিস্টেমে সেটি ইন্সটল করে নিতে হয়। ইন্সটল করতে pip(packege installer python) টুল ব্যবহার করা হয়। ম্যানুয়ালি ও করা যায়।

আমরা ইতিমধ্যেই অনেক জায়গায় import কি ওয়ার্ড দেখেছি। যেমন...

```
import this
import math
```

এভাবে ইম্পোর্ট করলে মডিউলের মেথড ব্যবহারের সময় প্রতিটি মেথডের আগে ডট(.) দিয়ে মডিউলের নাম দিতে হয়। তা না হলে পাইথন সে ফাংশনকে চিনতে পারে না।

```
math.sqrt(3)
```

এই ঝামেলা এড়ানোর জন্য from কি-ওয়ার্ড ব্যবহার করা হয় যেখানে নির্দিষ্ট করে মেথডের নাম বলে দেয়া যায়।

```
from math import sqrt
```

একাধিক মেথড ইম্পোর্ট করতে কমা দিয়ে লিখতে হয়।

```
from math import sqrt, pi
```

একটি মডিউলের সব কিছু ইম্পোর্ট করতে হলে (*) সাইন ব্যবহার করা হয়।
যদিও এটি একটি "ব্যাড প্র্যাকটিস"

```
from math impor
```

অধ্যায় ৮

HTTP রিকোয়েস্ট

পাইথনের একটি মডিউল হচ্ছে "requests" মডিউল। এ মডিউলের কাজ হচ্ছে HTTP(hyper text transfer protocol) নিয়ে বিভিন্ন কাজ করা। ডেটা পাঠানো, ডেটা নিয়ে আসা ইত্যাদি। এ মডিউল নিয়ে কাজ করতে হলে আমাদের প্রথমে এটি ইন্সটল করে নিতে হবে। CMD তে নিচের কমান্ডটি দিলেই মডিউলটি ডাউনলোড হয়ে যাবে।

pip install requests

GET

প্রথমেই আমরা দেখবো কিভাবে কোন ওয়েবসাইট থেকে ডেটা নিয়ে আসতে হয়। বলে রাখা ভালো এই মেথড HTML কোড সহ ডেটা নিয়ে আসবে। নিচের উদাহরণ টি লক্ষ্য করো…

```
>>> import requests
>>> r = requests.get('https://facebook.com')
>>> print(r.content)
```

Squeezed text (3174 lines).

এখানে GET মেথডের ভেতর আমরা facebook এর লিংক দিয়েছি।এখন GET এর ডেটা গুলো মানে HTML কোড গুলো নিয়ে এসেছে। কিন্তু যখন তার content গুলো আমরা প্রিন্ট করতে গিয়েছি তখন এক অদ্ভুত আউটপুট দেখাচ্ছে। আসলে কোড গুলো এত বড় যে idle তা দেখাতে চাচ্ছে না। আমরা লিখাটাতে ডাবল ক্লিক করলে অবশ্য দেখাবে।

এখন আমরা যদি শুধু টেক্সট ডেটা দেখতে চাই তার জন্য আমরা r অবজেক্টের text প্রপার্টি ব্যবহার করতে পারি।

```
>>> print(r.text)
```

Squeezed text (170 lines).

POST

POST রিকোয়েস্ট হচ্ছে GET রিকোয়েস্টের বিপরীত। POST রিকোয়েস্টের কাজ হচ্ছে ডেটা পাঠানো যেখানে GET এর কাজ হচ্ছে ডেটা নিয়ে আসা। পাইথনে POST মেথড ব্যবহার করে আমরা POST রিকোয়েস্ট করতে পারি। POST মেথড নিয়ে কাজ করার জন্য আমরা httpbin নামক একটি ওয়েবসাইটের সাহায্য নিব। সে সাইটে আমাদের ডেটা পাঠানোর চেষ্টা করব।

```
>>> import requests
>>> p = {'username':'admin','password':'nothing'}
>>> r = requests.post('http://httpbin.org/post',data=p)
>>> print(r.text)
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "password": "nothing",
    "username": "admin"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "31",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.24.0",
    "X-Amzn-Trace-Id": "Root=1-5f9665db-73565a6f437b82fb3c6b9a49"
  },
  "json": null,
  "origin": "103.58.73.117",
  "url": "http://httpbin.org/post"
}
```

আমরা p ভ্যারিয়েবলের মাঝে একটি ডিকশনারি রেখেছি যেখানে আমাদের username ও password আছে। এখন এই ডেটা আমরা httpbin.org/post এই লিংকে পাঠাবো। post মেথডের ভেতর প্রথম আর্গুমেন্ট হিসেবে সাইটের লিংক ও data আর্গুমেন্টের ভ্যালু হিসেবে p ভ্যারিয়েবল পাঠানো হয়েছে। আমরা যখন টেক্সট প্রিন্ট করেছি সেখানে কিন্তু আমাদের username এবং password দেখতে পাচ্ছ।

Session & Cookies

HTTP একটি স্টেটলেস প্রোটোকল। Session এবং Cookies, HTTP কে স্টেটফুল করে । cookie সবসময় ক্লায়েন্ট-সাইডে থাকে। ব্রাউজার cookie হ্যান্ডেল করে। Session বেশিরভাগ সময় সার্ভারসাইডে থাকে তবে সবসময় না।

মনে করুন, আপনি একটা ই-কমার্স ওয়েবসাইটে গেলেন। গিয়ে সেখানে কিছু প্রোডাক্ট Cart এ রাখলেন। তারপর ভাবলেন, নাহ পরে কিনব। ওইভাবে রেখেই আপনি সাইটটি ক্লোজ করে দিলেন। কিন্তু পরে যখন আবার সে সাইটে ঢুকলেন দেখলেন Cart এ এড করা প্রোডাক্ট গুলো আগের মতই আছে। এ কাজ টা কিভাবে হয়? আপনি তো সাইট টি অফ করেছেন এমনকি আপনার কম্পিউটার ও হয়তো অফ করেছেন। এই কাজ টা হয় session ও কুকি্র মাধ্যমে। আপনি যা যা করেছিলেন সেগুলোর লাস্ট স্টেট সার্ভারে স্টোর থাকে আর তা আইডেন্টিফাই করার জন্য একটা cookieআপনার ওয়েব ব্রাউজারে সেভ থাকে যেখানে session আইডি থাকে। আপনি যখন পরের বার ওই সাইটে ভিজিট করেন তখন আপনার ব্রাউজার সেই cookie সার্ভারে পাঠায় এবং সার্ভার ওই session আইডির জন্য যে session আছে তা রেস্পন্স হিসেবে পাঠায়।

session ভিন্ন ভিন্ন ব্যবহারকারীর জন্য ডেটা সঞ্চয় করতে ব্যবহৃত হয় যা একটি session আইডি দ্বারা ট্র্যাক করা হয়। ডেটা সার্ভারে সংরক্ষণ করা হয় এবং session আইডি একটি কুকির মাধ্যমে ক্লায়েন্টকে সরবরাহ করা হয়।

cookie হল একটি ইউনিক ডেটা যা সার্ভার ইউজারের ওয়েব ব্রাউজারে পাঠায়। সার্ভারে ব্যবহারকারীর জন্য বিদ্যমান ডেটা পুনরুদ্ধার করতে session আইডি ব্যবহার করা হয়।

এখন আমরা দেখব কিভাবে আমরা session তৈরি করতে পারি ও সেশনে ডেটা স্টোর করতে পারি ও সেখান থেকে ডেটা নিয়ে আসতে পারি। Sessions তৈরি করার জন্য আমরা session অব্জেক্ট তৈরি করব।

```
import requests

sObj = requests.Session()
```

sObj একটি session অবজেক্ট এবং requests API এর যে মেথড গুলো আছে সেগুলো আমরা এই অবজেক্টের জন্যও ব্যবহার করতে পারবো। যে ডেটা গুলো আমরা সেশনের মাধ্যমে স্টোর করতে চাই সে ডেটা গুলো আমরা প্রথমে দুটো ডিকশনারির মধ্যে রাখবো।

username = {“username”: “arafat"}

password = {“password” : “356654”}

```
>>> import requests
>>> sObj = requests.Session()
>>> username = {'username':'arafat'}
>>> password = {'password':'354566'}
>>> cookieURL = 'https://httpbin.org/cookies/set'
>>> getcookie = 'https://httpbin.org/cookies'
>>>
>>> sObj.get(cookieURL, params= username)
<Response [200]>
>>> sObj.get(cookieURL, params= password)
<Response [200]>
>>> response = sObj.get(getcookie)
>>> print(response)
<Response [200]>
>>> print(response.text)
{
  "cookies": {
    "password": "354566",
    "username": "arafat"
  }
}
```

cookieURL টি একটি API endpoint. এটি আমরা ব্যবহার করব সার্ভারে একটি cookie তৈরি করার জন্য। যে ডেটা গুলো আমরা সার্ভারে রেখেছি সেগুলো ফিরে পাওয়ার জন্য আমরা getcookie এই endpoint টি ব্যবহার করব।

যেমনটা আমরা আগে জেনেছি, session অবজেক্টের (sObj) জন্য get রিকোয়েস্ট পাঠাতে আমরা get() মেথড ই ব্যবহার করব।

প্রথম get রিকোয়েস্টে আমরা username ডিকশনারিতে রাখা ডেটা স্টোর করেছি, দ্বিতীয় রিকোয়েস্টে আমরা password ডিকশনারির ডেটা স্টোর করেছি, এবং তৃতীয় রিকোয়েস্টে আমরা স্টোর করা ডেটা গুলো নিয়ে এসেছি।

এভাবে যতক্ষন পর্যন্ত session অবজেক্টের মেয়াদ থাকবে ততক্ষন আমরা একাধিক রিকোয়েস্টের মাধ্যমে ডেটা স্টোর ও করতে পারবো, আবার রিট্রিভ ও করতে পারবো।

```
>>> sObj.get(cookieURL,params={'age':'24'})
<Response [200]>
>>> response = sObj.get(getcookie)
>>> print(response.text)
{
  "cookies": {
    "age": "24",
    "password": "354566",
    "username": "arafat"
  }
}
```

এরপর আমরা response অবজেক্টের url, encoding, status_code, content, text, json ইত্যাদি অ্যাট্রিবিউটের ভ্যালু দেখতে পারি।

```
>>> print(response.url)
https://httpbin.org/cookies
>>> print(response.encoding)
None
>>> print(response.status_code)
200
>>> print(response.headers)
{'Date': 'Mon, 26 Oct 2020 06:19:03 GMT', 'Content-Type': 'applic
ation/json', 'Content-Length': '93', 'Connection': 'keep-alive',
'Server': 'gunicorn/19.9.0', 'Access-Control-Allow-Origin': '*',
'Access-Control-Allow-Credentials': 'true'}
>>> print(response.content)
b'{\n  "cookies": {\n    "age": "24", \n    "password": "354566",
\n    "username": "arafat"\n  }\n}\n'
>>> print(response.cookies)
<RequestsCookieJar[]>
>>> print(response.json)
<bound method Response.json of <Response [200]>>
```